FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

E-APK: Energy Pattern Detection In Decompiled Android Applications

Nelson Alexandre Saraiva Gregório

Mestrado em Engenharia Informática e Computação

Supervisor: João Carlos Viegas Martins Bispo

Supervisor: João Paulo de Sousa Ferreira Fernandes

Supervisor: Sérgio Queiroz de Medeiros

October 21, 2022

E-APK: Energy Pattern Detection In Decompiled Android Applications

Nelson Alexandre Saraiva Gregório

Mestrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

October 21, 2022

Resumo

A eficiência energética é um requisito não-funcional que os programadores devem considerar. Este requisito é particularmente relevante na construção de software para dispositivos que operam com bateria como os telemóveis: a longa duração da bateria é um requisito essencial para o utilizador ter uma experiência agradável.

Foi demonstrado que muitas aplicações móveis contêm ineficiências que fazem com que a bateria seja descarregada mais rapidamente do que o necessário. Algumas destas ineficiências resultam de padrões de software que têm sido catalogados na literatura. Os catálogos normalmente fornecem alternativas energeticamente mais eficientes.

Embora a literatura relacionada seja vasta, as abordagens até agora assumem como requisito fundamental que se tenha acesso ao código fonte de uma aplicação a fim de se poder analisá-la. Este requisito torna a análise energética independente uma tarefa desafiante, ou mesmo impossível, por exemplo, para um consumidor ou, mais apropriadamente, para uma App Store que tente fornecer informações sobre a eficiência de uma aplicação a ser submetida para publicação.

O nosso trabalho estuda a viabilidade de procurar padrões energéticos nas aplicações, descompilando-as e analisando o código resultante. Para tal, descompilamos e analisamos 420 aplicações de código aberto. Estendemos uma ferramenta existente para ajudar neste processo, tornando-a capaz de descompilar de forma transparente e analisar aplicações Android. Com os dados recolhidos, efectuamos uma estudo comparativa da presença de padrões energéticos entre o código fonte e o código descompilado.

Efectuamos dois tipos de análise: i) comparando o número total de deteções; ii) comparando a semelhança entre as deteções. Ao comparar o número total de deteções no código fonte com o código descompilado, descobrimos que aproximadamente 79,05% das aplicações reportaram o mesmo número de deteções.

Para testar a semelhança entre o código fonte e os *APK*s, calculamos, para cada aplicação, uma pontuação de semelhança baseada nos nossos quatro detetores implementados. De todas as aplicações, 34,53% obtiveram uma pontuação perfeita de similaridade com valor 4, e 89,47% obtiveram uma pontuação de 3 ou mais em 4. Além disso, apenas duas aplicações obtiveram uma classificação de 0.

Quando analisados em conjunto, os resultados das duas análises que realizámos apontam numa direção promissora. Acreditamos que as técnicas de análise estática, tipicamente utilizadas em código fonte, podem ser um método viável para inspeccionar *APK*s quando o acesso ao código fonte é restrito, e é portanto valioso continuar a investigação nesta área.

Abstract

Energy efficiency is a non-functional requirement that developers must consider. This requirement is particularly relevant when building software for battery-operated devices like mobile ones: a long-lasting battery is an essential requirement for an enjoyable user experience.

It has been shown that many mobile applications include inefficiencies that cause battery to be drained faster than necessary. Some of these inefficiencies result from software patterns that have been catalogued in the literature. The catalogues often provide more energy-efficient alternatives.

While the related literature is vast, the approaches so far assume as a fundamental requirement that one has access to the source code of an application in order to be able to analyse it. This requirement makes independent energy analysis challenging, or even impossible, e.g. for a mobile user or, most significantly, an App Store trying to provide information on how efficient an application being submitted for publication is.

Our work studies the viability of looking for known energy patterns in applications by decompiling them and analysing the resulting code. For this, we decompiled and analysed 420 open-source applications. We extended an existing tool to aid in this process, making it capable of transparently decompiling and analysing android applications. With the collected data, we performed a comparative study of the presence of energy patterns between the source code and the decompiled code.

We performed two types of analysis: i) comparing the total number of detections; ii) comparing the similarity between detections. When comparing the total number of detections in source code against decompiled code, we found that approximately 79.05% of the applications reported the same number of detections.

To test the similarity between source code and *APK*s, we calculated, for each application, a similarity score based on our four implemented detectors. Of all applications, 34.53% achieved a perfect similarity score of 4, and 89.47% got a score of 3 or more out of 4. Furthermore, only two applications got a score of 0.

When viewed in tandem, the results of the two analyses we performed point in a promising direction. We believe static analysis techniques, typically used in source code, can be a viable method to inspect *APK*s when access to source code is restricted, and further research in this area is worthwhile.

Keywords: code patterns, energy efficiency, static analysis, decompiler, mobile, android, metaprogramming, compilers

Agradecimentos

Obrigado Mãe, por ainda hoje me ajudares, pela paciência e amor que nunca me faltou.
Obrigado Colegas, pela companhia e toda a parvoíce comunal que partilhamos.
Obrigado Orientadores, pelo apoio e suporte nesta jornada.

Nelson Alexandre Saraiva Gregório

À memória de Nelson Almeida Gregório

Contents

List of Figures

List of Tables

Listings

Abbreviations and Acronyms

AOT	Ahead-of-Time
API	Application Programming Interface
APK	Android Package Kit
ART	Android Runtime
AST	Abstract Syntax Tree
CLI	Command-line Interface
dex	Dalvik EXecutable
E-APK	Energy-aware Android Patterns for Kadabra
EGAP	Energy-Greedy Android Patterns
EMC	Excessive Method Calls
HMU	HashMap Usage
IDE	Integrated Development Environment
IFDS	Interprocedural Finite Distributive Subset
IG	Internal Getter
IL	Intermediate Language
JIT	Just-in-Time
MIM	Member Ignoring Method
OS	Operating System
UUID	Universally Unique Identifier

Chapter 1

Introduction

1.1 Context

Smartphones are an essential part of our lives nowadays: besides continuous communication and connection services, smartphones provide access to various productivity tools and immersive entertainment forms. However, since smartphones are most often battery-operated devices, energy efficiency is crucial to provide these services and, consequently, to achieve a satisfactory user experience. Studies show that smartphone users consider long battery life the most important feature when shopping for a new smartphone [1], which reinforces the nature of energy efficiency as a key non-functional requirement for mobile applications.

One of the avenues researchers have taken to tackle energy efficiency in the mobile space is by developing guidelines and processes to improve battery consumption in mobile applications [42, 22, 16, 12, 6]. Some of this work has culminated in defining energy patterns, software patterns with energy implications, that inform how developers should write their code while optimising for energy efficiency [23, 17, 47, 11, 16].

To assist developers in constructing more efficient applications, automatic detection and refactoring tools to inspect and optimise source code and packaged applications (*APK*s) have been developed [27, 32, 33, 25, 28, 22, 50]. Providing such tools helps reduce development costs and raises awareness of good energy practices for all stakeholders. However, most tools are designed with developers in mind and rely on source code availability, with only a few capable of analysing *APK*s through its Dex bytecode [32, 36].

1.2 Motivation

By developing tools to analyse energy efficiency, developers can hopefully improve the energy efficiency of their applications, ideally without incurring in significant development costs. However, besides developers themselves, other stakeholders may benefit from energy-efficiency analysis and

[1]`https://www.statista.com/chart/5995/the-most-wanted-smartphone-features/` (Last Access: Aug 9, 2022)

that do not necessarily have access to the source code of applications, notably smartphone users and App Stores. Taking the latter as an example, being able to analyse and compare the energy efficiency of different applications could be considered the first step towards providing energy labels for the applications being commercialised. We note that energy labels are nowadays assumed as standard[2], e.g., for home appliances, and these forms of independent analysis can further motivate developers to adopt better energy practices.

