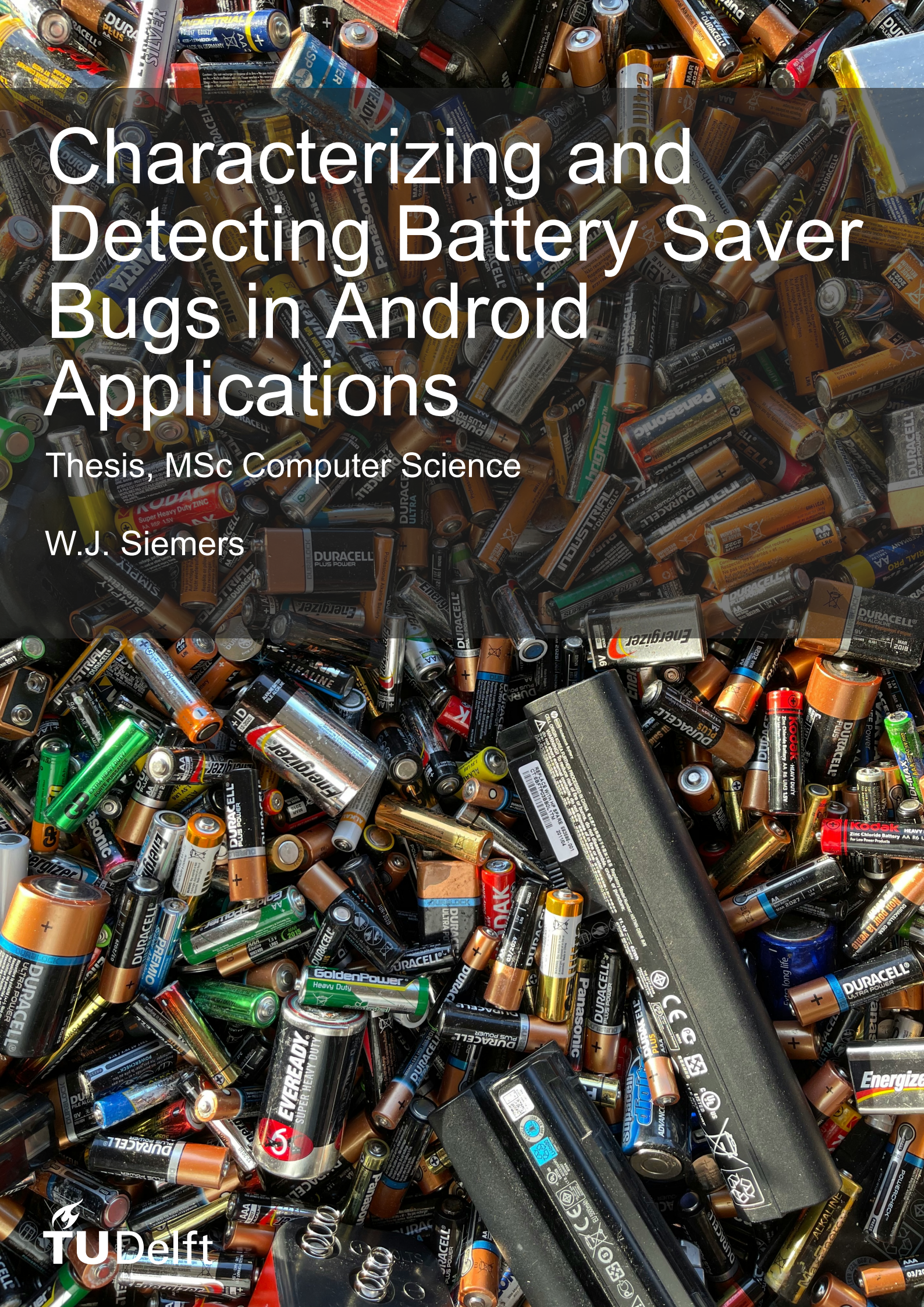

Characterizing and Detecting Battery Saver Bugs in Android Applications

Thesis, MSc Computer Science

W.J. Siemers

TU Delft

Characterizing and Detecting Battery Saver Bugs in Android Applications

Thesis, MSc Computer Science

by

W.J. Siemers

to obtain the degree of Master of Science

at Delft University of Technology,

to be defended publicly on Friday June 21st, 2024 at 15:00.

Student number:	4594002	
Project duration:	September 2023 – June, 2024	
Thesis committee:	Prof. dr. ir. A. Van Deursen[1]	Supervisor
	Dr. L. Miranda da Cruz[1]	Daily supervisor
	Dr. M. Fazzini[2]	External supervisor
	Dr. P. Pawełczak[1]	

[1]Delft University of Technology
[2]University of Minnesota

TU Delft

Preface

Before you lies my MSc thesis titled "Characterizing and Detecting Battery Saver Bugs in Android Applications". It has been written to fulfill the graduation requirements of the Computer Science program at Delft University of Technology.

Mobile software has interested me since at least 2012. I saw a kid quite like me describe on a television talk show how he learned to build mobile apps. I figured if he could do it, why couldn't I? I asked my parents for a book on Objective-C programming and started learning.

It's now 12 years later, and I am finishing up my formal education. Somehow, mobile software has remained interesting and fun to me. There is something personal and intimate about devices you carry with you all the time. Software is eating the world, as Marc Andreessen famously said. This small bite is my last academic contribution, but I am sure I will be back in the field.

My education has been an exercise in determination, although I've enjoyed large swathes of it. I've learned the value of discipline over motivation and that trying hard is often worth it simply because it's more fun.

I'd like to thank my mother, Ingrid, and honor the memory of my father, Ronald, for their unwavering support. My father, in particular, would be over the moon to see me graduate from his — and his father's — alma mater.

I'm grateful to my daily supervisors, Luís Miranda da Cruz and Mattia Fazzini, for their direct feedback, expert knowledge, and calm supervision. You've kept me sharp and helped me navigate the challenges of close to ten months of solo work.

W.J. Siemers
Delft, June 2024

Summary

Poor battery life is one of smartphone users' top frustrations about their devices. This fact, in combination with the limited supply of battery minerals, the working conditions of mining, and its environmental impact, has led to high interest in reducing smartphone energy consumption. Smartphone manufacturers have introduced power-saving features on their products and guide developers to use energy-efficient software engineering practices.

In literature, the increased awareness of the need to reduce energy consumption and reduce emissions has led to the birth of a *Green Software* research field. Focusing on mobile software, prior research has focused on quantifying energy use and finding instances of software using more energy than is reasonable for its intended purpose. However, the ubiquitous power-saving features of smartphones have hardly been studied until now. In particular, bugs stemming from Android's *Battery Saver* mode, a power-saving technology introduced in 2017 by Google, have not been studied at all. This thesis aims to address this research gap.

To do so, we first characterize these issues by systematically collecting documentation pages, bug reports, and forum questions relating to these bugs. We find 13 separate problems, most of which (9 out of 13) have considerable user impact. Additionally, most problems are reported multiple times independently (mean reporting frequency = 4.1) outside of Google documentation. Four of the problems have never been officially documented.

Driven by this characterization, we build a static analysis tool that detects one of the characterized issues. It builds a *Conditional Call Graph* to find invocations of the implicated Application Programming Interfaces (APIs) without asserting this API is available. The tool is fast and does not require source code to be available. It runs on the Java Virtual Machine for portability.

To evaluate the tool, we use a two-pronged approach. We first determine the ground truth for all 1,472 Google Play Store applications that are also available in the FDroid repository. We write a script to identify all suspicious API invocations, the bug candidates. We find and attempt to reproduce 178 of these bug candidates. We manually review all candidates to determine the ground truth. We evaluate our tool using the same data set. The tool reaches a precision of 0.911 and a recall of 0.911 and identifies 41 reproducible issues.

Lastly, we report the reproduced issues identified by the tool to the developers of the affected applications. To date, nine issues have been confirmed by the developers, and two issues have already been addressed.

Contents

List of Figures

List of Tables

Glossary

AOSP Android Open Source Project. The original, open-source version of Android, published by Google and often referred to as 'clean Android' since it contains no OEM modifications. 37, 38, 40

API Application Programming Interface. APIs enable different software components to communicate, allowing developers to access and use specific features or data from other programs in their own applications. ii, 1, 8, 10, 12–15, 19, 20, 22, 23, 26, 27, 29, 37–39, 41

CCG Conditional Call Graph. A specialized control-flow graph in which edges contain the conditions under which the control-flow transition is possible. v, 26, 27

GPS Global Positioning System. A satellite-based radio navigation system run by the United States. 4, 5

GPU Graphics Processing Unit. A specialized computer chip for executing graphics operations. 20

Java A high-level, object-oriented programming language. 24, 26, 27

OEM Original Equipment Manufacturer. In the context of Android smartphones, this term refers to companies that manufacture and market Android devices, such as Samsung and Huawei. vii, 14, 37, 40

OLED Organic light-emitting diode. A display technology often employed in modern smartphones. It has the notable property that power draw depends on the brightness of the color displayed. 5, 11

UI User Interface. Denotes the space where interactions between humans and machines occur. In modern smartphones, the primary user interface is usually the touchscreen. 9–11, 22

Wi-Fi A widely used family of network protocols for providing wireless Internet access. 5, 8

Wi-Fi triangulation A method of determining the user's location by measuring the strength of the signal (a proxy for distance) to known Wi-Fi access point locations. 5

1

Introduction

Smartphone users depend more than ever on their devices and want longer battery life. Battery sizes have increased, both in physical size and due to chemical improvements. However, workloads have also become heavier, including running AI models on-device and sophisticated camera processing. Therefore, to maintain battery life, meet customer expectations, and reduce emissions, general interest in mobile energy consumption has increased. While individual mobile devices are quite efficient and do not consume much power, the eight billion mobile phones in use are significant collectively. Furthermore, reducing the energy need of mobile devices allows for smaller batteries. Batteries often contain rare minerals mined in concerning conditions, so using a smaller battery is beneficial both socially [49] and environmentally [77].

Most smartphones worldwide run Google's Android operating system. Since 2010, a growing number of devices running Android have included a software 'Low Power Mode' for temporarily reducing battery consumption. This usually comes at the expense of lower performance and reduced background activity. However, this is a trade-off that users who want maximum battery life, such as while traveling away from a power source, are willing to make. Google later responded by launching a power-reducing feature called *Battery Saver*, built into Android itself. It restricts computationally expensive tasks and functionalities, such as background activity and user interface animations. In recent Android versions, Google has progressively introduced more battery-saving features to further increase efficiency and reduce the number of loopholes available for apps to avoid being restricted [18, 29].

Previous work in the area of mobile energy consumption has focused on quantifying energy use, i.e., energy testing, and on bugs causing excessive battery drain: so-called *energy bugs*. Energy testing work is not directly relevant to this work, since we do not measure energy consumption. Energy bug work includes studies on 'no-sleep' bugs [116], where applications keep the device from entering a power-saving sleep state and work on optimizing graphics drawing [48]. Our work differs from these studies because it specifically addresses bugs stemming from the use of the Battery Saver mode.

1.1. Problem Statement

While these features are useful, they also present opportunities for new bugs by restricting APIs that developers could previously count on always working. Power-saving features modify the system in subtle ways that are not always clearly documented. Developers may forget to enable these features manually during app testing, if at all possible, and testing every feature in multiple power-saving states

is highly cumbersome. Related topics, such as issues caused by other settings, have been widely studied. However, to our knowledge, this is the first study regarding Battery Saver bugs specifically.

These issues may significantly affect user experience, as we show in section 4.2. They may not be apparent during standard testing scenarios, highlighting the need for a comprehensive investigation into the interplay between power-saving features and app behavior. Additionally, common Android development tooling, such as Android Studio, does not provide warnings to developers for any of these issues.

1.2. Research Objectives

Having established that these power-saving features cause bugs, it appears that there is no research focused on characterizing and preventing these issues in practice. This thesis intends to fill that gap by providing a better understanding of the failure modes of Android apps under power restriction regimes and offers potential solutions to these failures. To the best extent of our knowledge, it is the first study into the topic of bugs caused by Android power-saving features.

More explicitly, this thesis aims to:

1. Provide a systematic characterization of issues stemming from the use of the Battery Saver functionality on Android devices
2. Find the issues mentioned in Objective 1 in a representative set of real-world applications
3. Aid developers by providing tooling to identify and resolve these issues in their applications

1.3. Methodology Overview

A rigorous characterization methodology was designed to achieve the stated research objectives. We first systematically determine keywords to find developer reports of problems surrounding Battery Saver. We query documentation and well-known public databases, including Stack Overflow and GitHub, and categorize and characterize the issues found.

We then develop a tool to address one of the identified issues with high user impact and a feasible fix. We evaluate the tool on a large base of public Android applications and manually reproduce all identified issues to determine its accuracy. We report the reproduced issues to developers to assess the usefulness of the bug reports.

1.4. Scope

We limit the scope of this research to smartphone devices running Android. Android is the world's most-used smartphone operating system and has a large third-party app ecosystem. The other major mobile operating system, Apple's iOS, also includes a power-saving mode [114], but since the iOS platform is more closed-off than Android, it is harder to develop tools to identify bugs automatically. Additionally, iOS disallows some of the functionalities of Android, such as extensive background processing, regardless of battery-saving mode, so there may be less surface area for bugs.

We only consider bugs appearing while running stock Android, such as the version of Android on Google Pixel devices and the Android emulator shipping with Android Studio. This has two main reasons: 1. Android implementations are diverse, and manufacturers have implemented many custom battery-saving mechanisms that may or may not be implemented by other manufacturers. 2. To analyze apps in a scalable way, using the Android Emulator instead of physical devices is

preferable so we can run multiple apps at the same time or quickly refer to different versions of a single app.

1.5. Outline

This thesis is organized as follows:

- In Chapter 2, the importance of this research is motivated, and relevant background information is presented.
- In Chapter 3, related work is considered and contrasted to this work.
- In Chapter 4, the methodology of this thesis is laid out.
- In Chapter 5, a systematic characterization of the identified issues is presented.
- In Chapter 6, a tool that automatically detects one identified issue in real-world apps is presented.
- In Chapter 7, the evaluation process for the tool is described.
- In Chapter 8, the results of evaluating the tool are presented.
- In Chapter 9, the findings are discussed, and limitations are considered. Then, concluding remarks are given, and recommendations for future work are made.

2

Background

Smartphones are one of the most impactful technologies of our time. They have revolutionized communication and information access and have had a major social and cultural impact, too. Users report preferring their smartphones over sex [67] and spending an average of about a third of their waking time using their smartphones [117].

2.1. Battery Developments

Amidst the rapid development and adoption of smartphones, battery life has remained a pressing concern. While many parts of smartphones have matured (processors, screens, form factors) and the rate of change has slowed, most users still report desiring longer battery life. Battery life is one of the most important factors determining smartphone customer satisfaction. However, raw battery capacity is not increasing at a fast rate, with densities improving at about 4-5% per year over the last decade [128]. The 2014 Samsung Galaxy S5 contains a 2800 mAh battery [41], while the 2023 Samsung Galaxy S23 contains a 3900 mAh battery [82]. Although the newer device has a 39% higher battery capacity, its internal volume has also increased by 30% [41, 82].

Observed battery life is sometimes even reported to decrease year-over-year due to more energy-intensive components [109]. Users place increasing demands on their smartphones, desiring ever-improving camera systems with sophisticated (and energy-intensive) processing, high-speed networking over 5G, and running apps that perform on-device machine learning, augmented reality, and high-resolution graphics.

Users benefit from all these improvements, and many would not dream of going back to their then-flagship device from five years ago. However, there are scenarios when users would prefer longer battery life over maximum performance and functionality. Users may be traveling without access to a power source, performing particularly power-intensive tasks like GPS navigation, or have older phones with chemically degraded batteries. We cannot increase physical battery capacity temporarily, and modern phones usually do not have user-replaceable batteries, so unless the user brings and remembers to charge an external battery pack, we need to look toward software.

2.2. Battery Optimization

Luckily, many effective software optimizations for increasing battery life are possible. Some of the most power-intensive components in a modern smartphone are the screen, location access, especially using GPS, and the processors. To reduce screen

power consumption, we can lower the brightness, display darker colors on OLED screens, and reduce the screen refresh rate [10]. To reduce GPS power consumption, we can simply use it less, depending on other methods of determining location, such as Wi-Fi network Wi-Fi triangulation if possible. Processors can be kept asleep for as long as possible by batching operations and postponing non-time-sensitive operations, reducing power consumption.

Many of these optimizations, however, have trade-offs, such as reduced screen visibility, less accurate location determination, and delayed processing. Therefore, manufacturers tend to package these optimizations into modes that the user can enable and disable at will. This allows the user to trade off battery life versus functionality. Smartphone manufacturers HTC (in 2011, on its *Desire* smartphone [46]) and Samsung (in 2012, on its *Galaxy SII* smartphone [40]) introduced battery-saving modes in the early 2010s, and many other manufacturers followed suit. In 2014, Google shipped a built-in *Battery Saver* mode with Android 5.0 [83] and further expanded it with subsequent releases.

2.3. History of Android Power-Saving Features

Let us take a look at the high-level power-saving features introduced in Android over the years.

- In 2014, as stated before, Google introduced Android 5.0 Lollipop, featuring Battery Saver [5], a battery-saving mode users could enable manually or set to enable when battery charge dropped below a predefined level. It restricts background activity and networking [108], background location access [108], disables user interface animations [61], vibration [78], and 5G networking [108], and slows down the processor [37], among others.
- 2015 brought more power-saving features, including Doze mode [27], which only periodically wakes up background apps when on battery power, and App Standby [27], which reduces background activity for unused apps.
- Android 7.0, launched in 2016, introduced an extension of Doze [17], informally often called "Light Doze", which applies a subset of the Doze restrictions after turning the screen off. Compared to 'full' Doze, its measures are less restrictive, and they are activated earlier: for example, the system does not wait until the device is stationary.
- In 2017, restrictions were placed on broadcasts [21], a feature that allows apps to send messages to other apps.
- In 2019, these broadcast restrictions were strengthened, and Google launched App Standby Buckets [19], a refinement of App Standby that further limits apps based on how frequently and recently they have been used. Based on this data, apps are placed into five buckets, and the system limits device resources based on the app's bucket. 2019 also brought improvements to Battery Saver [18], such as putting background apps to standby more aggressively, disabling location when the screen is off, and applying some restrictions to all apps regardless of target API level.
- 2021 brought features to further delay alarms [29], which allow an app to schedule itself to be woken up by the system.
- In 2023, new features were launched to freeze cached applications and optimize broadcasts for cached applications [7].

2.4. Potential for Bugs

As becomes clear from the above, Android contains an impressive array of features that reduce power consumption. Apps are automatically regulated so as not to drain the user's battery. This benefits developers, who have to worry less about

their app's power consumption, and users, who can count on their devices lasting a long time. However, it also comes with an increased need for testing and validation. Apart from all the existing hardware fragmentation (different screen sizes, camera, and sensor setups), developers now also have to concern themselves with a new software mode (Battery Saver) and many types of restrictions on background processing. It is not surprising that developers struggle with these modes, and there are hundreds of questions on Stack Overflow and thousands of public issues on GitHub on this topic.

2.5. Testing Approaches

For software testing in general, it is helpful to distinguish two main approaches: *static* and *dynamic* testing. Their core difference is their mode of analysis: static testing does not execute the code of the program under test, whereas dynamic testing does. Static testing has the advantage of not being limited by the running speed of the software under test; it can, therefore, often be faster than dynamic testing. Dynamic testing, on the other hand, has the advantage of being able to see software behavior in action: it is, therefore, less likely to lead to false positives and can detect sophisticated behavior bugs that may not be detectable from source code alone.

In the next subsections, we provide more detail on the trade-offs between static and dynamic testing approaches, based on four key dimensions: accuracy, realism, performance, and coverage.

2.5.1. Accuracy

The accuracy of a bug detection tool is paramount, as it directly impacts the reliability of the test outcomes. Static analysis often falls short in terms of accuracy due to its inherent limitations in handling dynamic behavior and runtime conditions. It may generate a higher rate of false positives and false negatives because it abstracts away some runtime specifics.

On the other hand, dynamic analysis offers more accurate results by executing the code and observing its behavior. This method is particularly effective in complex scenarios where bugs manifest only under specific runtime conditions, thus reducing the likelihood of false positives. However, its accuracy heavily depends on the comprehensiveness of the test cases used.

2.5.2. Realism

Realism in testing refers to how closely the test environment and scenarios mimic actual operating conditions under which the software will run. Dynamic testing excels in this domain as it evaluates the software in a state that closely resembles its execution in a live environment, thereby providing a realistic assessment of its behavior and performance.

Static testing, while useful for preliminary checks, often cannot replicate the complex user interactions and other environmental variables that can influence the behavior of the software, leading to a gap between test results and real-world performance.

2.5.3. Performance

Performance testing is crucial to ensure that the software not only functions correctly but also meets the required speed and efficiency standards under various conditions. Static analysis is generally faster and less resource-intensive as it does not require the program to be executed. This makes it suitable for regularly run checks and fault detection early in the development cycle.

Conversely, dynamic testing, which involves code execution, tends to be slower and

more resource-demanding. However, it is indispensable for accurately measuring the software's performance metrics, such as response time and resource utilization, under realistic conditions.

2.5.4. Coverage

Coverage measures the extent to which a testing method can examine the code base. Static analysis typically offers higher coverage in less time because it analyzes all code paths without executing them. This approach ensures that even the parts of the code that are rarely executed in a typical runtime scenario are examined.

Dynamic testing, though more constrained in coverage due to its reliance on executing specific scenarios, is critical for uncovering issues that only manifest during runtime. Its coverage is limited by the defined scenarios and test cases or exploration strategy, which might not encompass all possible execution paths.

To summarize, we collect the analysis above into Table 2.1. It is clear from the table that static and dynamic analysis have different strengths and weaknesses.

	Static Analysis	Dynamic Analysis
Accuracy	+	++
Realism	-	+
Performance	+	-
Coverage	++	-

Table 2.1: Comparison of Static and Dynamic Analysis

2.6. Automated Test Generation

Testing software is standard practice to ensure quality and limit the number of bugs and their severity [71]. However, it is tedious to exercise every function of the target application manually. To address this, automatic tools for dynamically testing Android applications have been developed. Random tools, such as the *UI/Application Exerciser Monkey* [30] developed by Google, often shortened to *Monkey*, stress test applications by generating random events such as touches, system navigation actions such as *Back*, and scroll actions. However, due to its random nature, it struggles with common patterns such as deep navigation hierarchies and login screens since it, for example, cannot systematically navigate to new screens or enter a password. Therefore, improved approaches have been proposed, which usually employ a model that stores which action leads to a screen to limit repetition. These include, in increasing order of activity coverage, *Stoat* [105], *Ape* [42], and *Fastbot2* [62].

2.7. Energy Testing

Energy testing can be divided into two approaches : *direct* and *indirect measurement*. In direct measurement, energy consumption is measured at the physical level by measuring currents and voltages. The hardware for this is included in most phones; when this is used, this is called internal direct measurement. We can, however, also attach the device to an external power meter, leading to the name *external direct measurement*. While quite accurate on the device level, these approaches cannot naturally differentiate the different sources of power draw.

In *indirect measurement*, energy consumption is modeled by taking software characteristics and relating them to energy consumption. By definition, this approach is less accurate for device-level measurement because the model estimates rather than measures power consumption. However, this approach is more flexible because factors like idle power consumption and running multiple applications concurrently can be accounted for. It can be divided into three categories. *Working-time models*

aggregate the time various smartphone systems (CPU, Wi-Fi, Bluetooth, Location) are in use. *Instruction energy models* look at the instructions themselves. API call energy models look up the APIs used in an application in a table of known APIs and their energy usage.

3

Related Work

In this chapter, we review the prior work related to defects caused by power-saving features in Android applications. While no previous work addressing our exact topic exists, several works partially overlap. We show a comparison of the relevant points of comparison in Table 3.1. We provide more details on the points of comparison in the next sections.

Table 3.1: Comparison of Closely Related Work

	Our tool	CiD	FlowDroid	SetDroid	SetChecker	PATDroid	PREFEST
Detects bugs directly	✓	✓		✓	✓	✓	✓
Target Battery Saver bugs	✓			✓	✓		
Battery Saver bugs detected	41	N/A	N/A	0	0	N/A	N/A
Metamorphic testing only				✓			
Targets non-crash bugs	✓		✓	✓	✓	✓	✓
Static/Dynamic	Static	Static	Static	Dynamic	Static	Hybrid	Hybrid
Static sensitivities	Flow Path Context	Flow Path	Flow Context Field Object	N/A	Unreported	N/A	N/A

3.1. System Settings Bugs

Battery Saver is a system setting on Android, that is, it is setting controlled from outside the application. Therefore, other work on problems surrounding such settings is relevant. Sun et al. [107] develop *SetDroid* for finding setting-related defects: defects that appear depending on a system setting, such as user interface (UI) language. The authors start from the assumption that when changing a setting, except for the behavior change desired by changing the setting, the application should remain the same. For example, after changing the language setting, all buttons originally present should remain present, as only their text content may have changed. This approach is named *metamorphic testing*. The authors use modified random test seeds. In an extending study [106], the authors propose a static analyzer named *SetChecker* that allows a broader range of defects to be identified. Compared to our work, these studies do not identify any bugs caused by Battery Saver mode. Neither does *FlowDroid*, presented in [2], but its approach is interesting because it includes many sensitivities.

Permission defects are empirically investigated in [119], and tools to detect them are proposed in [118, 81, 60]. Of these, *PATDroid* [81] and *PREFEST* [60] detect a

limited subset of settings and only support detecting crashing bugs. *Aper*, proposed in [118], statically detects permission defects caused by updated Android APIs.

3.2. Data Loss Bugs

An important type of bug relevant to this work is the data loss bug. Appropriately named, it refers to bugs in which user data is lost inadvertently. These bugs are relevant to this work because, somewhat surprisingly, toggling Battery Saver mode can also cause data loss issues.

Early, dynamic work on this problem [126, 1] requires testing suites. *DLDetector*, a dynamic tool proposed in [80], improves upon these approaches by including an exploration strategy to avoid needing a testing suite. Early static work such as *KREFinder* [85] does not require a hand-built model that the early dynamic approaches do. Later approaches are often static too: they include *LiveDroid*, proposed in [36], compared to [85] the authors add consideration of UI elements in resource files. Furthermore, while *KREFinder* [85] erroneously assumes all variables are instance state, *LiveDroid* [36] reasons about what should be saved, preventing over-saving, which can lead to low performance. *iFixDataLoss*, put forward in [43], combines static and dynamic analysis by first building a transition graph statically and then exercising the app dynamically using guided exploration. It generates the methods for implementing saving instance state. As mentioned in chapter 2, there are multiple techniques for saving state. However, *iFixDataLoss* only supports one, namely saving instance state. Lastly, *DDLDroid* [127] outperforms earlier work with a fully static tool that has the advantage of substantially shorter running time than competing approaches. It uses data flow analysis and excludes immutable properties to avoid over-saving.

3.3. Detecting Problematic APIs

Several studies [58, 59, 57, 120] have focused on automatically identifying incompatible API calls in Android applications. As APIs malfunctioning in Battery Saver mode may be seen as an "incompatible" call too, this work is relevant for our problem. The tool *FicFinder*, proposed in [120], uses static analysis to find compatibility issues automatically, although this process still requires manual effort. Liu et al. [58], and its extension [59] analyze Android apps to find incompatible API calls, i.e., APIs that exist in one Android version but are later removed in a newer version. The related topic of silently-evolved methods, that is, APIs that are changed without the corresponding documentation being updated, is addressed by Chen et al. [9].

3.4. Android Battery Saver

To our best knowledge, only a single study has been performed addressing the non-functional aspects of Battery Saver mode on Android devices. In [39], the performance of the default Android web browser in this mode is measured. As Battery Saver mode may limit the clock frequency of the central processing unit (CPU), the authors hypothesize that web browsing loading times increase when this mode is active. They analyze a large-scale data set and then show that this slowdown occurs on some devices, in particular, on Sony and Huawei devices, but newer high-end models do not exhibit it. It differs from our study in that it considers performance instead of functionality.

3.5. Android Doze Mode

Recent work has also addressed Doze mode, another power-saving feature in modern versions of Android. In particular, Chen et al. [9] attack Doze mode by designing an application that intentionally does not allow Doze mode to function by

constantly scheduling alarms that wake up the device. Running the application drains the user's battery up to six times faster than the baseline.

3.6. Dark Mode

A system-level so-called 'Dark Mode' or 'Dark Theme' is a recent addition to mobile operating systems. It darkens the UI colors to increase comfort in low-light settings. Interestingly, a generally darker screen also reduces power consumption on OLED smartphone screens. For this reason, Android's Battery Saver mode enables dark mode, which makes this work relevant to this study. Several works address the power consumption of mobile devices with dark mode enabled. Dash and Hu [13] build a model for the power consumption of OLED screens. Other studies [115, 50] compare web browsing energy consumption in both light and dark themes. These works treat modeling and reducing power consumption, whereas our work focuses on bugs caused by Battery Saver mode.