Given that App Stores usually only have access to *APK*s, we focus on static analysis of *APK*s. Research and tools already exist to analyse applications through their bytecode, so instead, we look towards static analysis of decompiled Java code as a new potential way to perform independent energy analysis of Android applications. We can take advantage of existing work related to energy patterns and their detection in Source code. If this type of analysis proves successful, since we are reconstructing source code, an added benefit is the ability to send feedback to unaware developers about possible fixes and improvements they can perform.

Altogether, by empowering App Stores and smartphone users with tools to assess energy efficiency and, more generally, energy labels and standards, we can create a feedback loop where: App Stores give energy efficiency ratings to applications; consumer awareness towards energy grows; developers allocate more time and effort for energy requirements.

1.3 Objective

Given the wealth of research on android applications' energy patterns, we set out to assess the viability of looking for these patterns in *APK*s. For this, we took 420 open-source applications, extracted from Couto et al. [16], for which we have access both to their source code and *APK*s. Our idea was then to compare in a systematic way the energy patterns that we could find in: i) the source code obtained from decompiling the *APK*s with ii) their original source code.

In essence, our aim with this work is to explore and answer the following main research question:

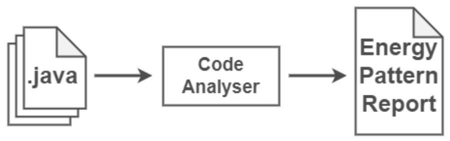
RQ *Can static analysis of decompiled code from Android applications be a viable method of detecting energy patterns?*

To implement our analysis, we extended Kadabra [3], a source-to-source compiler based on the LARA framework [13, 46]. We added a Java decompiler to Kadabra, JADX [4], making it capable of automatically decompiling *APK*s. We then developed a library of detectors, E-APK (Energy-aware Android Patterns for Kadabra), dedicated to locating energy patterns in Java source code, taking advantage of Kadabra's Abstract Syntax Tree (AST) model and its expressive query-based search system to analyse both the applications' source code and their decompiled code. By running the

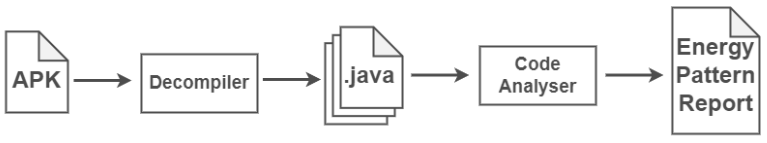
[2]https://ec.europa.eu/info/energy-climate-change-environment/standards-tools-and-labels/products-labelling-rules-and-requirements/energy-label-and-ecodesign/about_en (Last Access: Jul 26, 2022)

[3]https://specs.fe.up.pt/tools/kadabra/ (Last Access: Jul 26, 2022)

[4]https://github.com/skylot/jadx (Last Access: Jul 26, 2022)



(a) Source code analysis



(b) APK analysis

Figure 1.1: Data flows for source code and APKs

same detectors in both versions of an application's code, we can try to ascertain if patterns present in the source code are also present and visible in decompiled code. Figure 1.1 gives an overview of how the analysis is done. From the analysis process, E-APK generates a standardized report that we can use to easily compare versions.

Having analysed all 420 applications and comparing the obtained results, we found empirical evidence supporting our RQ. Our results show that the decompilation process does not have a systematic negative impact on detecting energy patterns. As for the rate of similarity, that is, how many energy pattern instances detected in source code can be detected and matched in the *APK*, with our worst performing detector reporting a 75% rate of similarity on average, we believe the type of analysis we propose to be a promising direction to pursue.

In summary, the main contributions of our work are:

- A new energy pattern detection tool implemented in Kadabra, capable of analysing source code and APKs.
- A comprehensive study with over 400 applications about detecting energy patterns in source code and APKs.
- Empirical evidence supporting the possibility of using energy pattern detection techniques in decompiled code, as a means of performing independent energy analysis.

A replication package is available in a zenodo repository [5]. It contains instructions for preparing the dataset used, executing Kadabra and E-APK, and reproducing the analysis results and graphs.

1.4 Document Structure

Over the current chapter, we presented an overview of the dissertation. We went over its context and motivation, and outlined the main objectives as well as our solution. Chapter 2 provides some

[5] https://doi.org/10.5281/zenodo.7083540 (Last Access: Aug 16, 2022)

background knowledge on the processes and tools involved in our work. In Chapter 3 we give an overview of the current research found on energy efficiency for the mobile space, particularly on Android applications. In Chapter 4 we present our solution for detecting energy patterns in decompiled applications, along with subset of patterns we selected and their detectors. Chapters 5 and 6 explain our testing methodology and the considerations we take in our experiments and discuss the results we obtained. We close with Chapter 7 by summarizing the work done so far and what can be done in the future to expand our work.

Chapter 2

Background

In this section, we present essential background knowledge and some additional technical details of the processes involved in our work. Our main objective was to study whether looking for energy patterns through static analysis, as performed on source code, can be done on decompiled code. To this end, we analysed and compared source code and decompiled code to verify if they yielded similar detection results. The workflow required for this is divided into two areas, decompilation and analysis, each with its challenges and requirements.

2.1 Inner workings of Android applications

Android applications are packaged in *APK* (*Android Package Kit*) files that contain the compiled source code in a bytecode format called *Dex* [21], short for *Dalvik Executable*, as well as resources and native code.

Initially, the android operating system used *Dalvik* [21], a virtual machine to execute *Dex* code. From Android 4.4 onwards, the operating system changed to *Android Runtime* (*ART*) [1]. The significant change between the two is that instead of using a *Just-in-Time* (*JIT*) compiler, it now compiles application code *Ahead-of-Time* (*AOT*) [26]. Essentially instead of compiling the application during its execution, this is now done during the installation process. However, applications are still distributed as *APK*s with *Dex* bytecode. Another relevant aspect to note about ART is how energy efficient it is compared to Dalvik; Georgiev et al. reported average energy savings from 0.48 Joules to 124.72 Joules across multiple benchmarks [26].

[1] `https://source.android.com/devices/tech/dalvik` (Last Access: Aug 9, 2022)

Figure 2.1 gives an overview of the lifecycle of an *APK* file. The *package* section is where our work will take effect by extracting any Dex files in the *APK* and decompiling them into Java source code. Since the changes from *Dalvik* to *ART* only affect the installation and execution phases, our work is not restrained to older or newer applications or OS versions.

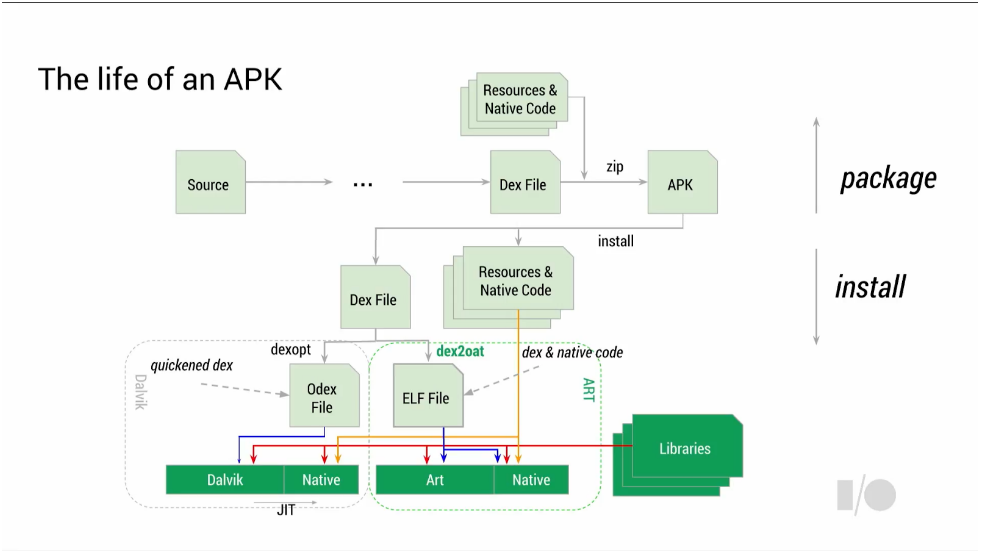


Figure 2.1: APK lifecycle[2]

2.2 Decompilation of Android applications

Decompilation can be achieved using decompilers, programs made to reverse engineer executable files and translate them into human-readable source code. In the case of Android applications, a decompiler will take the *APK*, extract the *Dex* file inside, disassemble the *Dalvik* bytecode into an Intermediate Language, like smali or Jasmin [3, 2], and finally translate that to Java source code. We list the decompilers we found and their differences in the upcoming Section 3.3.

Decompilation is a multi-step process that can fail in multiple ways, and it is not guaranteed that perfect recovery and reconstruction of the original source code is possible. In fact, decompilation can be severely hindered by obfuscation [20, 53], a process that makes source code as unreadable to humans as possible. However, just as there are compilers and decompilers, there also exist obfuscators [1, 5] and deobfuscators [8, 9].

Since obfuscators modify the code, this raises some interesting questions: What effects does obfuscation have on code patterns? Can code patterns still be detected? Our work does not measure the impact of obfuscation on energy code patterns, at least not directly, so dedicated research on the topic is a good starting point for future work.

[2] `https://youtu.be/EBlTzQsUoOw` Google I/O 2014 (Last Access: Jul 26, 2022)

In the Android ecosystem, and of particular concern to our work, two well-known obfuscators are ProGuard and, in more recent years, R8 [3]. These obfuscators can be used when creating *APK*s. When using the Gradle build system, the build property *minifyEnabled* will activate three compile-time tasks for builds: code shrinking, obfuscation, and optimization. We must therefore consider and register this property's presence during our analysis.

In the upcoming Section 3.3, we list the decompilation tools we found, some of which are specifically for Android and are also capable of deobfuscating code to some degree.

2.3 LARA and Kadabra

When deciding how to perform the analysis, our requirements lead us to the development of a custom solution. For instance, many existing tools are tied to an IDE, like Android Studio [4], and we needed a command-line tool capable of operating purely with source code files instead of whole projects. We also needed to analyse a large set of applications for testing and validation purposes, so automating the process was essential. Another crucial requirement was the capability of handling an *incomplete classpath*. Within the Java environment, an important parameter is the *classpath*, which acts as a map between Java classes and their location in the filesystem. The decompilation process can lead to an *incomplete classpath*, a situation where a reference to a class exists but not the actual class itself (and its file). Tools like Android Lint, used by some existing tools, require projects in a compilable state with a *complete classpath*.

We chose *Kadabra* [5], a source-to-source compiler based on the LARA framework [13, 46] as a base for our energy detectors. Source-to-source compilers implemented using the LARA framework have a common abstraction layer for accessing the AST, and analyses are implemented using scripts written in JavaScript. The abstraction layer is language-agnostic, so there are LARA compilers for several languages (e.g., C/C++ [10], MATLAB [48]) using the same abstraction. Kadabra, in particular, uses Spoon [44] underneath as the Java parser and AST.

With *Kadabra*, we can handle the *incomplete classpath* problem, since Spoon offers this capability [6], and perform a scalable analysis by taking advantage of parallelism and command-line interface capabilities, so it meets our main requirements. For each pattern, we can develop single implementation analysis strategies for both source code and decompiled code; this ensures the detection logic used during our tests is the same for source code and *APK*s, making the testing methodology as fair as possible. The end result is a source code analysis tool capable of seamlessly decompiling Android applications and analysing them.

[3] `https://developer.android.com/studio/build/shrink-code` (Last Access: Aug 3, 2022)
[4] `https://developer.android.com/studio` (Last Access: Jul 26, 2022)
[5] `https://specs.fe.up.pt/tools/kadabra/` (Last Access: Jul 26, 2022)
[6] `https://spoon.gforge.inria.fr/launcher.html` (Last Access: Aug 20, 2022)

Chapter 3

Related Work

Energy efficiency in mobile applications is a relatively recent research area, with most work done within the last decade [47, 32, 52, 14, 6, 17, 16, 50]. Contributions from different angles are required to improve energy efficiency as a whole. We can separate many of these contributions into three main groups:

- Development of energy profiling techniques to measure energy consumption on smartphone devices.
- Identification and cataloguing of energy patterns and good practices proven to increase energy efficiency.
- Development of tools to assist developers in identifying and implementing these patterns.

Another relevant area for our work is the decompilation of Android applications. Our strategy to identify energy patterns in *APK*s is to decompile the applications and apply techniques similar to some existing tools in the decompiled source code. We review existing decompilation tools capable of regenerating Java source code from Android package files (*APK*s).

The following subsections look into existing energy pattern catalogues, energy detection/refactoring tools, and decompilation tools.

3.1 Energy Patterns in mobile applications

Several works focus on identifying energy-greedy patterns and finding energy-efficient alternatives. As defined by Feitosa et al. [23], these energy patterns can be seen as a tuple of problem, the energy-greedy form, and solution, the energy-efficient alternative.

A wide range of patterns is documented in the following catalogues, which aggregate data from different sources using different methods. Couto et al. [16] present code patterns collected from previous literature and test them individually and in combinations. Cruz et al. [17] review commits, issues and pull requests from 1000+ applications to find and document common design patterns that affect energy efficiency. Reimann et al. [47] reference common bad practices according to android development guidelines and other sources. A catalogue is available online with

30 patterns [11]. A few works also reference, test and develop tools for patterns not included in catalogues [42, 22, 27, 12, 6].

In total, more than 60 patterns have been documented in the literature. It is also important to note that we found some overlap between catalogues, as some patterns refer to the same pair of problem/solution using different names. An example would be *Early Resource Binding*, also called *Binding Resources too early* or *Open Only When Necessary* [47, 11, 17, 42]. Table A.1 summarises the patterns found and the instances where duplicates exist.

These patterns and catalogues are a foundation for many of the tools listed in Section 3.2. Our work also takes these patterns as a starting point, but instead of the usual approach of detecting them in source code, we hope to find them in decompiled code.

3.2 Existing tools for energy efficiency in mobile applications

While documenting energy patterns is a crucial step towards more energy-efficient applications, it is only part of the solution. Catalogues by themselves only reach developers already concerned with energy. To combat this point, another vital research direction with a growing body of works is the development of tools to assist developers. Such tools help verify and improve how energy-efficient applications are without incurring in significant development costs, such as acquiring the necessary expertise to refactor code manually and extra development time.

Some important distinctions can be made between these tools:

- The broadest one is the division of detection-only tools and tools capable of refactoring the application automatically.
- Another is about who uses such tools and when are they used. Different stakeholders have different access to applications. Developers benefit from analysis throughout development and have access to source code. Tools capable of analysing source code, preferably integrated with the IDE, are a must-have. App Stores or smartphone users only have access to a compiled and packaged version of the application after its release, and tools that do not rely on source code availability are the only option in this case.

There is a visible correlation between the type of tool and the context where it is used. Many detection-only tools are standalone, using *APK*s as input and return some form of a report detailing the patterns encountered. Most refactoring tools are IDE plugins that work with the project and its source code.

3.2.1 Detection tools

Greenness [27] Android lint is a code scanning tool integrated with Android Studio that developers can extend with new functionalities. Goaër et al. propose a new category for Android Lint called Greenness in their paper. It adds energy-oriented inspections that deal with some of the existing patterns. It is the sole detection-only tool we found in an IDE plugin form, operating directly with the project and its source code.