4

Characterization

In this chapter, the methodology for characterizing and analyzing Battery Saver issues is presented. The methodology aims to provide a path to meeting the research objectives in section 1.2. The resulting characterization seeks to provide developers and researchers with a clear picture of the failure modes of Android apps under Battery Saver mode.

4.1. Characterization Methodology

The section presents an outline of the stages of the research process and their interconnections. It presents four steps, followed in order. In the following subsections, each step will be discussed in detail to aid reproduction. We then report the results of applying this methodology, consisting of 13 discrete bugs. We describe and analyze them in detail based on their manifestation, the API implicated, severity, and potential fixes or workarounds.

4.1.1. Determine Keywords

To systematically identify relevant keywords for finding Battery Saver issues on Android, we started by identifying the common terms for this feature. To find unbiased and representative keywords, we used a systematic process starting from the most basic set of known relevant terms: the marketing term for this feature, 'Battery Saver'[1] and the relevant API to check whether this feature is currently enabled, `PowerManager.isPowerSaveMode()`. We then transform these terms systematically into their constituent parts to get our query set:

1. battery saver
2. power manager
3. PowerManager
4. isPowerSaveMode
5. is power save mode

We remove 'is' from the fifth query since it does not add to the term semantics.

We then add the quoted versions of these queries to the query set. For single-word queries, we do not add quoted versions since these return the same result as the original query. Therefore, we add:

1. "battery saver" 2. "power manager" 3. "power save mode"

Resulting in the total keyword set:

[1] See https://support.google.com/pixelphone/answer/6187458?hl=en

1. battery saver
2. power manager
3. PowerManager
4. isPowerSaveMode
5. is power save mode
6. “battery saver”
7. “power manager”
8. “power save mode”

These queries are sufficient for querying Android documentation. However, to disambiguate them for the other resources, that is, GitHub and Stack Overflow, we modify these queries by prepending “android”, resulting in the following keyword set for GitHub and Stack Overflow:

1. [android] battery saver
2. [android] power manager
3. [android] PowerManager
4. [android] isPowerSaveMode
5. [android] is power save mode
6. [android] “battery saver”
7. [android] “power manager”
8. [android] “power save mode”

4.1.2. Query Documentation and Issues

To find relevant problematic behaviors and APIs, we query the Android documentation, the public issue database of GitHub, and the Stack Overflow programming help website. We use the keywords previously identified, that is:

1. [android] battery saver
2. [android] power manager
3. [android] PowerManager
4. [android] isPowerSaveMode
5. [android] is power save mode
6. [android] “battery saver”
7. [android] “power manager”
8. [android] ”power save mode”

We perform one query per keyword per resource (documentation, GitHub, Stack Overflow). We check the resulting document for inclusion in the order returned by the resource, which for all three is set to be based on ‘relevance’. To make the process more efficient, if 20 subsequent issues are deemed irrelevant, we terminate the process.

We retrieve up to 200 documents per query per resource[2]. If we reach the stopping criterion of 20 consecutive irrelevant documents, we stop the search. Otherwise, we retrieve 200 more documents, repeating this process until we either reach the stopping criterion or exhaust the set of documents.

[2]The search may return fewer than 200 results, in that case we simply take all results

The official Android documentation is not accessible via an API, therefore, we query it manually. GitHub and Stack Overflow have an API that allows us to script the data retrieval.

After this step, we are left with 1894 artifacts.

4.1.3. Data Preprocessing and Cleaning

Before we can identify the APIs and system features used in them, we must clean the data since it comes from an unfiltered data set. We removed duplicates and then used the following inclusion criteria:

Inclusion criteria
Artifacts must be:

1. Addressing a software problem.
2. Relevant to the Android operating system.
3. Relevant to native Android development.
    - Native, cross-platform frameworks like React Native[3] or .NET MAUI[4] are also to be included.
4. Directly related to the built-in Android feature Battery Saver
5. Clear about the failing component, for example, by mentioning the problematic API explicitly or providing a code sample.
6. Available. Deleted posts or issues may be returned by the search but will be discarded since they provide no useful information.

Exclusion criteria
Artifacts must not be:

1. Reported only to appear on a single manufacturer's device other than Google
    - Artifacts describing issues caused by OEM Battery Saver additions or changes are not to be included
2. Documented in a language other than English. The Android documentation has been translated into various other languages and including them would lead to duplicated results.

These criteria ensure that only artifacts relevant to our research questions are included and that they are clear enough to be useful for answering them.

4.1.4. Collect and Categorize APIs

We review all artifacts to determine the problem described. For each problem, we categorize it based on the following factors:

1. Related API(s)
2. Severity, based on Atlassian's incident management classification [3]
   In decreasing order:
    (a) Critical, having a high level of user experience impact, such as crashes with data loss or core functionality impacted.
    (b) Major, having a moderate level of user experience impact, such as crashes without data loss or secondary functionality affected.
    (c) Minor, low impact. Minor inconvenience or affecting niche feature. Workaround may be available.

[3]See `https://reactnative.dev`
[4]See `https://dotnet.microsoft.com/en-us/apps/maui`

3. Frequency of mention - Issues mentioned more often may occur more often in general. It also lends credibility to the report: while a single bug report may be a fluke, especially on an open platform like GitHub or Stack Overflow, multiple reports corroborating the same problem increase our confidence that the issue exists.
4. Possibility of building oracle - Is there a feasible automatic test to see if this issue occurs? If not, automatic tools for fixing this issue are hard to build.

We then group and collate all artifacts into a master repository. We group them by how they manifest. For example, 'background jobs do not run' could be one entry. There may be multiple reasons for this bug occurring, but since they manifest in similar ways, we can group them here.

Our collation process leaves us with a set of distinct problems, including any info collected from the underlying GitHub issues, Stack Overflow questions, and relevant documentation pages. We present these problems in detail in section 4.2.

4.2. Characterization Results

In this chapter, we report the results of applying the methodology in chapter 4. We share the outcomes of the previously outlined characterization process and analyze them in detail.

The GitHub search returns 1456 results. After deduplication, 807 issues are left. Following the process set out in chapter 4, we include 21 issues. The most common reasons for exclusion are "not concerning Battery Saver" (171 instances), "not reporting a problem" (29 instances), and "not related to clean Android' (17 instances). The Stack Overflow search returned 1448 results. After deduplication, 998 issues remain. Of these, we include 30. The most common reasons for exclusion are "not concerning Battery Saver" (111 instances), "not related to Android" (27 instances), and "not related to clean Android" (19 instances). The Android documentation only returns a maximum of 50 results per query. Applying the exclusion criteria, 163 issues remain, of which nine meet all inclusion criteria.

The characterization process provides us with 57 artifacts describing issues caused by Battery Saver. In this section, we look at these artifacts in more detail and consider the manifestation, severity, frequency of mention, documentation status, and suggested fix of these issues. We then group the artifacts that mentioned the same or similar issues, resulting in 13 separate issues. We summarize the collated data in Table 4.1. It shows all identified issues, the issue manifestation, APIs implicated, severity, support, potential fixes or workaround, and further notes if applicable. We have also assigned a unique identifier to each issue for easy identification, shown in the leftmost column.

4.2.1. Detailed Characterization of Identified Issues

The following sections provide an in-depth look at the specific issues identified during the study on Battery Saver bugs in Android applications. Each issue is analyzed in terms of its manifestation, the APIs it affects, its severity, and possible fixes or workarounds.

BS1: Location Services Disabled on Screen Off

An important Battery Saver modification is that location services are disabled when the screen is turned off under Battery Saver mode. This bug affects important mapping and navigation apps relying on continuous location updates. The primary API implicated in this issue is `android.location.LocationManager`. This issue is classified as major due to its potential to severely disrupt location-dependent services. No straightforward workaround is available since this behavior is system-level, suggesting a need for systemic changes in app handling under Battery Saver

Table 4.1: Summary of Identified Issues

Issue ID	Issue Manifestation	API(s) implicated	Severity	Support	Fix/Workaround	Details/Notes
BS1	Location services disabled on screen off	`android.location.LocationManager`	Major, core feature of mapping / navigation apps affected	5 reports + Google-documented	None proposed, whitelisting probably doesn not even work since this is system-level, not app-level	See https://developer.android.com/develop/sensors-and-location/location/background#checklist
BS2	Background networking disabled	Any networking API, seemingly. Also will report status `disconnected`	Major, core feature of e.g. communication apps affected	12 reports + Google-documented	Use a foreground service	-

BS3	Network looks disabled even if it is not	`android.net.NetworkInfo.isConnected()`
		
		`android.net.NetworkInfo.isConnectedOrConnecting()`
			Minor, not a core feature	3 reports + Google-documented	Suggested to try pinging a server	Deprecated solution: get detailed info or use isConnectedOrConnecting(). Google points to new API, unclear how it behaves.

BS4	App cannot run infinitely in background	`android.app.IntentService`	Major, core feature of many apps	4 reports	This is deprecated in Android 11 and unsupported on modern Android. Use ForegroundService or JobScheduler / WorkManager APIs	Deprecated in Android 11(level 30) and affected since 8 (level 26). While documentation does not mention it, Battery Saver is reportedly involved too
BS5	JobScheduler jobs are not run	`android.app.job.JobScheduler`	Major, can be an important feature, background services often migrated to this.	3 reports + Google-documented	-	Status reported with: `android.app.job.JobParameters.PENDING_JOB_REASON_DEVICE_STATE` and `android.app.job.JobParameters.PENDING_JOB_REASON_DEVICE_STATE`

BS6	Alarms are not fired	`android.app.AlarmManager.setInexactRepeating()`
		
		`android.app.AlarmManager.setAndAllowWhileIdle()`
		
		`android.app.AlarmManager.setExact()`
		
		`android.app.AlarmManager.setWindow()`
			Major, sometimes a core feature, especially if `setAlarmClock` breaks	3 reports + Google-documented	-	Only `setExactAndAllowWhileIdle` is reliable, but can only fire once per 15 mins and may be ordered arbitrarily. `setAlarmClock` can also be used but is only designed for highly visible things like alarm clocks, and one report of it not firing either

BS7	Activity restarts in dark mode	`android.app.Activity`
		
		`androidx.appcompat:appcompat.MODE_NIGHT_AUTO_BATTERY`
			Critical, can cause data loss and crashes	7 reports + Google-documented	Implement data/state saving or disable activity redrawing, which may lead to other issues and is not recommended by Google	Dark theme gets enabled, causing activity to redraw and potentially causing data loss

BS8	> 60Hz frame rates clipped to 60Hz	`Surface.setFrameRate()`
		
		`SurfaceControl.Transaction.setFrameRate()`
		
		`ANativeWindow_setFrameRate()`
		
		`ASurfaceTransaction_setFrameRate()`
			Minor, just a nuisance usually	Google-documented	No workaround mentioned	-

BS9	Work requests may not execute	`androidx.work:work-runtime.WorkRequest`
		
		`androidx.work:work-runtime.WorkRequest.setExpedited()`
		
		`androidx.work:work-runtime.Constraints.Builder.setRequiresBatteryNotLow()`
			Minor, usually not a core feature, otherwise would be e.g. a foreground service	Google-documented	Use alternatives such as a foreground service	Throws exception when foregrounding is not allowed, see https://developer.android.com/develop/background-work/background-tasks/persistent/getting-started/define-work#backwards-compat

BS10	Animations do not run	`android.Animator` and subclasses (e.g. `ValueAnimator, ObjectAnimator`)
		
		`android.widget.ProgressBar`
		
		`com.google.android.material.progressindicator.CircularProgressIndicator`
			Critical, can cause core functionality to break down confusingly if elements never appear on screen	10 reports	Use `animation.*` APIs. Check that non-animating behavior is clear (especially: reaches end state). Check that animators are enabled using ValueAnimator.AreAnimatorsEnabled() or put in `Runnable` to schedule after view lays out (see report 9)	android.view.animation subclasses explicitly **not** affected according to Xamarin Forms issue #8382, also mentioned by 2048-Battles issue #261

BS11	Cannot start foreground service from background	`android.app.Service`	Major, no big user impact reported	2 reports + Google-documented	Disable background optimizations or a slew of other options, see https://developer.android.com/develop/background-work/services/foreground-services	-
BS12	Foreground service gets killed by Battery Saver	`android.app.Service`	Major, affects core features such as data upload	3 reports, bug verified and fixed by Google	Workaround by starting separate process for service, mentioned to be fixed in Android 8+	Confirmed bug in Android. Issue tracker only mentions Doze but reported twice on Battery Saver too. Hardly documented. Only Android 6/7
BS13	GPS does not respond	`LocationManager`	Major, may break core piece of application	1 report	Disable BS, potentially whitelist?	Not well corroborated

mode and/or user education. Popular apps like Strava [111] and Ride with GPS [6] have begun warning their users to disable Battery Saver on Android phones.

Supported by [18, 64, 84, 113, 72]

BS2: Background Networking Disabled
Another significant issue involves the disabling of background networking, which impacts any application relying on network connectivity to perform background operations. This bug primarily affects communication apps and is linked to several networking APIs. The severity of this bug is also classified as major. A possible fix is the use of a foreground service to ensure connectivity remains active.

Supported by [18, 63, 12, 68, 76, 65, 14, 47, 79, 75, 74, 73]

BS3: False Network Disconnection Reports
Applications might receive incorrect signals that the network is disconnected when it is not. This minor issue involves APIs such as `android.net.NetworkInfo .isConnected()` and can lead to poor user experiences due to unnecessary handling of "no network" errors. Developers can mitigate this by implementing additional checks on the Battery Saver status or pinging a server to verify connectivity.

Supported by [66, 91, 87]

BS4: App Suspended in Background
Under Battery Saver mode, apps are often prevented from running indefinitely in the background, which affects services designed for continuous operation. The implicated API `android.app.IntentService` has been deprecated, pushing developers towards using `JobScheduler` or `ForegroundService` for background tasks. This issue is a major concern as it requires significant changes to app architecture for compliance with modern Android standards.

Supported by [88, 101, 94, 103]

BS5 JobScheduler Jobs Are Not Run
A major issue can arise when jobs that are scheduled via the `android.app.job. JobScheduler` API do not execute under Battery Saver mode, significantly affecting apps that rely on background tasks for functionality such as syncing data or processing tasks. This problem can severely impact the reliability of applications that use `JobScheduler` for important operations. The severity of this issue is classified as major due to its potential to disrupt critical app functionalities such as file uploads.