Paprika [32] While not dealing directly with energy patterns, this tool detects code smells, object-oriented and Android-specific, which have been proven to impact energy efficiency indirectly. It is a standalone tool based in Soot [51, 38]. It uses a Soot module called Dexpler [7] to decompile an application's *APK* and, from there, builds a graph model to perform bytecode analysis.

StateDroid [52] Xu et al. propose a static analysis technique called state-taint analysis to identify resource leaks. Resources like GPS, WiFi, camera and others can consume energy continuously and should be properly closed when not in use. StateDroid is their tool implementation. It decompiles *APK*s and builds a control flow graph of the application where it performs the paper's proposed state-taint analysis implemented upon the IFDS framework [49].

SAAD [36] Similar to StateDroid [52], Hao et al. propose in their paper a static analysis technique called SAAD capable of detecting not only resource leaks but also layout defects. It decompiles *APK*s using Apktool [1] to try and detect resource leaks using the Dalvik bytecode files and search for layout defects using layout files.

PatBugs [39] Also worth mention is Lian et al. paper that propose a tool, PatBugs, to detect patterns in cross-platform mobile applications. It requires the application's source code to perform its analysis.

3.2.2 Refactoring tools

Most of these tools are implemented as an IDE extension or plugin, and collectively they cover the existing code patterns documented in the literature.

For Android studio, we found **EcoAndroid** [50], **aDoctor** [33], **RAndroid** [25], **AEON** [28], **xAL** [22]. A few tools are also integrated into the Eclipse IDE, such as, **AsyncDroid** [40], **EnergyPatch** [6], **Leafactor** [18], **Nguyen et al.** [43]. Besides these, we found four tools; some of which were developed as auxiliary tools to assist the research presented in their respective papers: **Chimera** [16], **HOT-PEPPER** [14], **EARMO** [42] and **DelayDroid** [12]. With the exception of **Chimera** [16], the remaining three have the unique distinction of working directly with the application *APK* instead of the source code, and they perform their analysis using bytecode. **Chimera** is also the work we use the most as a foundation; we take advantage of some of its documented Energy-Greedy Android Patterns (EGAPs) and the dataset they used as explained in Chapter 5.

[1] `https://github.com/iBotPeaches/Apktool` (Last Access: Jul 26, 2022)

3.3 Decompilation tools for Android Applications

Our objective is to perform static analysis in decompiled code to determine if energy patterns can be detected in *APK*s when the source code is unavailable. We found several tools to decompile *APK*s and extract relevant information and data. The tools below are actively mentioned in the literature, currently maintained and are therefore good candidates for our work.

We can categorize the tools we found in two modes, according to what inputs they were designed to receive and what outputs they can produce. Tools can take Java bytecode or Dalvik bytecode as input, and the output produced can be JVM assembly or Java code.

JD Project, *Cfr*, *Procyon*, *Krakatau* and *Fernflower* are decompilers for Java. *Procyon* and *Krakatau* in particular also work as Java bytecode assemblers/disassemblers.

dex2jar, *Dexpler*, *Apktool*, *androguard* and *Jadx* were made for Dalvik bytecode and *APK*s. *dex2jar*, *Dexpler* are exclusively assemblers/disassemblers often used in tandem with other tools (some presented in 3.2). *Apktool* and *androguard* perform a variety of tasks on *APK* resources and bytecode. *Jadx* is a purposely built dex to java decompiler and can also decode *APK* resources like the AndroidManifest.xml.

Table 3.1 lists all the tools mentioned above with links to their respective repositories. It also contains the date of the last commit, where we can see they are all reasonably recent, with some being actively developed. Lastly, we also include the publications we found that mention these tools.

We did not test all of these tools, but we chose *Jadx* as our decompilation tool. It met our requirements of being actively developed, capable of selectively extracting the classes we want and featuring deobfuscation mechanisms. *Jadx* is made explicitly for decompiling *APK*s, and studies show it as the best performing decompiler available [41, 35]. Another crucial advantage is the possibility of (and instructions on) using JADX as a library. Integration with *Kadabra*, our analysis tool, was straightforward and resulted in an automatic and more scalable testing environment.

Tool	**Repository**	**Last Commit Date**	**Referenced in**
Jadx	`https://github.com/skylot/jadx`	Jul 26, 2022	[41, 35]
Androguard	`https://github.com/androguard/androguard`	Jul 26, 2022	[19]
Cfr	`https://github.com/leibnitz27/cfr`	Jul 11, 2022	[35, 41, 37]
Apktool	`https://github.com/iBotPeaches/Apktool`	Jul 10, 2022	[3]
Fernflower	`https://github.com/fesh0r/fernflower`	Jul 4, 2022	[41, 37]
Dexpler (Soot)	`https://github.com/soot-oss/soot`	Jul 1, 2022	[7, 3]
Krakatau	`https://github.com/Storyyeller/Krakatau`	Jun 16, 2022	[37]
dex2jar	`https://github.com/pxb1988/dex2jar`	Nov 3, 2021	[3]
Procyon	`https://github.com/mstrobel/procyon/wiki/Java-Decompiler`	May 27, 2021	[41, 37]
JD Project	`https://github.com/java-decompiler/jd-core`	Feb 26, 2020	[35]

Table 3.1: Disassemblers and Decompilers for Java and Android

3.4 Summary

The number of refactoring tools we found is higher than detection-only tools. It might indicate more research being done with developer assistance in mind rather than other stakeholders. From the list of patterns compiled, we filtered for code patterns and mapped which energy patterns are treated by each refactoring tool. Figure 3.1 visually presents the landscape of pattern coverage, and from our research, while we did not test all of the energy tools, we assume that every pattern has at least one tool capable of refactoring it.

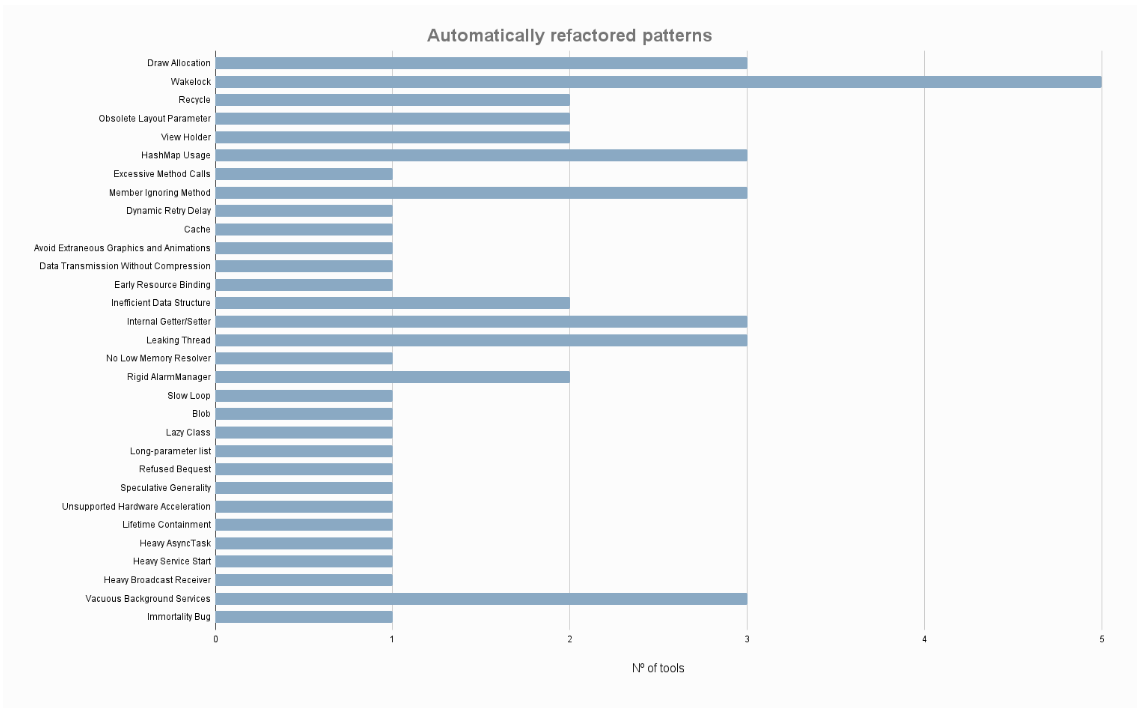


Figure 3.1: Automatically refactored code patterns

Another important point relevant for future research is the apparent fragmentation of energy analysis tools. It seems developers wanting to analyse their applications for energy efficiency require multiple tools that are spread across multiple IDEs. Based on our research, no tool provides total coverage of the existing energy patterns catalogues.

As for the type of analysis being done, most tools depend on source code availability, therefore only satisfying developer needs. The tools we found that analyse *APK*s directly perform bytecode analysis. However, directly related to our work, no research was found on the feasibility of analysing decompiled Java code.

In conclusion, we prepared a good knowledge foundation of energy pattern catalogues $^{3.1}$ and energy tools $^{3.2}$ from which to base our work. Since the tools we found are mostly tied to IDEs and cannot analyse Java code with an *incomplete classpath*, we must develop our custom analysis tool. As explained in Section 2.3, we chose *Kadabra* for its rich feature set that enabled us to perform a fair comparison of energy code patterns between source code and *APK*s. To extract and translate the bytecode of *APK*s into Java source code, from the listed tools in Section 3.3, we selected *Jadx* as our decompilation tool since it is referenced as the best performing decompiler in the literature [41, 35]. In the next chapter, we go over the requirements and overall architecture of our energy analysis tool, and we also present detailed explanations for the four patterns we selected and implementation details of their respective detectors.

Chapter 4

E-APK: Energy Pattern Detection in Decompiled Android Applications

This chapter will cover the technical aspects of our work. As we mentioned in Section 2.3, to properly explore and test our RQ, we developed a custom solution based on *Kadabra*.

Our goal with this solution is to analyse Java source code, regardless of the code origin, i.e., source code from a project or decompiled code from an *APK*; this approach facilitates our concrete objective of assessing the viability of analysing decompiled Java code and also promotes our broader objective of expanding the available tools for independent energy analysis.

The following sections detail the requirements of our solution and the overall architecture and give an in-depth look into the four energy code patterns we selected.

4.1 Requirements

The following requirements, functional and non-functional, define the main features and intents of our solution.

Functional requirements:

- Detect energy code patterns in Java source files or *APK*s.
- Produce a report with traceability data for each pattern after analysis.
- No user intervention required besides inputting the source files or *APK*.

Non-Functional requirements:

- Detector implementations must be compatible with both source code and decompiled code.
- Execution time per application should be as low as possible for an easier integration within an *APK* validation pipeline.
- Easily extensible to support more patterns.

4.2 Architecture

The two primary components of our solution are Kadabra [13, 46] and Jadx [35, 41] and in Figure 4.1 we can see an architecture diagram, where the three different colours represent the main development efforts of our solution.

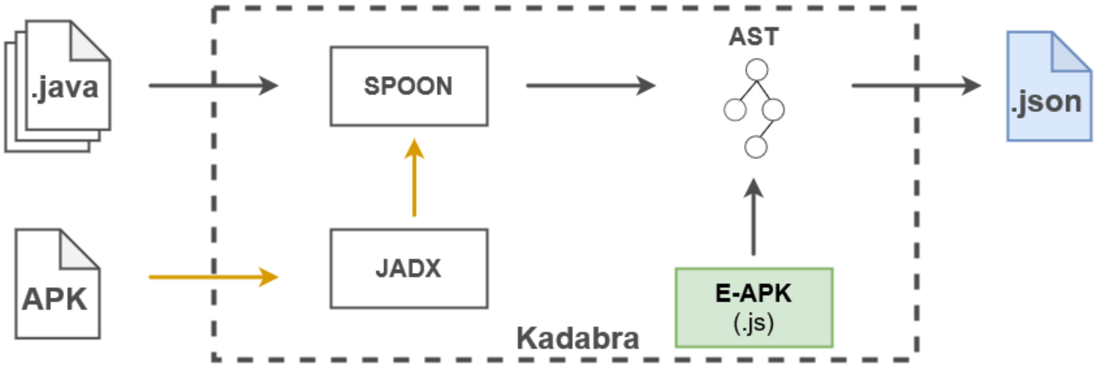


Figure 4.1: Architecture Diagram

We wanted to generalise the analysis process by internally decompiling *APK*s with Jadx and running the resulting Java source files through our detectors; this is represented by the yellow coloured arrows in Figure 4.1. Kadabra uses Spoon internally to parse and build an AST, and with our integration of Jadx into Kadabra, we can now take *APK*s as input files that are transparently decompiled and forwarded to Spoon for analysis.

For our analysis, we developed a set of detectors that exist as a collection of Kadabra scripts written in javascript that search the AST for instances of energy patterns; it is represented by the green box labelled E-APK, or Energy-aware Android Patterns for Kadabra.

The output of our analysis is structured in a JSON file, represented by the blue box. It contains the location of each detected pattern and will be presented in more detail in the next section.

4.3 Implementation

The development of our solution was divided into two phases:

Decompilation of Android applications - This phase pertains to the integration of Jadx into Kadabra to transparently decompile *APK*s (yellow arrows in Figure 4.1).

Analysis of Android applications - This phase encompasses the development of Kadabra scripts to detect energy code patterns in Java source code and the production of a report with the detected patterns (green and blue boxes in Figure 4.1)

To properly explore the impact and viability of performing static analysis on decompiled code, selecting a subset of code patterns and a large enough dataset to test everything on was essential.

Couto et al. [16] studied the impact of refactoring certain energy-greedy code patterns on energy consumption. In their work, they defined a pattern as being an Energy-Greedy Android Pattern (EGAP) and compiled a set of 11 EGAPs.

From that set of EGAPs, we selected the top three most frequently detected patterns in their dataset: Member Ignoring Method or **MIM**; Excessive Method Calls or **EMC**; HashMap Usage or **HMU**. Together with these three patterns we included a fourth one, Internal Getter or **IG**.

Selected Energy Patterns
HashMap Usage [16, 42, 30]
Excessive Method Calls [16]
Member Ignoring Method [16, 47, 11, 30]
Internal Getter [47, 11, 42]

Table 4.1: Energy Patterns

In the following subsections, we will explain in detail the decompilation and analysis processes, how each detector works and how the final report is structured.

4.3.1 Jadx integration

We started by integrating Jadx into Kadabra, and the process was reasonably straightforward since Jadx is available as a library and provides clear documentation and examples. However, we had to address a significant side-effect of the decompilation process.

When decompiling an *APK*, code from 3rd-party libraries like Google and Android APIs are also present, so filtering is essential to extract only the actual application code for analysis. For reference, in a test application built with Android Studio, unfiltered decompilation resulted in 3066 classes; after applying a filter, that number was reduced to the 14 relevant classes.

Jadx already has a feature for filtering classes, but we encountered an issue where, despite filtering for a particular class, if that class had any dependencies, i.e., imported another class, it would also be extracted; this was an obvious problem since many classes in an application have dependencies from the Android API.

We reported this problem to Jadx's developers and submitted a pull request that was successfully merged [1]. With those changes, we were able to successfully use the application package name to filter for the relevant application code without any extra unwanted code.

We wanted to give the user the ability to specify any class filter they wanted. For example, in an application called "com.my.application", we could use its name as a filter to exclude code from "com.google" and others, but it could also be useful to only exclude code from "com.google" and extract everything else.