Developers should also consider handling the `android.app.job.JobParameters. PENDING_JOB_REASON_DEVICE_STATE` callback to determine if a job is pending due to device state restrictions and implement conditional logic to reschedule jobs or handle failures appropriately. This approach requires a robust error-handling and job-management strategy to ensure that app functionality remains consistent even under restrictive conditions imposed by Battery Saver mode.

Supported by [28, 26, 4]

BS6: Alarms Do Not Fire
Scheduled alarms using `AlarmManager` may fail to fire under Battery Saver mode, which disrupts both apps that rely on precise timing (e.g., reminder and alarm apps) and apps that use alarms to perform scheduled work. The severity of this issue is major, especially for applications where timing is crucial. The workaround involves using the `setExactAndAllowWhileIdle` alarm instantiation method, although this method imposes limitations on the frequency of alarm execution.

Supported by [15, 38, 102]

BS7: Activity Restarts in Dark Mode
Switching to Battery Saver mode can trigger an unwanted activity restart when the system transitions to a dark UI mode. This is critical because it can lead to data loss or crashes if the activity state is not properly managed. The APIs involved include `android.app.Activity` and `androidx.appcompat:appcompat.MODE_NIGHT_AUTO_-BATTERY`. A possible fix involves implementing robust state saving and restoration techniques or explicitly handling mode changes without redrawing the activity, which, although easy to implement, is not recommended due to potential side effects on app behavior.

Supported by [25, 16, 20, 86, 89, 95, 92]

BS8: Frame Rate Limitations
Under Battery Saver mode, high frame rates are often clipped to 60Hz to save power on devices that support higher refresh rates, affecting apps that rely on higher frame rates for smooth visual experiences, such as games or video applications. The APIs affected include `Surface.setFrameRate()` and its variants. This issue is considered minor as it generally only causes a visual nuisance rather than a functional impairment. There is no direct workaround other than adjusting app expectations concerning frame rates when Battery Saver mode is active.

Supported by [24]

BS9: Work Requests May Not Execute
Scheduled work requests using the `WorkManager` API may not execute as expected under Battery Saver mode, which can delay or prevent background tasks from running. This issue typically impacts non-critical background tasks but can escalate to major if critical operations are deferred. The affected API is `androidx.work:work-runtime.WorkRequest`. Developers can use alternatives such as a foreground service to ensure reliability.

Supported by [22]

BS10: Animations Are Not Run
Animations within apps do not run in Battery Saver mode, which can confuse users if UI elements rely on animations to appear or convey information. This affects various animator APIs and is classified as critical if it prevents core functionalities from being accessible. Developers are advised to ensure that UI elements reach their end state even without animations or to check the animation status dynamically and adjust the behavior accordingly.

Supported by [34, 123, 33, 32, 125, 112, 100, 104, 99, 93]

BS11: Cannot Start Foreground Service from Background
Starting a foreground service from the background can fail under Battery Saver mode, potentially affecting apps that need to promote a background service to the foreground to continue tasks without interruption. The implicated API is `android.app.Service`. This issue is major due to its impact on service continuity. Workarounds include requesting the user to disable battery optimizations or adjusting the app design to avoid needing to start services from the background.

Supported by [23, 110, 56]

BS12: Foreground Service Gets Killed
Even when configured correctly, foreground services may be terminated by the system under Battery Saver mode, affecting apps that rely on continuous operation to function properly, such as music players or location trackers. This bug, primarily associated with `android.app.Service`, has been addressed in recent Android

updates but may still occur in older versions. A possible fix is to implement a separate process for the service or to apply for exemptions from battery optimizations where feasible.

Supported by [90, 96, 97]

BS13: GPS Does Not Respond
A disruptive issue encountered under Battery Saver mode is the lowered accuracy and non-responsiveness of GPS services, which can critically impact apps that depend on real-time location tracking, such as navigation and fitness tracking apps. This issue is associated with the `LocationManager` API and is classified as major due to its potential to completely disable core functionalities of location-dependent applications.

This issue has only been reported once and has not been documented by Google, so it would benefit from more evidence.

The severity of this issue may require users to manually disable Battery Saver mode to restore GPS functionality, which is a poor solution as it compromises the device's battery life. Developers might consider implementing a fallback mechanism or alerting users about the reduced functionality when Battery Saver mode is detected. Additionally, requesting users to whitelist the app from battery optimizations can mitigate it.

Supported by [98]

4.3. Conclusion

In this chapter, we laid out the methodology that we follow to characterize the relevant issues caused by Battery Saver mode on Android devices. We systematically collect reported problems and rate them in various dimensions to identify the issues with the highest impact.

We then provide the results of executing this methodology, a systematic characterization of Battery Saver bugs across various Android APIs. Each identified issue presents unique challenges and necessitates specific developer interventions to mitigate the impact on app functionality and user experience. Continuing advancements in Android's power management features will require ongoing vigilance and adaptation from app developers to ensure smooth and reliable app operations, even under restricted power conditions.

As can be seen in the table, most problems have considerable user impact (rated major to critical). Diving deeper into the results, we see that 3/13 are classified as having 'Minor' severity, 7/13 as 'Major', and 2/13 as 'Critical'. All 'Major'-rated and up issues except one are reported at least three times independently. Four of these 'Major' and up issues, including one rated as 'Critical', are additionally not found in the documentation and are therefore easy to miss for developers.

Additionally, the problems are, on average, independently reported 4.1 times, showing these issues occur in practice and are noticed. Two of the issues have never been reported but have been documented by Google.

5

Bug Detection

In the previous chapter, we identified 13 problems caused by the Battery Saver functionality in Android. In this chapter, we turn our attention to solutions. We design and implement a tool that can automatically detect a subset of the problems we have identified. The goal is to warn developers during development when they use an affected API without taking proper precautions.

5.1. Problem Selection

From the characterization in section 4.2, we have the information required to prioritize the issues identified. We want to focus on issues where we can make an impact for developers and users alike. Issue BS11 looks promising: it has high impact, a potential fix, and is well corroborated. Sorted just by impact, issues BS4 and BS13 rank equally high. However, issue BS4 does not have a feasible detection strategy because it requires us to know whether apps are expected to run infinitely, while BS13 has weak corroboration and is only reported once.

Only issue BS11 *Animations Do Not Play*, has a viable detection strategy and is strongly corroborated. Additionally, it is not documented well by Google: it is not mentioned on the developer website for the feature, but only mentioned within the documentation for specific, non-obvious APIs, such as `ValueAnimator`.

5.2. Motivating Example

As a motivating example, we look at *Wikimedia Commons* [122], an Android application published by the Wikimedia Foundation. *Wikimedia Commons* is an open-source Android application that allows users to upload media to Wikimedia, which is not just the media repository of *Wikipedia* but also a separate project that "seeks to document the world with photos, videos and recordings" [122]. When run in Battery Saver mode, the application shows a confusing user interface, as shown in Figure 5.1.

Instead, when Battery Saver is enabled, the developer should provide an alternative experience, possibly showing but not animating the item.

5.3. Problem Specification

Most animations do not and should not play in Battery Saver Mode on Android devices. Animations may require complex graphics rendering, which taxes the Graphics Processing Unit (GPU), and require a high frame rate to be visually pleasing on modern devices, often 120Hz or even higher. Disabling animations is, therefore, one of the measures taken by the Battery Saver mode built into Android.

Figure 5.1: Example of an animation bug in the *Wikimedia Commons* application. Reported as issue #5710 [11].

In most cases, this poses no significant problem to developers or users. Animation durations are simply set to zero, so they complete instantly. Instead of animating smoothly to the final position or size, elements simply 'jump' to that state immediately. While this is less visually pleasing and maybe even slightly confusing because the user may see elements jump jarringly around their screen, the end state of these animations remains consistent with what the developer intended.

In some cases, however, this paradigm breaks down. In some Android versions, disabling animations stops some UI elements from appearing at all. This happens for some elements that do not have a clear end state: the elements may animate forever until they are removed from the screen. A common element, and the one we will focus on, is the progress bar shown in the motivating example. Such a UI element shows that a longer-running action is occurring and that the user needs to be patient. An example is provided in Figure 5.1. When run on Android 8.1 and earlier, progress bar elements only draw a static, circular arrow in Battery Saver mode. This confuses the user because it *seems* to represent a 'refresh view' or 'restart' button, both of which commonly use the same glyph. The progress bar, however, does not respond to taps, so users may repeatedly tap it without any effect. Indeed, in the *Wikimedia Commons* application shown in the example, tapping the 'reload' icon does nothing.

To make matters worse, progress bars with a custom graphical interface may not

be rendered at all, as shown in the `SpinKit` animation library [124]. Here, the developers provide a custom interface, which usually starts with an empty view and then smoothly animates an element into view and out again. With Battery Saver enabled, the Android animation engine now immediately draws the end state, which, for this repeating animation, is equal to the start state. This start/end state, however, is empty. Therefore, a device in Battery Saver mode will simply display empty space. This is confusing to the end user since the progress is not displayed anymore. The addition of the progress bar to the UI in the first place directly shows that the developer thinks the user may not realize they need to wait. The developer, however, may not be aware that this element never shows for users in Battery Saver mode. Even the user may not realize that disabling Battery Saver would fix the issue.

The key element in this bug is the *indeterminate* nature of these progress bars. Since they do not represent measurable progress, their animation is their sole source of utility. Removing this and not replacing it with a useful UI element in its place removes its meaning entirely. A solution to this problem should detect these cases of unclear or disappearing UI elements and report them to the developers.

5.4. Tool Approach

Based on the characterization presented in the previous chapter and the motivating example, we can now design a tool to automatically detect this bug at scale.

We first need to pick whether we will use the *static* or *dynamic* flavor of software testing. The concerns around this choice were already set out in chapter 2.

Given these concerns, we opt for static testing, picking large-scale application and coverage over maximum accuracy. This is for three main reasons: Firstly, this being the first study on these bugs, we want a sense of the scale of the problem, for which static testing is more appropriate. Secondly, the issues seem reasonably detectable with a static tool since we know the exact API that causes them. Lastly, the performance of static testing also more easily enables continuous testing as part of an automatic testing suite.

Looking at related work, we find the work of Li et al. [52] is quite similar in its purpose. The authors propose a method for detecting *API compatibility issues*: issues where developers invoke APIs that are unavailable on the current operating system version. Similarly to the bug we want to address, the bugs identified in Li et al. [52] should be addressed by adding a conditional statement, as shown in line 4 in Listing 5.1. This statement checks that the Android version of the device executing this code is new enough, in this case, equal to or newer than version Marshmallow, released in 2015. The API `ContextCompat.getColor(context, colorResId)` was only added in Android Marshmallow and invoking it on an older version causes a crash. On older versions, an older fallback method is invoked.

Listing 5.1: Example of an addressed API compatibility issue

```
1  public static int getColor(Context context, int colorResId) {
2      // Check if the Android version is Marshmallow
3      // (API level 23) or higher
4      if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
5          // Use the ContextCompat.getColor method introduced
6          // in API level 23
7          return ContextCompat.getColor(context, colorResId);
8      } else {
9          // Use the older method for API levels below 23
10         return context.getResources().getColor(colorResId);
11     }
12 }
```

The work by Li et al. [52] is not only similar but also quite influential, being cited

142 times at the time of writing. It builds upon the ideas of Li et al. [53], which won the *FOSS Impact Paper Award* at MSR 2018 [51].

Li et al. do not address any Battery Saver issues identified in this work. Moreover, their tool *CiD* is not suitable for finding issue BS11 *Animations Do Not Play* that we want to address due to the limitations we present in section 5.7. However, due to its relevant focus on conditional statements in building the call graph and the quality of the work, we decide to use their `CiD` tool as a foundation and modify and expand it to find Battery Saver animation issues.

5.5. Challenges

A main challenge bug detection is accuracy. Related to Android in particular, challenges include Android apps containing multiple entry points, complicated lifecycles that are component-specific, dynamic user events, inter-component communication, and third-party libraries [54]. For problem BS11 in particular, a challenge lies in separating problematic uses of `ProgressBar` from innocuous cases.

We can partly quantify these accuracy challenges and safeguard against them with manual effort. To avoid false positives, we can manually inspect the bugs identified by the tool and verify their veracity. To avoid false negatives, we test our tool on known faulty applications.

5.6. Technical Analysis

Having established that the source of the bug is the disablement of animations and its effects on the progress bar, we can now analyze the details of this bug to drive us toward a potential solution.

As shown in the previous section, we are interested in detecting indeterminate progress bars. Our first requirement, therefore, is detecting these elements. Since these appear problematically in Battery Saver mode, we need to know whether the app is checking if this mode is enabled. If the application only shows a `ProgressBar` when the device is not in Battery Saver mode, the app will not exhibit this bug.

There exists an API for checking whether the device is currently in Battery Saver Mode: `PowerManager.isPowerSaveMode()`. While this allows us to express the idea in the previous paragraph, we should be even more precise: while on Google Pixel devices, this mode disables animations, devices from different third-party manufacturers may or may not do so. Luckily, all manufacturers should report on the animation status engine specifically using the `ValueAnimator.areAnimatorsEnabled()` API. Furthermore, on many devices it is possible to disable animations separately using 'Developer mode' settings. Applications should therefore use the specific `ValueAnimator.areAnimatorsEnabled()` API instead of the high-level `isPowerSaveMode()` check.

A naive implementation could, therefore, reason as follows:

> If an indeterminate `ProgressBar` is initialized and its parent statement is not a condition on `areAnimatorsEnabled()`, report potential bug

However, there is a problem with this approach. There are multiple ways for developers to 'protect' their progress bars in Battery Saver Mode. Instead of a direct call to `areAnimatorsEnabled()`, they may also call a method that returns the value of `areAnimatorsEnabled()`. Obviously, a wide variety of these scenarios is possible: the app may, for example, implement a subtype of `ProgressBar`, or call a closure that calls `areAnimatorsEnabled()`, et cetera.