To provide a more flexible filtering system within Kadabra, we implemented an optional execution parameter, "-APF (Android Package Filter)", where the user can pass any number of matching

[1] `https://github.com/skylot/jadx/pull/1467` (Last Access: Sep 08, 2022)

filters to easily extract only the required classes. The filters can be written with a simple mini-DSL with the following rules:

- "com.my.application" - match exactly "com.my.application"
- "?com.my" - match any classes beginning with "com.my"
- "foo?" - match any classes ending with "foo"
- "?bar?" - match any classes containing "bar"
- "!?google?" - exclude any classes containing "google" ("!" can negate any of the rules above)

We mentioned the application package name several times as a way to filter an application, but this implies the user must know the package name. We also provided a special filtering option for situations where that information is not readily available. The package name can be acquired from the encrypted AndroidManifest.xml inside the *APK*, and Jadx can decrypt it. Using the "-APF" parameter mentioned before with the filter "package!" will make Jadx retrieve the package name automatically and use it as a filter.

4.3.2 Detector structure and logic

With the decompilation process fully integrated, we moved to our detector implementations. All detectors follow a base structure and logic to fulfil our non-functional requirement of extensibility. Every detector extends a base detector class with a custom *analyseClass* method with the specific detector logic and a *save* method to extract relevant information from the analysis results. The base detector has a generic *analyse* method to iterate through all classes. Listing 4.1 provides a simplified structure of our detectors.

```
 1 class Detector {
 2  analyse() {
 3   Query.search("class").get().forEach(
 4     (c) => this.analyseClass(c)
 5   );
 6  }
 7 }
 8
 9 class ExampleDetector extends Detector {
10   analyseClass(jpClass) {...}
11   save() {...}
12 }
```

Listing 4.1: Simplified Detector Structure (JS code)

Kadabra provides extensive techniques and mechanisms to analyse code, particularly its query-like language to search and analyse ASTs. In lines 3-5, we can see the query system and its method chaining syntax that drive our detection logic.

To give a more detailed view of the query functionality, examine the following example written in natural language:

Get all void methods inside classes named "foo"

We can translate this into a search query that Kadabra can process like this:

Query.search("class", {name: "foo"}).children("method", {returnType: "void"}).get()

where the query starts by looking for classes with the name "foo", then searches for children of type "method" with a return type of "void". Every detector takes advantage of this query system to model its search criteria as we will show in the next four subsections.

4.3.3 Member Ignoring Method

This pattern occurs when a non-static method in a class does not interact with any instanced field or method and is not an override or overridden. As stated by Couto et al. [16], static methods are allocated in a different memory block, away from objects, and only a single instance of the method is created regardless of the number of class instances; the reduction in energy consumption comes from this process.

```java
public class Foo {
  public int sum(int a, int b) {
    return a + b;
  }
}
```

Listing 4.2: Example of Member Ignoring Method (MIM)

Listing 4.2 provides an example of the pattern as it appears in Java source code. Our search criteria must consider several attributes. We want to find methods that are neither static, final, nor involved with inheritance. They must also not use any non-static fields or methods.

We start by filtering out methods that are already static, that are final or contain "@Override" annotations as seen on Listing 4.3. After this first filtering phase, we check if the remaining methods use only parameter variables (if any) and static fields and that any method calls performed inside them are also to static methods. The final verification is for inheritance, where we check if the method is an override or overridden by another method. This check requires comparing against all other methods and is an expensive operation, so we precompute and cache a map of all methods with the same name to improve performance.

```
let mightBeStatic = Query.childrenFrom(jpClass, "method", {
  isStatic: false,
  isFinal: false,
  annotations: noOverrideAnnotationFilter,
}).get();
```

Listing 4.3: Member Ignoring Method Query Logic (JS code)

4.3.4 Excessive Method Calls

This pattern refers to when a method is needlessly called inside a loop and can be moved outside without changing the application's behaviour.

Couto et al. [16] rightly states that needlessly calling a method can have a negative impact on performance; calls typically require pushing arguments to a stack and writing to CPU registers, and the method body can also contain complex operations, therefore reducing the number of calls can, by proxy, result in reduced energy consumption. It can be seen as a form of Loop-invariant code motion [4].

```
public int sum(int a, int b) {
  return a + b;
}

public int uselessSum() {
  int sum = 0;
  for (int i = 0; i < 10; i++) {
    //Call can be moved outside the loop
    int res = sum(1, 1);
    sum += res;
  }
  return sum;
}
```

Listing 4.4: Example of Excessive Method Class (ECM)

Listing 4.4 provides an example of the pattern as it appears in Java source code. There are many ways to go about detecting this pattern; our implementation performs two searches in the AST: i) collect data on reads and writes of variables (local or fields) inside loops; ii) analyse method calls inside loops against the collected data. Calls that do not read or write loop variables are marked as invariants and can be safely removed from the loop.

As an example, in a method call like *A(X, B(), C(Y))*, by analysing the nodes from the inside out, i.e., starting with Y, then C() and ending with A(), if an inside node is declared variant, all outside ones will be as well; if *B()* is variant, *A()* is also variant.

We operate over *loop* nodes this time. These nodes are an abstraction in Kadabra to represent any type of loop, e.g., for or while loops. During the first pass over the AST, in line 4 in Listing 4.5, we build a structure with each method's variable reads and writes and a boolean flag to signal missing info, i.e., a call to a method not in the classpath. In the second pass, in lines 6-7 we analyse all method calls recursively to check if they either require any data modified in the loop or if they modify data used in the loop, directly or indirectly.

```
1  analyseLoop(jpLoop) {
2    this.resetDetector();
3
4    this.collectLoopInfo(jpLoop);
5
6    let calls = this.getFirstDescendentsOfTypes(jpLoop, ["call"]);
7    calls.forEach( (c) => this.analyseLoopCall(c) );
8  }
```

Listing 4.5: Excessive Method Calls Query Logic (JS code)

4.3.5 HashMap Usage

This pattern comes as an official Android recommendation where usage of HashMaps is discouraged in favour of ArrayMaps [2]. As a side note for our HMU implementation, while Couto et al.'s implementation [16] looks for all instances of HashMap or Map, our implementation only looks for HashMap instantiations.

We reason that HashMap instantiations are inherently made by the developer, and upon refactoring an instantiation to ArrayMap, a developer must also update any connected code. Furthermore, our implementation addresses a potential problem where the detector could flag an external HashMap source outside the developer's control, e.g., an API outputting a HashMap. We can also argue that using the generic Java Map interface is a good practice. For these reasons, we believe that looking for explicit instantiation cases is a better approach to this pattern.

```
public void foo() {
  HashMap<String, String> hashMap = new HashMap<String, String>();
  Map<String, String> map = new HashMap<String, String>();
}
```

Listing 4.6: Example of HashMap Usage (HMU)

Listing 4.6 provides an example of the pattern as it appears in Java source code. The search criteria is quite simple for this pattern. We look for instantiations of HashMaps from the package "java.util". Instantiations are syntactically defined by the *new* keyword, and in Kadabra's AST

[2]https://developer.android.com/reference/android/util/ArrayMap

new is a type of node. We search for all *new* nodes with a type of "HashMap" and filter them to only return those whose package is "java.util".

```javascript
let hashMapRefs = Query.searchFrom(jpClass, "new", { type: "HashMap" })
  .get()
  .filter(
    (jp) =>
      jp.typeReference !== undefined &&
      jp.typeReference.package === "java.util"
  );
```

Listing 4.7: HashMap Usage Query Logic (JS code)

4.3.6 Internal Getter

This pattern is unrelated to the previous three. It appears in one of the catalogues mentioned before [11] and is covered by a few tools [42, 33, 14]. This pattern is more relevant in older Android JIT compilers [3]. It presents itself as a simple code pattern and appears to be optimisable by the compiler, so it is an interesting pattern to study possible side-effects of the compilation and decompilation processes. As implied by the name, our implementation focuses only on *getters*, i.e., methods that purely return a field inside their own class.

```java
 1 int id = 123;
 2
 3 public int getID() {
 4   return id;
 5 }
 6
 7 public void foo() {
 8   //...
 9   bar = getID();
10   //...
11 }
```

Listing 4.8: Example of Internal Getter (IG)

Listing 4.8 provides an example of the pattern as it appears in Java source code. Our search criteria are fairly simple here. Looking at line 9, we want to find calls of internal getter methods inside the class where the method is defined. An internal getter is defined as a method containing a single field return statement.

Listing 4.9 shows the class analysis logic for *IG*. In lines 6-14, we collect all methods that meet our *Internal Getter* criteria. With *Query.childrenFrom(jpClass, <criteria>)* we select, inside the

[3] `https://stackoverflow.com/questions/4912695/what-optimizations-can-i-expect-from-dalvik-and-t` `4930538#4930538` (Last Access: Sep 05, 2022)

class we are currently analysing, all methods that are not void and not static. Then with consecutive *.children(...)* methods, we further refine the search with methods whose body only have a single return statement of a single field variable. In lines 16-20, we cross-reference the getter methods we found with the method calls inside the class and return any calls of internal getters.

```
 1 analyseClass(jpClass) {
 2   super.analyseClass(jpClass);
 3 
 4   const notVoid = (r) => r !== "void";
 5 
 6   let simpleGetters = Query
 7     .childrenFrom(jpClass, "method", {
 8       returnType: notVoid,
 9       isStatic: false,
10     })
11     .children("body", { numChildren: 1 })
12     .children("return")
13     .children("var", { isField: true })
14     .chain().map((m) => m["method"]);
15 
16   let internalCalls = Query
17     .searchFrom(jpClass, "call", {
18       decl: (d) => d !== undefined &&
19       simpleGetters.some((sg) => sg.same(d)),
20   }).get();
21 
22   this.results.push(...internalCalls);
23 }
```

Listing 4.9: Internal Getter Query Logic (JS code)

4.3.7 Detection report

Each detector outputs an array with the detections found, as shown in Listing 4.10. This array can be exported to a JSON file individually if the user only requires data from a particular detector.

```
[
  "Theraphy.java/Theraphy/prepareListData:27",
  "UserSession.java/UserSession/getUserDetails:44",
  "UserSession.java/UserSession/getUserDetails:50"
]
```

Listing 4.10: Example of HashMap Usage Detector Output

Here it is crucial to consider another challenge of decompilation: how to define and represent the pattern location in the report. Whether we want to compare results from source code against decompiled code or analyse an *APK* and return feedback to the developer, we need to trace the location of a detection as accurately as possible.

A naive approach would be to record the file and line associated with a code pattern; however, the compilation and decompilation processes can change the order of the code, remove whitespaces and comments and even introduce new debug comments. All these factors can change the line location of the patterns; therefore, comparing patterns based on line location is unreliable.

We attempt to solve this problem by representing each pattern detection with a string in the following format:

”*<fileName>/<className>/<methodName>*[*/<callName>*] : *<lineNumber>*”

Each part can be seen as a branching point in the AST, and the *<methodName>* (or *<callName>* where applicable) is the highest precision we can achieve for locating patterns. The *callName* information is optional, only applied to two detectors, *EMC* and *IG*. We also keep the line number at the end of the string since it is valuable information, but as stated before, it is unreliable for comparing patterns.

In Figure 4.2, we can see the creation process of the E-APK report. When analysing the source code of an application or its corresponding *APK*, our solution executes all existing detectors and compiles the individual reports from each detector into a single JSON report.

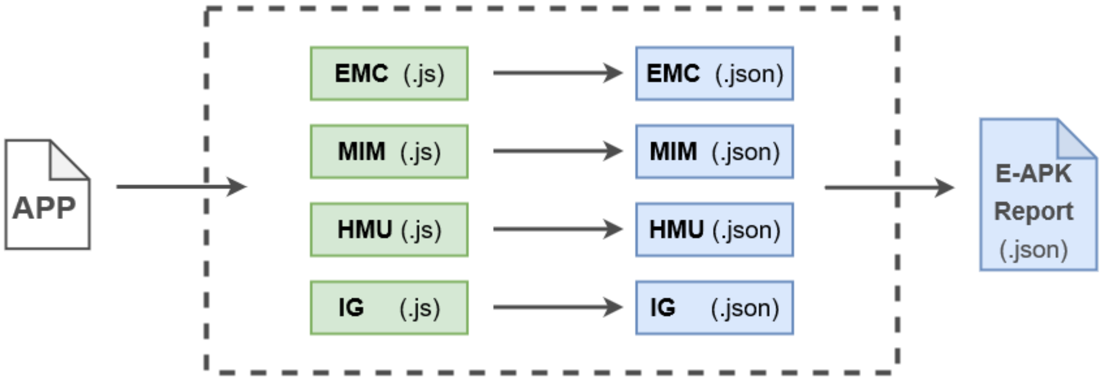


Figure 4.2: Creation of E-APK Report

The report, as exemplified in Listing 4.11, is in JSON format and contains the total number of patterns detected, details for each detector and the source of the analysis, i.e. the folder analysed or the *APK* file. In this particular example, we can see in line 3 the source pointing to a folder with the source code. We have a total of 3 pattern detections in line 5, and those detections are instances of HMU.

```
 1 {
 2   "sources": [
 3     "E:/apps/003f67dd-c926-4f96-bb1b-5a2dd4bf344f/latest/app/src/
         main/java"
 4   ],
 5   "total": 3,
 6   "detectors": {
 7     "Excessive Method Calls Detector": [],
 8     "HashMap Usage Detector": [
 9       "Theraphy.java/Theraphy/prepareListData:27",
10       "UserSession.java/UserSession/getUserDetails:44",
11       "UserSession.java/UserSession/getUserDetails:50"
12     ],
13     "Internal Getter Detector": [],
14     "Member Ignoring Method Detector": []
15   }
16 }
```

Listing 4.11: Example of an E-APK report

In the next chapter, we will explain how we used this solution to test a large dataset with 420 applications and how we compiled and analysed all the collected E-APK reports.

Chapter 5

Evaluation

Previous research, like the catalogues described in Section 3.1, has shown that refactoring energy code patterns can positively impact energy efficiency. Our objective is to understand if energy patterns, initially present in the source code, remain present and detectable after undergoing the *APK* compilation and decompilation processes. To achieve this, we take a sample of Android applications and test their source code and *APK*s with our detectors previously introduced in Chapter 4. We can then compare the results obtained across the multiple versions.

In this chapter, we explain the dataset of open source Android applications we used and how we prepared it; we present the methodology and reasoning used to collect the report data. Finally, we show how the data analysis was done and how the results presented in Chapter 6 were obtained.

5.1 Open Source Android Applications Dataset

To properly answer our RQ, a significantly sized dataset is required. Couto et al.[16] provided a dataset with roughly 600 applications originally used to study a set of energy patterns they call EGAPs. As explained in Section 4.3, three of our detectors are from this EGAP group, and the dataset contains over 100 instances of these same patterns; these are relevant characteristics that make it suitable to be re-used in our context.

In order to fairly compare source code against *APK*s, we had to impose a few selection criteria and perform some data preparation, and indeed, we found some nuances in the way the dataset was organised. The projects were all inside individual folders named with a UUID, but some folders contained multiple subprojects inside, e.g., the actual application and sometimes a sample project. Some projects also had multiple *APK*s available, e.g., distribution, production or development *APK*s.