This shows that we need two properties: *path-sensitivity*, i.e., we need to know which conditions were true when the progress bar is initialized or modified, and *interprocedurality*, i.e., we must be able to transcend method boundaries with our

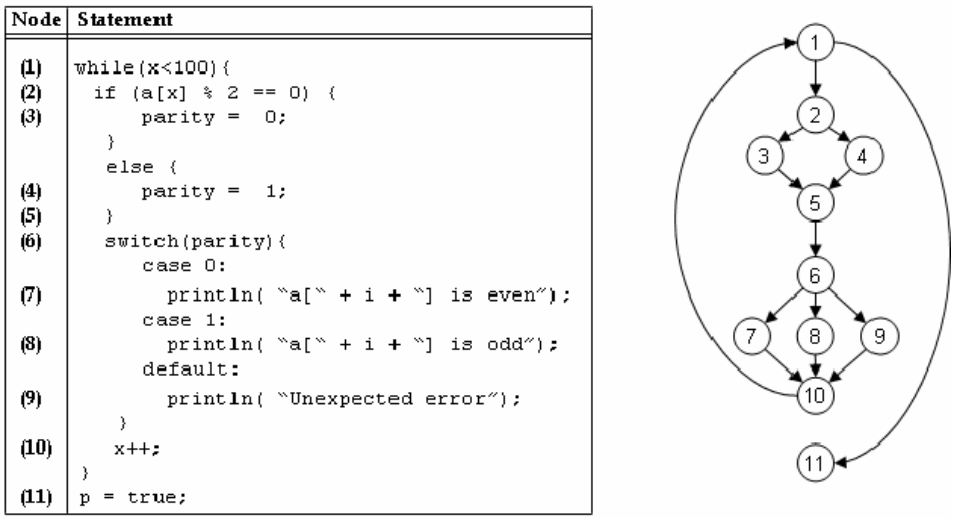

Figure 5.2: Example of a Java program and its corresponding control flow graph [8]

analysis to handle cases where the check is called in a different method and its value returned.

The tool needs to find calls to create an indeterminate `ProgressBar` that are not, somewhere up the control flow chain, protected by an appropriate check. This expression naturally leads to the idea of a control flow graph, in which we represent statements and control flow structures such as methods, conditional statements, and exception handling in a directed graph. An example of a Java program and its control flow graph are shown in Figure 5.2. Every statement is represented as a node in the graph, the edges show the possible control flow paths from one statement to the next.

More specifically, we can view the application as a graph with multiple roots, where each root represents an *entry point*. Instead of the familiar Java `main(String[] args)` method, Android applications can be launched in multiple ways. These *entry points* are defined by the Android framework and called by the operating system at the appropriate time.

For any potentially problematic statement, we can use the graph to find its predecessor statements by following the arrows in reverse. These statements can then be analyzed to determine if they contain checks protecting against the bug.

5.7. Required Modifications to *CiD*

In this section, we highlight two key static analysis problems the tool should handle, distinguishing invocations and distinguishing arguments.

5.7.1. Distinguishing Invocations

A problem in the CiD algorithm we discovered during testing is its context-insensitivity. That is, we cannot distinguish multiple invocations of the same method. This leads to unexpected and erroneous behavior, as can be seen in the (contrived) example in Listing 5.2.

Listing 5.2: Example of repeated invocation

```
1 // No bug (properly protected)
2 if (!ValueAnimator.areAnimatorsEnabled()) {
3     progressBar.setIndeterminate(true)
4 }
5
6 ...
```

```
// Bug (unprotected)
if (progressBar != null) {
    progressBar.setIndeterminate(true)
}
```

In Listing 5.2, the `setIndeterminate` method is invoked twice. The comments indicate the status of these invocations: the first is properly protected, whereas the second is not. While the second invocation is wrapped by an `if`-statement, the condition of the `if`-statement does not check the system-level animation setting. However, the CiD algorithm would erroneously mark the second unprotected invocation as protected.

Since CiD is context-insensitive, method invocations are not separately modeled. CiD is only sensitive to method *signatures*. Therefore, two invocations with the same `setIndeterminate` signature cannot be distinguished. Having multiple invocations of this method is a common occurrence, for example, in the real-world example from the well-known video player application *VLC* shown in Listing 5.3.[1][2]

Listing 5.3: Repeated invocations in the *VLC* application

```
if (scanProgress.inDiscovery && !scanProgressBar?.isIndeterminate) {
    scanProgressBar?.isVisible = false
    scanProgressBar?.isIndeterminate = true
    scanProgressBar?.isVisible = true
}

if (!scanProgress.inDiscovery && scanProgressBar?.isIndeterminate) {
    scanProgressBar?.isVisible = false
    scanProgressBar?.isIndeterminate = false
    scanProgressBar?.isVisible = true
}
```

Clearly, these two invocations are independent, and one can be protected while the other is unprotected. We should, therefore, model them separately.

5.7.2. Distinguishing Arguments

We are only interested in invocations of `setIndeterminate()` passing the argument `false`. If `true` is passed, we create a *determinate* ProgressBar, which does not exhibit the problem. *CiD* does not model or store arguments, therefore preventing distinguishing problematic and benign invocations of the `setIndeterminate()` method.

To address this problem, we need to include information about the arguments to method invocations in the graph to be able to filter out benign cases during further processing.

5.8. Architecture

In this subsection, we describe the architecture of the tool. We adapt the algorithm of *CiD* to our purpose of finding issue BS11. We update the list of Android support libraries that are excluded from analysis to reduce false positives and increase performance. To refine our static analysis and address the issues described above, we introduce modifications to *CiD* that allow for making a distinction between different method invocations and retaining information about invocation arguments.

[1] An attentive reader may notice that instead of a `setIndeterminate` method, an `isIndeterminate` method is invoked here. This is an example of a *synthetic property* in Kotlin. The two variants, `set-` and `is-`, are semantically identical. See `https://kotlinlang.org/docs/java-interop.html`

[2] Retrieved from `https://code.videolan.org/videolan/vlc-android/-/blob/2176328541cb020a04ada3a19b3e177a4ff3c080/application/vlc-android/src/org/videolan/vlc/gui/AudioPlayerContainerActivity.kt`

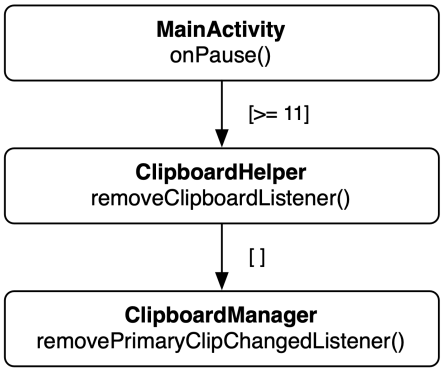

Figure 5.3: Example of a *Conditional Call Graph* (CCG) from Li et al. [52]

We do not describe the parts of *CiD* that are irrelevant to our purpose, such as API usage extraction and API lifetime modeling. These features are tied to the specific and separate problem of compatibility issue detection.

5.8.1. Main Components

The tool contains three main components: the `Conditional Call Graph`, `Transformer` and `Traverser`. In short, `Transformer` finds relevant classes in the application to analyze for problems. `Traverser` runs on every class found by `Transformer` to find invocations of the API we are interested in and returns their call stacks. `Traverser` also modifies the global CCG.

Conditional Call Graph

A Conditional Call Graph CCG is a call-graph with each edge representing a method invocation. Compared to a control-flow graph, it is less expressive, as it only shows method invocations instead of the full range of control-flow possibilities. However, the call-graph representation is enough for our tool.

An example of a CCG is given in Figure 5.3 The *conditional* nature of this graph means that edges are labeled by the conditions under which they can be taken. These conditions may also be the empty set, indicating that the edge can always be taken. This is a translation of the concept of a conditional statement in Java, such as an `if`-statement, to a call-graph representation of a program.

We modify *CiD*'s algorithm and conditional call graph to include richer edges, as explained in subsection 5.8.2.

In the following subsections, we explain the responsibilities of the main `Transformer` and `Traverser` components of the tool and explain the main loop of the tool execution.

Transformer

1. Find all the classes in the project, also in separate dynamically loaded libraries in `.dex` files. These libraries may be loaded at runtime, so we need to analyze them too.
2. Filter out library classes starting with prefixes `android.support`, `androidx`, `com.google.android`, and `kotlinx`. *CiD* only excludes the first of these, however, we found that this causes false positives and we need a broader set of excluded prefixes
3. Call `Traverser` on every method in every class.

Traverser

1. Construct a `Soot` graph for every method body.

2. Traverse this body consisting of `Soot Units`, which loosely represent Java statements.
   (a) Finding assignment statement: Ignore and continue loop, as variable resolution is not supported.
   (b) Finding `if`-statement: Check if it checks `areAnimatorsEnabled()`, recur on its body, passing animator status to ensure that conditions earlier in the control-flow graph affect the later nodes.
   (c) Finding method invocation: Add `Edge` with the conditions and call site to the CCG. If the method is a defined in an interface and not private, add edges to all classes implementing it. Otherwise, it cannot be extended, so we do not need to find its extending classes as there are none.
   (d) Finding return statement: Continue loop, this path has been fully explored.

Main Loop

1. Set up an empty, global CCG.
2. For each API of interest, obtain every invocation and the call stacks for them using `Transformer`.
3. Filter the returned call stacks on the developer prefix to ensure the invocation is not from a third-party library.
4. Check whether any call up the call stack checks whether animators are enabled.
   (a) If not, the invocation is not protected, report BUG.

5.8.2. Modifications to the Edge Representation

To address the problems identified in section 5.7, we make the following modifications to *CiD*'s edge:

1. The `Edge` class has been modified to include the call site and arguments along with the existing elements.
2. During program traversal, the call site and arguments are now captured, enhancing the granularity of our analysis.

To accommodate these changes, the structure of the `Edge` class used in the `CiD` algorithm was significantly altered, as shown in Listing 5.4 and Listing 5.5.

Listing 5.4: `CiD Edge`

```
class Edge(source: String, target: String, condition: Set<String>) {...}
```

Listing 5.5: Our enriched `Edge`

```
class Edge(source: Invocation, target: String) {...}

class Invocation(sig: String, callSite: CallSite, conditions: Set<String>,
    arguments: List<Value>) {...}

class CallSite(className: String, lineNr: Int) {...}
```

In the original `CiD` model, shown in Listing 5.4, edges consist of a source method signature, a target method signature, and a set of conditions. Our modified version, shown in Listing 5.5, introduces an `Invocation` class that includes the method signature, call site, and arguments, providing a more detailed and context-sensitive analysis. The `CallSite` class captures the class name and line number, facilitating precise and unique identification of the invocation context.

These changes allow the tool to capture significantly more information which is required for accurately finding issue BS11 we are interested in. In the next chapter, we evaluate the tool's performance and the usefulness of the bugs it identifies in open-source applications.

6

Tool Evaluation

This chapter provides a detailed evaluation of the diagnostic tool developed to identify and report energy inefficiencies in Android applications. The assessment focuses on the prevalence and nature of energy bugs within a curated set of open-source Android applications from the F-Droid repository that are also available on Google Play. The presence on Google Play suggests that these applications are more likely to be maintained actively since Google periodically prunes outdated applications, thus enhancing the relevance and impact of the evaluation. The availability on F-Droid ensures we can access the source code of the application, which is not required for executing the tool but aids bug reproduction and reporting.

6.1. Tool Evaluation Research Questions

The evaluation of the tool is structured around several research questions:

RQ1. **How prevalent are Battery Saver animation bugs in open-source Android apps?** We write a script to automatically flag potentially problematic API calls in more than 1400 publicly available apps and attempt reproduction for each call to get a ground-truth measure for the prevalence of these bugs.

RQ2. **How effective is the diagnostic tool in identifying these bugs?** We execute our diagnostic tool on the set of flagged applications, reproducing every positive. We then compare the results to the ground-truth labeled set to determine false positive and false negative rates.

RQ3. **How efficient is the diagnostic tool in identifying these bugs?** We evaluate the runtime of the tool to assess its practical feasibility.

RQ4. **Is the output of the diagnostic tool useful for verifying and addressing these bugs?** We conduct a live study by reporting the positives found by our tool as bug reports to the developers of the open-source applications under study.

6.2. Tool Evaluation Methodology

We divide the evaluation of the tool into five steps. We first select applications from F-Droid and Google Play. We then filter the applications on the API call of interest. After filtering, we determine the ground truth for each call by manually reproducing the bug. We set up the tool to run on every application with at least one reproduced bug. Using the ground truth we determined, we can find the false negative and false positive rate of the tool. Lastly, we report all reproduced issues to developers and track their responses to help developers address these issues

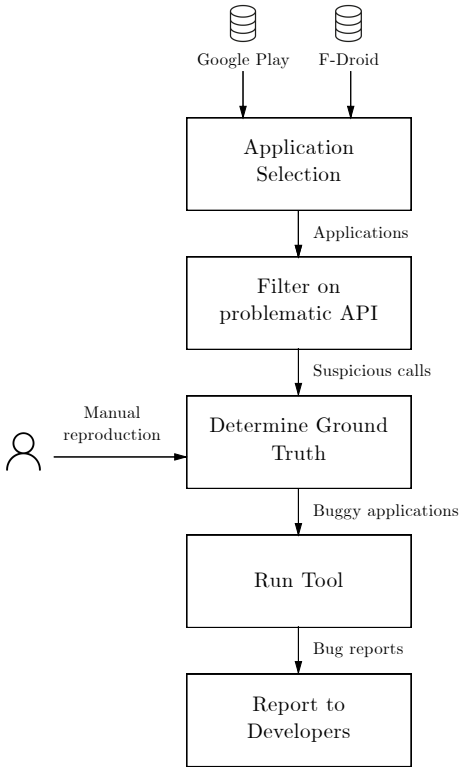

Figure 6.1: Overview of evaluation methodology

and determine the real-world usefulness of the tool. We show an overview of this evaluation methodology in Figure 6.1. The following sections describe each step in more detail.

6.3. Selection of Applications

Applications for this evaluation are automatically selected based on their availability on both F-Droid and Google Play. This condition is introduced to ensure real-world relevance and active maintenance. Their availability on F-Droid also ensures that source code is available, since this is a requirement to be listed on this repository. The availability of source code greatly aids our analysis of the applications. For every app in the F-Droid index, we attempt to find its unique identifier in the Google Play Store. In total, 1472 applications meet this criterion.

6.4. Application Filtering

We develop a script[1] to determine which applications invoke the method we have previously linked to the bug we are interested in. The script simply analyzes a compiled Android app and prints any calls of the signature we are interested in. It does not know the context of this call, whether this call is part of the application code or of a third-party library, or whether the call is protected by a preceding conditional statement in the control-flow graph. It is, therefore, a rough indication of which apps to inspect further, not an accurate diagnostic tool. It, by design, suffers from a high false-positive rate but, crucially, a low false-negative rate. If an application does not invoke this method, we define it to be out of scope for our analysis.

[1]This 'script' is actually a simple static analyzer built using the Soot framework. However, for the purpose of disambiguating it from our diagnostic tool, which is also a Soot static analyzer, albeit a more sophisticated one, we will refer to it as a 'script' here.

For the filtered applications, we collect the following metrics: number of commits, number of stars on the repository, and download count on Google Play. This data can provide insights into the apps' maintenance status and popularity.

6.5. Ground Truth Determination

For each flagged app in the set produced by the filtering process, we attempt to reproduce the issue. To this end, we load the application onto an emulated Pixel 7 device running Android 8.1 Oreo. We use the output of the filtering script to determine which parts of the application to pay particular attention to. For example, if the source code calls the problematic API in a `DownloadManager` class, we attempt to navigate to functionalities that relate to downloading data. In a mapping app, this could include tapping a menu item reading "Manage Map Downloads".

For each problematic API call, we record whether we were able to find it during app exploration and, if so, whether we could reproduce the problem. If we manage to reproduce the issue, we record a screen recording or screenshot for documentation purposes.

Naturally, not all applications can be run to perform this exploration. Some apps may crash at startup, require login to an accessible service, or even fail to install onto the emulated device. We also record such instances and the reason for our failure to explore them and reproduce the bugs. We exclude them from further analysis since we do not know their ground truth.

6.6. Experimental Setup

The experimental setup involves running the diagnostic tool on each of the filtered applications and recording the output. We document the versions of the applications, the configuration of the tool and the computing environment specifications to ensure the reproducibility of the results. We monitor for any instances where the tool times out or crashes to subsequently analyze the potential cause of these problems.

We execute the tool on an Apple MacBook Pro with an octa-core M1 Pro processor and 16 GB of RAM running MacOS 14.5 and Java 1.8.

6.7. Reporting Issues

We report all non-false positives to the developers of the applications under test. This means we report confirmed true positives (where we could reproduce the issue) and unconfirmed positives (where we did not manage to reproduce the issue). However, we exclude confirmed false positives: cases where the tool returns a positive, but we assert this is incorrect.

We choose to also report non-confirmed positives because we believe that even if we are not able to run the application or find the required functionality in it, the developers might well be, by virtue of their intimate knowledge of the application and its structure.

We report the bugs in the standard format to all repositories, except if the repository specifies its own required style. If possible, we provide screen recordings or screenshots. We engage with the developers to provide clarification or provide additional information where required.

We track the following statuses for the issue reports:

- Pending
- Developer accepted

- Developer marked as ‘not a bug’
- Developer plans not to address
- Developer needs more information
- Developer fixed issue

7

Tool Results

7.1. RQ1: Prevalence of Battery Saver Animation Bugs

To determine the prevalence of Battery Saver animation bugs, we analyze a total of 1,472 applications. We test each using our diagnostic tool, and manually verify the results to ensure accuracy.

Out of the 1,472 applications analyzed, 65 apps invoke the `setIndeterminate` method we are interested in, for a total of 178 invocations. These apps have been downloaded more than 121 million times collectively on Google Play, have an average of 1,251 stars on their code repository, and an average of 5,865 commits.

We attempt to manually trigger every `setIndeterminate` invocation to determine whether it leads to a bug in the host application. Out of 178 invocations, we are able to reproduce 41 instances of the bug. We find 59 invocations that do not cause bugs, the large majority of them simply passing `false` as an argument, therefore creating a *determinate* progress bar that does not exhibit the issue we are looking for. We exclude 78 instances where we were not able to determine whether a bug occurs. The most common reasons for this are not finding the element in the app, not being able to launch the app, or not being able to get past a setup phase, such as a sign-in screen.

These data suggest that reproducible Battery Saver animation bugs occur in 2.6% of applications, highlighting the need for effective diagnostic tools to identify and address these issues.

7.2. RQ2: Effectiveness of Diagnostic Tool

The effectiveness of the diagnostic tool was measured by its ability to detect animation bugs in the applications tested. The tool detected 45 potential bugs across the 65 previously flagged applications.

The identified bugs were verified through manual reproduction. Table 7.1 and Table 7.2 present the relevant metrics.

Metric	Score
Precision	0.911
Recall	0.911
F1 Score	0.911

Table 7.1: Accuracy Metrics

	Actual Positive	**Actual Negative**
Predicted Positive	True Positives = 41	False Positives = 4
Predicted Negative	False Negatives = 4	True Negatives = 35

Table 7.2: Confusion Matrix for Diagnostic Tool

The diagnostic tool demonstrated a high level of accuracy with both precision and recall at 0.911. The F1-Score, defined as the harmonic mean of precision and recall, is also 0.911. This confirms the tool's accuracy in identifying Battery Saver animation bugs.

7.2.1. Error Analysis

We note eight errors in total: four false positives (Type I errors) and four false negatives (Type II errors).

7.2.2. False Positives

Analyzing the false positives, which occur in *OSMDashboard*, *Libre BusTO*, *WiGLE Wireless Wardriving*, and *Seadroid* [70, 55, 121, 44], we find that three out of four false positives are caused by the animated element being set to be hidden or invisible directly after creation. A further false positive is caused by the `setIndeterminate` method being called from a `Runnable` construct. As found in the systematic characterization in section 4.2, this construct may prevent the bug from occurring in some configurations, but we choose to flag it because we cannot consistently reproduce its protective effect.

7.2.3. False Negatives

Analyzing false negatives, we find two types of tool shortcomings.

Two out of four false negatives are caused by a mismatch between the *application ID* defined on the Google Play Store and within the application. Each application submitted to Google Play must contain a globally unique application ID. To prevent false positives due to third-party libraries and frameworks, our tool filters classes to be analyzed based on the unique application ID defined by the developer. It assumes that code with a different application ID is from a third party. The application ID has a reverse-DNS notation such as `com.exampledeveloper.exampleapplication`. Our tool only analyzes class names that start with this identifier to ensure that it only analyzes the application code and no third-party frameworks or libraries. This filtering is not perfect, however, since developers sometimes change the application ID of the overarching application without changing the application code to match it.

For example, the developers of the *In The Poche* application intially used the application ID `fr.gaulupeau.apps.Poche`. However, the app is listed under the different ID `fr.gaulupeau.apps.InThePoche` in the Google Play Store. When filtering classes, the tool erroneously treats the classes under `fr.gaulupeau.apps.Poche` as being non-application code and excludes them from analysis.

A further two errors are caused by `setIndeterminate` method invocations from within an enumeration construct. The traversal implementation does not currently support enumeration constructs.

7.3. RQ3: Efficiency of Diagnostic Tool

To evaluate the efficiency of the diagnostic tool, we measured the time taken to run the tool on each application. The average time taken per application was 51.6 seconds. In total, running on 65 suspected buggy applications took 56 minutes and 45 seconds. The tool ran out of available memory on a single application and

successfully executed on all others.

These data show that executing this tool is feasible both for mass-testing publicly available applications and for common use during development or continuous integration (CI) pipelines.

7.4. RQ4: Usefulness of Diagnostic Tool

While accuracy numbers are useful, they do not tell the whole story for a bug detection tool. While a tool may be accurate at identifying bugs, developers may not find the bug reports useful. This can be due to a variety of factors. For example, the bugs themselves may not have enough user impact to warrant fixing. Conversely, the bug may be disastrous, but the bug report might be too unclear to be helpful in addressing it. Therefore, it is valuable to not just assess the accuracy but also the *usefulness* of the tool.

The usefulness of the diagnostic tool was assessed by reporting the identified issues to developers using the anonymous GitHub username *graciouselectric*, which we randomly generated. Developer feedback and subsequent actions taken by developers were recorded and categorized. We reported 39 of the 41 reproducible identified bugs. Two bugs could not be reported due to non-public issue trackers. We record the developer responses in Table 7.3.

Developer Response	**Count**
Pending	29
Accepted	9
- Fixed	2
Will Not Fix	2
Total Reported	39

Table 7.3: Summary of Developer Responses

This feedback indicates that the diagnostic tool is considered useful by the developer community, as a significant number of identified issues were acknowledged, and the first issues have already been addressed.

The addressed issues are represented in Table 7.4. The developers report "*Thanks for reporting the issue and suggesting the fix!*" and "*Hi, thank you for this issue. [...] So I corrected this. When the phone is in batery [sic] saving mode the loaders are not displayed anymore. Instead a static label is displayed in the left lower corner of the screen.*"

Application Name	**Issue Tracker Entry**
Unofficial Golem.de Reader	`https://github.com/eknoes/golem-android-reader/issues/50`
Planes Android	`https://github.com/xxxcucus/planes/issues/48`

Table 7.4: Fixed Issues

Some developers have confirmed the bug but have not addressed it. These 'accepted' issues are shown in Table 7.5. All of these have received comments from developers acknowledging the bug(s). A developer for the *Wikimedia Commons* project responded by saying "*Are we sure this is specific to API levels < 28? I remember seeing this reload icon even on an Android 12 device. I don't have that device now to confirm if it still happens, but it would be good to validate for all the cases*". The developer of the *Fruit Radar* application responded by saying "*Thank you very much for reporting this! I do not have time at the moment to work on it but you are very welcome to provide a pull request!*". The maintainer of *PinPoi*

Application Name	Issue Tracker Entry
Unofficial Golem.de Reader	`https://github.com/eknoes/golem-android-reader/issues/50`
Planes Android	`https://github.com/xxxcucus/planes/issues/48`
Wikimedia Commons	`https://github.com/commons-app/apps-android-commons/issues/5710`
Mensa	`https://github.com/famoser/Mensa/issues/43`
Fruit Radar	`https://github.com/niccokunzmann/mundraub-android/issues/329`
PinPoi	`https://github.com/fvasco/pinpoi/issues/42`
OSM Dashboard	`https://github.com/OpenTracksApp/OSMDashboard/issues/369`

Table 7.5: Accepted Issues

commented *"Thank you @graciouselectric, I agree with you, this graphic behavior should be improved. I cannot provide an ETA, yet, however PR is welcome."*. The developer of *OSM Dashboard for OpenTracks* replied "*Thanks for reporting. I've never experienced it, so far. Replacing the progress bar with a text in case of disabled animations feels a bit awkward, but I'll think about it.*"

Negative developer feedback, shown in Table 7.6, consists of the feedback on a single issue reporting two instances of the bug to the developers of the *aTalk* instant messaging application. The developers eventually report that they do not intend to address the bug, mentioning "*This is android OS 8.1 battery saver implementation problem. You can raise the issue with the android development team for further advice. aTalk will not attempt to resolve this type of problem.*"

Application Name	Issue Tracker Entry
aTalk	`https://github.com/cmeng-git/atalk-android/issues/215`

Table 7.6: "Will Not Fix" Issues

8

Discussion

8.1. Discussion

In this section, we present the implications of our research and discuss limitations and future work.

8.1.1. Implications

- **Developer Assumptions** - Many developers assume that the operating system will handle most energy-saving mechanisms, leading to a false sense of security. This assumption needs to be challenged, and developers should be encouraged to implement explicit checks within their applications. More automated tools for detecting and fixing these bugs would help the developer community address them. Furthermore, both Google and the research community can play a role in educating developers on the potential bugs introduced by system features like Battery Saver.
- **Fragmentation and OEM Customizations** - The diversity of Android devices and manufacturer-specific customizations further complicate the issues identified in this work. Android fragmentation is a well-known phenomenon that has been related to several other problem, such as malware detection issues [69] and vendor-specific bugs that are hard to address [45]. Apps that function well on AOSP may encounter problems on modified versions of Android. Manufacturers are sometimes aggressive in limiting energy use to the detriment of user experience. This fragmentation necessitates more robust and adaptable development practices. Furthermore, these manufacturer-specific 'optimizations' are often undocumented.
- **Legacy Support and API Evolution** - Older versions of Android and their APIs may not be well-supported, making it difficult for developers to address Battery Saver bugs across all versions. This highlights the need for tools and practices that can handle legacy systems and evolving APIs efficiently. An overview of the work on evolving Android APIs is provided by Liu et al. [59].
- **Infrequent Use Cases and Limited Testing** - Many Battery Saver bugs only manifest in specific use cases, such as background services or location tracking, which may not be frequently tested. This underscores the importance of comprehensive testing strategies that cover a wide range of scenarios. Developers need to be aware that a modern operating system has many non-default settings that significantly affect the application runtime.
- **Tool Limitations and Manual Effort** - While diagnostic tools can help identify many issues, they are not foolproof and often require manual verification. Developers need to be prepared for the additional effort required

to confirm and address these bugs. Additionally, Android Studio does not provide warnings on any of the issues identified, as opposed to previous work like `CiD` [52], where Android Studio already warns on the use of unsupported APIs and even suggests a fix inline.

- **Future Tool Development** - The insights from this research can inform the development of more sophisticated tools that not only detect but also help fix Battery Saver-related issues. The characterization provided in this work, along with the positive reactions from developers, suggest room for future tooling development on issues not addressed in this work.

Developer Interest
In general, while developers are concerned about these energy-saving functionalities, as evidenced by numerous questions on Stack Overflow and talks at conferences, we do not see this concern reflected much in real-world code.

Several factors contribute to this lack of interest:

- **Misalignment with user expectations** - Users generally expect their applications to function seamlessly without being aware of underlying energy-saving mechanisms.
- **Complexity and learning curve** - Implementing and testing for Battery Saver compatibility adds complexity and requires significant learning effort.
- **Time and resources required** - Developers often face constraints on time and resources, making it challenging to prioritize Battery Saver issues.
- **Platform fragmentation** - The diversity of Android devices and versions increases the difficulty of ensuring correct behavior across all devices.
- **System handling** - Developers may assume that the system handles most energy-saving aspects, leading to a false sense of security.
- **Support for older Android versions** - Older APIs may not be well supported, and migrating to newer APIs can be difficult.
- **Infrequent use cases** - Some bugs appear infrequently, affecting only specific use cases such as location tracking or background fitness apps.
- **Limited developer education** - There may be a general lack of awareness and understanding of Battery Saver features among developers.

Additionally, tools developed years ago may not run on modern systems, further complicating efforts to address these issues.

8.1.2. Limitations

To contextualize this work's outcomes, we confront the limitations inherent in the proposed approach, grouping them by category.

Characterization

1. **Limited reach** - Our study focuses on Android AOSP only, excluding variants like Samsung, Xiaomi, and others. This limitation increases the generalizability of our findings. However, manufacturer-specific bugs may not be included in the results of this study, reducing completeness.
2. **Incomplete Keywords** - The identified keywords for finding problematic APIs may be incomplete. We addressed this by collecting keywords from diverse sources, including Android documentation and GitHub issues, to create a more comprehensive set.
3. **Impact Assessment** - The scale on which the impact of bugs was assessed is open to debate. We based the scale on the impact scale used at Atlassian. Impact assessment of software bugs is not standardized within the software

industry. To mitigate this, we present the data informing the impact score so the reader may apply their own impact scoring system.

Tool Evaluation

1. **Insufficient Representativeness of Selected Apps** - The selected apps may not fully represent the diversity of the Android app ecosystem because We only select open-source applications. To mitigate this, we ensure the selected applications are also available on Google Play.
2. **Manual Inspection** - The potential inaccuracy from manual inspection is a concern. To address this, the author discussed unclear cases with colleagues to reach a consensus.
3. **Issue Reproduction** - Reproducing issues may introduce errors if there is an unclear link between the application user interface and the relevant code. It may seem that a certain misbehaving element is caused by a line of code. However, to be sure, we would need to attach a debugger, which is complicated for the pre-built Android Package (APK) files the tool ingests. To mitigate this, we are conservative with the reproduction status and only label an issue as reproduced if we see a clear link between the code and the misbehaving element on screen.

Tool Accuracy

1. **Variable resolution** - The diagnostic tool does not resolve variables. Therefore, it does not support assigning the result of a check to a variable, nor does it detect problematic calls in closures assigned to variables. Listing 8.1 shows a code example demonstrating the former. While this is a valid way of protecting the `setIndeterminate` call, our tool would raise a false positive here because it does not consider the `safe` variable as representing the status of the `areAnimatorsEnabled()` check.

Listing 8.1: Assigning the check to a variable

```
1
2 boolean safe = ValueAnimator.areAnimatorsEnabled();
3
4 if (safe) {
5     // Would be flagged by our tool as unprotected
6     progressBar.setIndeterminate(true)
7 }
```

2. **Enumeration support** - The diagnostic tool does not handle invocations from within enumeration constructs.
3. **Declarative layout files** - The diagnostic tool does not parse and analyze XML layout or Jetpack Compose files, which are a common way of specifying the user interface for an Android application.
4. **Other Progress Bar APIs** - The diagnostic tool solely identifies problems with the `android.widget.ProgressBar` API. There are other APIs for creating animating (progress) elements, but since bugs surrounding these were not found during the characterization process, we do not address these APIs here.

8.1.3. Future Directions and Recommendations

Future work can explore several areas:

- **Improving Test Oracles** - Developing test oracles for Battery Saver bugs remains challenging and requires expert knowledge. Several issues identified in this work can not currently be detected automatically due to a lack of oracles. Further research could focus on improving this process, for example, by applying machine learning to create accurate oracles.

- **Addressing Fragmentation** - Future work should consider manufacturer-dependent Battery Saver bugs, as apps that work on AOSP may not function properly on customized Android versions [31]. A challenge in this regard is the lack of documentation and the diversity of OEM approaches to battery-saving techniques. While AOSP source code is available for reference [35], the OEM modifications to it are usually not. These manufacturer-dependent Battery Saver issues are often hard to diagnose and address.
- **Improving Accuracy and Sensitivities** - Enhance the precision of the diagnostic tools to reduce false positives and false negatives. This can involve refining the static analysis algorithms to include more sensitivities.
- **Guided Exploration** - Guided exploration tools could be applied to help reproduce issues identified by our diagnostic tool. Furthermore, it could be used to corroborate rarely reported issues.
- **Automating Fixes for Common Issues** - Future research could focus on automating the remediation of detected Battery Saver issues. Integrating automated refactoring tools that apply known fixes to common problems could save developers significant time and effort.

9

Conclusion

This study provides a systematic characterization of Battery Saver bugs across various Android APIs. Each identified issue presents unique challenges and requires specific developer interventions to mitigate impacts on app functionality and user experience. As Android's power management features continue to evolve, ongoing adaptation from application developers will be essential to ensure smooth and reliable app operations under restricted power conditions.

This work provides a comprehensive examination of the challenges posed by Battery Saver mode on Android devices, particularly focusing on its impact on various APIs and the resultant bugs that can significantly affect user experience and app functionality. Through systematic characterization, we identify and analyze 13 distinct issue types, highlighting their severity, frequency, and the potential difficulties developers face in mitigating these problems.

One of the primary findings is that Battery Saver mode introduces a variety of bugs that are often not well-documented, making them difficult for developers to anticipate and address. Our research identified 13 distinct issue types, with the majority classified as having major to critical impact on app performance. This underlines the importance of developers being vigilant and proactive in testing their applications under different power-saving conditions.

The development and evaluation of a diagnostic tool to detect one of these characterized issues, specifically related to animations disabled in Battery Saver mode, showcases the practical application of our findings. We automatically analyze 1,472 open-source Android applications and identify 45 separate issues across 40 applications. The tool demonstrates a high level of precision and recall, both at 0.911, and successfully identifies nine bugs that were confirmed by developers. Two of these bugs have already been addressed. This not only validates the tool's effectiveness but also emphasizes the usefulness of the diagnostic tool in the development life cycle.

Future work should focus on expanding the capabilities of this diagnostic tool to cover more of the identified issues. Additionally, further research into improving test oracles and addressing fragmentation caused by manufacturer-specific modifications to Android could provide deeper insights and more robust solutions. The continued evolution of Android's power management features necessitates ongoing adaptation and vigilance from the developer community to ensure a reliable user experience.

In conclusion, this thesis has laid the groundwork for better understanding and mitigating the impacts of Battery Saver mode on Android applications. Developers

cannot assume that their apps will work well in this mode and must test it carefully. Both developer education and up-to-date tooling is required for identifying and addressing these bugs. By providing both a detailed characterization of the issues and a practical tool for their detection, we hope to contribute to more resilient and user-friendly app experiences in the face of increasingly sophisticated power management systems.

References

[1] Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. "Systematic execution of android test suites in adverse conditions". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 2015, pp. 83–93.

[2] Steven Arzt et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps". In: *ACM sigplan notices* 49.6 (2014), pp. 259–269.

[3] Atlassian. *Understanding incident severity levels*. url: https://www.atlassian.com/incident-management/kpis/severity-levels.

[4] Automattic. *Push notifications delayed or missing*. https://github.com/Automattic/pocket-casts-android/issues/1615. Accessed: 2024-06-07. 2023.

[5] Roberto Baldwin. *Google introduces Android 5.0 Lollipop*. Oct. 2014. url: https://thenextweb.com/news/google-introduces-android-5-0-lollipop.

[6] *Battery Saving Tips - Ride with GPS*. Dec. 2023. url: https://support.ridewithgps.com/hc/en-us/articles/14156247374747-Battery-Saving-Tips.

[7] Android Developers Blog. *Android 14 is live in AOSP - Android Developers Blog*. url: https://android-developers.googleblog.com/2023/10/android-14-is-live-in-aosp.html.

[8] Jorge Cardoso. "How to measure the control-flow complexity of web processes and workflows". In: *Workflow handbook* 2005 (2005), pp. 199–212.

[9] Ting Chen, Haiyang Tang, Xiaodong Lin, Kuang Zhou, and Xiaosong Zhang. "Silent Battery Draining Attack against Android Systems by Subverting Doze Mode". In: *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE. 2016, pp. 1–6.

[10] Xiang Chen, Yiran Chen, Zhan Ma, and Felix CA Fernandes. "How is energy consumed in smartphone display applications?" In: *Proceedings of the 14th Workshop on Mobile Computing Systems and Applications*. 2013, pp. 1–6.

[11] Commons-App. *[bug]: Confusing loading state in battery saver on older versions of Android · issue #5710 · commons-app/apps-android-commons*. url: https://github.com/commons-app/apps-android-commons/issues/5710.

[12] Flutter Community. *[Bug]: Incorrect connection status on Galaxy S10e in power saving mode*. https://github.com/fluttercommunity/plus_plugins/issues/2144. Accessed: 2024-06-07. 2024.

[13] Pranab Dash and Y Charlie Hu. "How much battery does dark mode save? An accurate OLED display power profiler for modern smartphones". In: *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*. 2021, pp. 323–335.

[14] Dimtion. *Fails to share a link when marshmallow's battery saver is ON*. https://github.com/dimtion/Shaarlier/issues/22. Accessed: 2024-06-07. 2024.

[15] Android Developer Documentation. *AlarmManager*. https://developer.android.com/reference/android/app/AlarmManager. Accessed: 2024-06-07. 2024.

[16] Android Developer Documentation. *Android 10 highlights*. https://developer.android.com/about/versions/10/highlights. Accessed: 2024-06-07. 2024.
[17] Android Developer Documentation. *Android 7.0 Behavior Changes*. url: https://developer.android.com/about/versions/nougat/android-7.0-changes.
[18] Android Developer Documentation. *Android Pie Power Management*. url: https://developer.android.com/about/versions/pie/power%5C#battery-saver.
[19] Android Developer Documentation. *App Standby Buckets - Android Documentation*. url: https://developer.android.com/topic/performance/appstandby.
[20] Android Developer Documentation. *AppCompatDelegate*. https://developer.android.com/reference/androidx/appcompat/app/AppCompatDelegate. Accessed: 2024-06-07. 2024.
[21] Android Developer Documentation. *Background Execution Limits*. url: https://developer.android.com/about/versions/oreo/background%5C#broadcasts.
[22] Android Developer Documentation. *Define work requests*. https://developer.android.com/develop/background-work/background-tasks/persistent/getting-started/define-work. Accessed: 2024-06-07. 2024.
[23] Android Developer Documentation. *Foreground services*. https://developer.android.com/develop/background-work/services/foreground-services?hl=en. Accessed: 2024-06-07. 2024.
[24] Android Developer Documentation. *Frame rate*. https://developer.android.com/media/optimize/performance/frame-rate. Accessed: 2024-06-07. 2024.
[25] Android Developer Documentation. *Implement dark theme*. https://developer.android.com/develop/ui/views/theming/darktheme. Accessed: 2024-06-07. 2024.
[26] Android Developer Documentation. *JobParameters*. https://developer.android.com/reference/kotlin/android/app/job/JobParameters. Accessed: 2024-06-07. 2023.
[27] Android Developer Documentation. *Optimize for Doze and App Standby - Android Documentation*. url: https://developer.android.com/training/monitoring-device-state/doze-standby.
[28] Android Developer Documentation. *Power management restrictions*. https://developer.android.com/topic/performance/power/power-details. Accessed: 2024-06-07. 2023.
[29] Android Developer Documentation. *Schedule alarms*. url: https://developer.android.com/training/scheduling/alarms%5C#inexact-after-specific-time.
[30] Android Developer Documentation. *UI/Application Exerciser Monkey*. url: https://developer.android.com/studio/test/other-testing-tools/monkey.
[31] *Don't Kill My App*. url: https://dontkillmyapp.com/.
[32] dotnet. *MAUI issue 19000*. https://github.com/dotnet/maui/issues/19000. Accessed: 2024-06-07. 2023.
[33] dotnet. *MAUI issue 19002*. https://github.com/dotnet/maui/issues/19002. Accessed: 2024-06-07. 2023.
[34] dotnet. *Port Forms Animation/Power Save test*. https://github.com/dotnet/maui/issues/16511. Accessed: 2024-06-07. 2023.
[35] *Download the Android Source*. url: https://source.android.com/docs/setup/download.
[36] Umar Farooq, Zhijia Zhao, Manu Sridharan, and Iulian Neamtiu. "Livedroid: Identifying and preserving mobile app state in volatile runtime envi-

ronments". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–30.

[37] *Fast and resilient web apps: Tools and techniques - Google I/O 2016*. May 2016. url: https://www.youtube.com/watch?v=aqvz5Oqs238&t=1185s.

[38] forrestguice. *Alarm not working in battery saver mode*. https://github.com/forrestguice/SuntimesWidget/issues/726. Accessed: 2024-06-07. 2020.

[39] Utkarsh Goel, Stephen Ludin, and Moritz Steiner. "Web performance with android's battery-saver mode". In: *arXiv preprint arXiv:2003.06477* (2020).

[40] Preston Gralla. *Galaxy S II: The missing manual*. O'Reilly Media, 2012.

[41] GSMArena. *Samsung Galaxy S5 - GSM Arena*. url: https://www.gsmarena.com/samsung_galaxy_s5-6033.php.

[42] Tianxiao Gu et al. "Practical GUI testing of Android applications via model abstraction and refinement". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE. 2019, pp. 269–280.

[43] Wunan Guo, Zhen Dong, Liwei Shen, Wei Tian, Ting Su, and Xin Peng. "Detecting and fixing data loss issues in Android apps". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2022, pp. 605–616.

[44] Haiwen. *Haiwen/seadroid: Android client for Seafile*. url: https://github.com/haiwen/seadroid.

[45] Dan Han, Chenlei Zhang, Xiaochao Fan, Abram Hindle, Kenny Wong, and Eleni Stroulia. "Understanding android fragmentation with topic analysis of vendor-specific bugs". In: *2012 19th Working Conference on Reverse Engineering*. IEEE. 2012, pp. 83–92.

[46] HTC. *HTC Desire User Guide*. 2010. url: http://files.customersaas.com/files/Manual/HTC_A8181_Desire_User_manual.pdf.

[47] K0shk0sh. *Notifications do not work (yet again)*. https://github.com/k0shk0sh/FastHub/issues/2867. Accessed: 2024-06-07. 2024.

[48] Chang Hwan Peter Kim, Daniel Kroening, and Marta Kwiatkowska. "Static program analysis for identifying energy bugs in graphics-intensive mobile apps". In: *2016 IEEE 24th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2016, pp. 115–124.

[49] Marcos Lerides et al. "Environmental and social impacts of Li-ion batteries". In: (2021).

[50] Ding Li, Angelica Huyen Tran, and William GJ Halfond. "Making web applications more energy efficient for OLED smartphones". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 527–538.

[51] Li Li. *Awards - Li Li*. url: http://lilicoding.github.io/awards.html.

[52] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. "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*. 2018, pp. 153–163.

[53] Li Li, Jun Gao, Tegawendé F Bissyandé, Lei Ma, Xin Xia, and Jacques Klein. "Characterising deprecated android apis". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. 2018, pp. 254–264.

[54] Li Li et al. "Static analysis of android apps: A systematic literature review". In: *Information and Software Technology* 88 (2017), pp. 67–95.

[55] *LibreBusTO · libre-busto*. url: https://gitpull.it/source/libre-busto/.

[56] linhvovan29546. *Issue 38 in React Native Full Screen Notification Incoming Call*. https://github.com/linhvovan29546/react-native-full-screen-notification-incoming-call/issues/38. Accessed: 2024-06-07. 2023.

[57] Pei Liu, Li Li, Yichun Yan, Mattia Fazzini, and John Grundy. "Identifying and characterizing silently-evolved methods in the android API". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE. 2021, pp. 308–317.

[58] Pei Liu, Yanjie Zhao, Haipeng Cai, Mattia Fazzini, John Grundy, and Li Li. "Automatically detecting api-induced compatibility issues in android apps: A comparative analysis (replicability study)". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2022, pp. 617–628.

[59] Pei Liu, Yanjie Zhao, Mattia Fazzini, Haipeng Cai, John Grundy, and Li Li. "Automatically Detecting Incompatible Android APIs". In: *ACM Transactions on Software Engineering and Methodology* (2023).

[60] Yifei Lu, Minxue Pan, Juan Zhai, Tian Zhang, and Xuandong Li. "Preference-wise testing for android applications". In: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2019, pp. 268–278.

[61] lukef. *Animations should disable or degrade when Battery Saver Mode is enabled (Android) · issue 11436 · flutter/flutter*. July 2017. url: https://github.com/flutter/flutter/issues/11436.

[62] Zhengwei Lv, Chao Peng, Zhao Zhang, Ting Su, Kai Liu, and Ping Yang. "Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–5.

[63] Organic Maps. *[android] Maps download/update is interrupted in the background when power saving settings are enabled*. https://github.com/organicmaps/organicmaps/issues/5586. Accessed: 2024-06-07. 2024.

[64] Mendhak. *Battery Saver Kills App*. https://github.com/mendhak/gpslogger/issues/813. Accessed: 2024-06-07. 2024.

[65] Solana Mobile. *When Android battery saver mode is enabled, incoming MWA connections from Chrome fail*. https://github.com/solana-mobile/mobile-wallet-adapter/issues/335. Accessed: 2024-06-07. 2024.

[66] mohamnexus. *Couldn't check ping when power saving is on*. https://github.com/stealthcopter/AndroidNetworkTools/issues/63. Accessed: 2024-06-07. 2020.

[67] Quenton Narcisse. *Adults in the U.S. prefer their smartphones instead of sex*. Feb. 2023. url: https://www.complex.com/pop-culture/a/quenton-narcisse/adults-prefer-smartphones-over-sex.

[68] Nextcloud. *Add a 'force upload/sync' button*. https://github.com/nextcloud/android/issues/7019. Accessed: 2024-06-07. 2024.

[69] Long Nguyen-Vu, Jinung Ahn, and Souhwan Jung. "Android fragmentation in malware detection". In: *Computers & Security* 87 (2019), p. 101573.

[70] OpenTracksApp. *OpenTracksApp/Osmdashboard: Openstreetmaps dashboard for OpenTracks*. url: https://github.com/OpenTracksApp/OSMDashboard.

[71] Alessandro Orso and Gregg Rothermel. "Software testing: a research travelogue (2000–2014)". In: *Future of Software Engineering Proceedings*. 2014, pp. 117–132.

[72] Stack Overflow. *Android 9: How to get locations with power saving mode and screen off*. https://stackoverflow.com/questions/52907856/android-9-how-to-get-locations-with-power-saving-mode-and-screen-off. Accessed: 2024-06-07. 2024.

[73] Stack Overflow. *Audio call stops working when android app goes in background and device is on battery*. https://stackoverflow.com/questions/

66238061/audio-call-stops-working-when-android-app-goes-in-background-and-device-is-on-ba. Accessed: 2024-06-07. 2024.

[74] Stack Overflow. *Manage internet connection on android nougat or android oreo with enabled battery*. https://stackoverflow.com/questions/46287015/manage-internet-connection-on-android-nougat-or-android-oreo-with-enabled-batter. Accessed: 2024-06-07. 2024.

[75] Stack Overflow. *Run background services on battery saver mode android*. https://stackoverflow.com/questions/49639229/run-background-services-on-battery-saver-mode-android. Accessed: 2024-06-07. 2024.

[76] Stack Overflow. *Unable to load images for the firebase notificationData messages in background*. https://stackoverflow.com/questions/57358894/unable-to-load-images-for-the-firebase-notificationdata-messages-in-background. Accessed: 2024-06-07. 2024.

[77] Jens F Peters, Manuel Baumann, Benedikt Zimmermann, Jessica Braun, and Marcel Weil. "The environmental impact of Li-Ion batteries and the role of key parameters–A review". In: *Renewable and Sustainable Energy Reviews* 67 (2017), pp. 491–506.

[78] Nikhil Rastogi. *Battery saver and vibration - Google Pixel Phone Help*. url: https://support.google.com/pixelphone/thread/197296069?hl=en&msgid=197468537.

[79] Realm. *Realm SyncManager takes up to 4-5 minutes to reconnect to ROS after waking device*. https://github.com/realm/realm-java/issues/7003. Accessed: 2024-06-07. 2024.

[80] Oliviero Riganelli, Simone Paolo Mottadelli, Claudio Rota, Daniela Micucci, and Leonardo Mariani. "Data loss detector: automatically revealing data loss bugs in Android apps". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2020, pp. 141–152.

[81] Alireza Sadeghi, Reyhaneh Jabbarvand, and Sam Malek. "Patdroid: permission-aware gui testing of android". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 220–232.

[82] Samsung. *The S23 Series Battery: How powerful is it? - Samsung Support*. Feb. 2023. url: https://www.samsung.com/ae/support/mobile-devices/the-s23-series-battery-how-powerful-is-it/.

[83] Dan Seifert. *Google announces Android 5.0 Lollipop*. Oct. 2014. url: https://www.theverge.com/2014/10/15/6982167/google-android-5-0-l-lollipop-announcement-release.

[84] Seva-coder. *[Feature] Ignore battery saver state*. https://github.com/Seva-coder/Finder/issues/29. Accessed: 2024-06-07. 2024.

[85] Zhiyong Shan, Tanzirul Azim, and Iulian Neamtiu. "Finding resume and restart errors in android applications". In: *ACM SIGPLAN Notices* 51.10 (2016), pp. 864–880.

[86] simplex-chat. *Push notifications not working when battery saver is on*. https://github.com/simplex-chat/simplex-chat/issues/2135. Accessed: 2024-06-07. 2022.

[87] StackOverflow. *Android check connectivity on low battery power saver mode*. https://stackoverflow.com/questions/60821414/android-check-connectivity-on-low-battery-power-saver-mode. Accessed: 2024-06-07. 2020.

[88] StackOverflow. *Background service stops in battery saver mode in Android R*. https://stackoverflow.com/questions/63459111/background-service-stops-in-battery-saver-mode-in-android-r. Accessed: 2024-06-07. 2020.

[89] StackOverflow. *Battery saver and night mode on Android 9 Pie relaunches whole activity, how to prevent it*. https://stackoverflow.com/questio

ns/55397866/battery-saver-and-night-mode-on-android-9-pie-relaunches-whole-activity-how-t. Accessed: 2024-06-07. 2019.

[90] StackOverflow. *Battery saver feature killing my music service.* https://stackoverflow.com/questions/36263882/battery-saver-feature-killing-my-music-service. Accessed: 2024-06-07. 2016.

[91] StackOverflow. *Can not check network info programmatically in battery save mode in Android device.* https://stackoverflow.com/questions/51878434/can-not-check-network-info-programmatically-in-battery-save-mode-in-android-devi. Accessed: 2024-06-07. 2018.

[92] StackOverflow. *Custom dialog crashing after theme change.* https://stackoverflow.com/questions/58016100/custom-dialog-crashing-after-theme-change. Accessed: 2024-06-07. 2019.

[93] StackOverflow. *FragmentTransaction setCustomAnimations doesnt work when battery saver is on.* https://stackoverflow.com/questions/35489828/fragmenttransaction-setcustomanimations-doesnt-work-when-battery-saver-is-on. Accessed: 2024-06-07. 2016.

[94] StackOverflow. *How force the app to opt out of battery saver mode when the service is on.* https://stackoverflow.com/questions/53115473/how-force-the-app-to-opt-out-of-battery-saver-mode-when-the-service-is-on. Accessed: 2024-06-07. 2018.

[95] StackOverflow. *How to avoid Android application UI changes on low battery level.* https://stackoverflow.com/questions/71480001/how-to-avoid-android-application-ui-changes-on-low-battery-level. Accessed: 2024-06-07. 2022.

[96] StackOverflow. *How to avoid having my foreground service stopped when on battery power saving mode.* https://stackoverflow.com/questions/36922608/how-to-avoid-having-my-foreground-service-stopped-when-on-battery-power-saving-m. Accessed: 2024-06-07. 2016.

[97] StackOverflow. *Is there any way to read the battery permissions of Xiaomi/Redmi for background.* https://stackoverflow.com/questions/57302686/is-there-any-way-to-read-the-battery-permissions-of-xioamiredmi-for-background. Accessed: 2024-06-07. 2019.

[98] StackOverflow. *LocationSettings dialog appears even if GPS is turned on.* https://stackoverflow.com/questions/55488779/locationsettings-dialog-appears-even-if-gps-turn-on. Accessed: 2024-06-07. 2019.

[99] StackOverflow. *Object Animator not working in battery saver mode post-JellyBean/Android 5.x.* https://stackoverflow.com/questions/36548097/object-animator-not-working-in-battery-saver-mode-post-jellybeanandroid-5-x. Accessed: 2024-06-07. 2016.

[100] StackOverflow. *ProgressBar disappears in battery saver mode Android 5.x.* https://stackoverflow.com/questions/35221706/progressbar-disappears-in-battery-saver-mode-android-5-x. Accessed: 2024-06-07. 2016.

[101] StackOverflow. *Run background services on battery saver mode Android.* https://stackoverflow.com/questions/49639229/run-background-services-on-battery-saver-mode-android. Accessed: 2024-06-07. 2018.

[102] StackOverflow. *SetAlarmClock is not exact and system adjusts the time for it.* https://stackoverflow.com/questions/72242900/setalarmclock-is-not-exact-and-system-adjusts-the-time-for-it. Accessed: 2024-06-07. 2022.

[103] StackOverflow. *Unable to send location updates to server because of battery saver mode in MIUI.* https://stackoverflow.com/questions/53537284/unable-to-send-location-updates-to-server-because-of-battery-saver-mode-in-miui. Accessed: 2024-06-07. 2018.

[104] StackOverflow. *ValueAnimator doesnt work as expected when battery saver is enabled API 21.* https://stackoverflow.com/questions/38483783/

valueanimator-doesnt-work-as-expected-when-battery-saver-is-enabled-api-21. Accessed: 2024-06-07. 2016.

[105] Ting Su et al. "Guided, stochastic model-based GUI testing of Android apps". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 245–256.

[106] Jingling Sun et al. "Characterizing and Finding System Setting-Related Defects in Android Apps". In: *IEEE Transactions on Software Engineering* (2023).

[107] Jingling Sun et al. "Understanding and finding system setting-related defects in Android apps". In: *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2021, pp. 204–215.

[108] Google Support. *Use Battery Saver on a Pixel phone*. url: https://support.google.com/pixelphone/answer/6187458?hl=en.

[109] Geoffrey A. Fowler The Washington Post. *It's Not your imagination: Phone battery life is getting worse*. Nov. 2018. url: https://eu.heraldnews.com/story/news/2018/11/01/it-s-not-your-imagination/9374880007/%5C#:~:text=%22Batteries%20improve%20at%20a%20very,up%20faster%20than%205%20percent.%22.

[110] thunderbird. *Issue 7416 in Thunderbird Android*. https://github.com/thunderbird/thunderbird-android/issues/7416. Accessed: 2024-06-07. 2023.

[111] *Troubleshooting GPS Issues*. Mar. 2024. url: https://support.strava.com/hc/en-us/articles/216918967-Troubleshooting-GPS-Issues.

[112] TylerCarberry. *Issue 161 in 2048-Battles*. https://github.com/TylerCarberry/2048-Battles/issues/161. Accessed: 2024-06-07. 2022.

[113] Tytydraco. *Is there plans for android 12 support?* https://github.com/tytydraco/Buoy/issues/12. Accessed: 2024-06-07. 2024.

[114] *Use low power mode to save battery life on your iPhone or iPad*. url: https://support.apple.com/en-us/HT205234.

[115] Matteo Varvello and Benjamin Livshits. "On the battery consumption of mobile browsers". In: *arXiv preprint arXiv:2009.03740* (2020).

[116] Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. "Towards Verifying Android Apps for the Absence of {No-Sleep} Energy Bugs". In: *2012 Workshop on Power-Aware Computing and Systems (HotPower 12)*. 2012.

[117] Jane Wakefield. *People devote third of waking time to mobile apps*. Jan. 2022. url: https://www.bbc.com/news/technology-59952557%5C#:~:text=People%20are%20spending%20an%20average,its%20research%20included%20watching%20TV..

[118] Sinan Wang et al. "Aper: evolution-aware runtime permission misuse detection for Android apps". In: *Proceedings of the 44th International Conference on Software Engineering*. 2022, pp. 125–137.

[119] Ying Wang et al. "Runtime permission issues in android apps: Taxonomy, practices, and ways forward". In: *IEEE Transactions on Software Engineering* 49.1 (2022), pp. 185–210.

[120] Lili Wei, Yepang Liu, and Shing-Chi Cheung. "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 2016, pp. 226–237.

[121] Wiglenet. *Wiglenet/wigle-WIFI-wardriving: Nethugging client for Android, from Wigle.net*. url: https://github.com/wiglenet/wigle-wifi-wardriving.

[122] *Wikimedia Commons*. url: https://play.google.com/store/apps/details?id=fr.free.nrw.commons&hl=en.

[123] Xamarin. *Issue with battery saver mode*. https://github.com/xamarin/Xamarin.Forms/issues/8382#issuecomment-54999G11. Accessed: 2024-06-07. 2023.

[124] Ybq. *Not showing in Battery Saver mode in android 7+ | issue #59 | ybq/android-spinkit*. url: https://github.com/ybq/Android-SpinKit/issues/59.

[125] ybq. *Issue 59 in Android-SpinKit*. https://github.com/ybq/Android-SpinKit/issues/59. Accessed: 2024-06-07. 2018.

[126] Razieh Nokhbeh Zaeem, Mukul R Prasad, and Sarfraz Khurshid. "Automated generation of oracles for testing user-interaction features of mobile apps". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE. 2014, pp. 183–192.

[127] Yuhao Zhou and Wei Song. "DDLDroid: A Static Analyzer for Automatically Detecting Data Loss Issues in Android Applications". In: *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 2023, pp. 1471–1474.

[128] Micah S Ziegler and Jessika E Trancik. "Re-examining rates of lithium-ion battery technology improvement and cost decline". In: *Energy & Environmental Science* 14.4 (2021), pp. 1635–1651.