To normalise the testing conditions, we took advantage of some standard Android project practices, like the existence of *src* and *build* folders by default. We recursively analysed all folders in the dataset and applied the following criteria in this specific order:

1. Select all *src* folders (even multiple per project).

   From the original 590 base application folders in the dataset, we found 1015 *src* folders.

2. Select all projects with a *build* folder containing *APK*s.

   From the 1015 candidate projects, only 423 had *APK*s, and all of those contained three versions of *APK*s: Debug is an *APK* built, zipaligned [1] and signed with debug keys and with the *debuggable* flag enabled; Debug(Unaligned) is similar to Debug without the zipalign process; Release is also similar to Debug but has the *debuggable* flag disabled.

3. Select all projects using Gradle.

   Most of the remaining projects used Gradle, with only two being removed in this step, reducing to suitable projects to 421.

4. Select all projects with an AndroidManifest.xml.

   From the 421 projects, only one was missing an AndroidManifest.xml file.

Ultimately, we ended up with 420 applications that comprise our study object. A final preparation step was performed where we extracted the application package name from the AndroidManifest.xml and the boolean values of the build property *minifyEnabled* for release and debug builds from the *build.gradle*. We compiled all the information into a JSON file, represented in Listing 5.1, with the 420 test applications, 14 of which have *minifyEnabled* Release *APK*s.

```json
{
"0": {
        "base": "003f67dd-c926-4f96-bb1b-5a2dd4bf344f",
        "applicationId": "com.githang.androidcrash.app",
        "dminified": false,
        "rminified": false,
        "src": "E:/apps/003f67dd-c926-4f96-bb1b-5a2dd4bf344f/latest/
            app/src/main/java",
        "apks": [
            "E:/apps/003f67dd-c926-4f96-bb1b-5a2dd4bf344f/latest/app
                /build/outputs/apk/app-debug-unaligned.apk",
            "E:/apps/003f67dd-c926-4f96-bb1b-5a2dd4bf344f/latest/app
                /build/outputs/apk/app-debug.apk",
            "E:/apps/003f67dd-c926-4f96-bb1b-5a2dd4bf344f/latest/app
                /build/outputs/apk/app-release-unsigned.apk"
        ]
    },
"1": {...}
}
```

Listing 5.1: Example of a test application from the processed dataset

[1] https://developer.android.com/studio/command-line/zipalign (Last Access: Aug 9, 2022)

5.2 Testing Methodology

In this section, we will explain the thought process during the data collection phase of our solution. We will also explain how we analysed that data and what metrics we used to compare the similarity of detections between source code and *APK*s.

5.2.1 Data Collection

Our end goal is to compare source code and decompiled code with respect to our ability to detect energy code patterns in them. To do this, we test each application four times, once for the source code and the other three times for each *APK* version: Release; Debug; Debug(Unaligned). In Figure 5.1 we can see how the four versions are organised in our experimental setup.

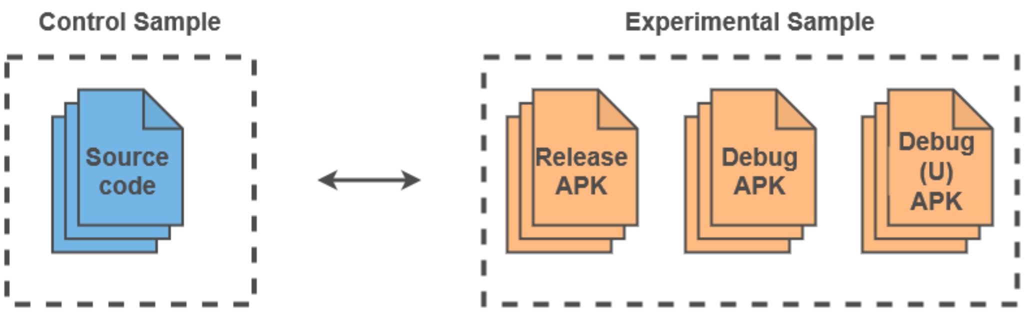


Figure 5.1: Control and Experimental Samples

The source code can be seen as our control sample or ground truth since no code information is missing, and any energy code pattern present must be detectable, assuming the detectors are working correctly. The release *APK* is the most likely candidate version that App Stores and consumers will have access to and is, therefore, the one we will draw most of our comparisons with.

The analyses of Debug and Debug(Unaligned) *APK*s were not a specific target but instead presented themselves as an opportunity; all of our 420 applications had those available, and their analyses could nonetheless yield interesting results.

We mentioned before that we extracted the application package name from the AndroidManifest.xml when preparing for analysis; we use the package name to filter out any code that is not from the developer. As explained in Section 4.3.1, with Jadx class filtering options and the simple filtering system we implemented in Kadabra, we try to ensure, as best as possible, a fair comparison between source code and the *APK*s, i.e., any decompiled code we extract also exists in source code.

We also took note of applications using the build property *minifyEnabled*, responsible for obfuscating code as explained in Section 2.2. We did it not only for analysis purposes but also to know when the deobfuscation options of Jadx were required. However, after testing, we found that keeping those options enabled at all times worked without any perceivable side effects.

In Section 4.3.7, we explained the structure of the E-APK report our solution outputs. For a single applicaiton, we generate four E-APK reports for each of the four versions. Listing 5.2 is an example of this, where we can see an ID, the values for the *minifyEnabled* Gradle build properties for release and debug builds, and the four E-APK reports.

```
{
  "id": "0",
  "minified_r": false,
  "minified_d": false,
  "result": [
    {<source report>},
    {<debug APK report>},
    {<debug(unaligned) report>},
    {<release report>}
  ]
}
```

Listing 5.2: Example of JSON with analysis results of an application

Initially, we ran our detectors for each of the four versions for the 420 applications. While the execution time per application was acceptable, ranging from 10 seconds to over a minute in some cases, cumulatively, it took roughly three hours. We were executing Kadabra using Python, writing the results to a JSON file on disk, reading it and repeating the process for the next application.

To accelerate the analysis time and facilitate a more iterative analysis, we took advantage of Kadabra's *runParallel* feature to analyse each version in parallel. The total execution was roughly reduced by a factor of four to less than an hour. The end product was a final JSON file with the analysis results of all 420 applications, and we used it to perform the analysis detailed in the next section.

5.2.2 Data Analysis

We performed our analysis in two phases:

1. Analysis over the total number of detections

   This initial, more general analysis, looks at the total number of detections between source code and *APK*. We also briefly compare the obfuscated with the non-obfuscated applications, despite the small sample size of obfuscated applications in our dataset.

2. Analysis of similarity between detections

   The second analysis, more in-depth, verifies the actual similarity between detections in each version using different metrics to compare detection instances in their string form.

Our initial analysis takes the total number of detections as the comparison point. For a given application, for a given detector, we consider a perfect match between source code and *APK* when both versions have the same number of detected patterns, e.g., detector EMC on application X reported three instances in source code and three instances in *APK*.

An obvious flaw of this analysis is the lack of differentiation between instances, e.g., the three EMC detections on application X can be in *class foo* in source code and *class bar* in *APK*. In total, both versions have three detections, but they are not the same. We address this flaw with our second analysis.

The takeaway from analysing total detections, however, is still quite relevant. It tries not to prove our RQ but instead tries to disprove it. If there is a significant mismatch between versions, we know our hypothesis, or our approach at the very least, is not viable.

For the second analysis, we focus on comparing the results of each detector individually. We analyse and compare the results using two different metrics: Jaccard Index [34, 24] and Fuzzy Matching [31, 15].

Both metrics serve the same purpose: calculating a similarity score between two versions. Each detector gives a normalised similarity score ranging from zero to one. To represent the similarity per application, we use a cumulative score calculated with the sum of all four detectors where the maximum value is four, i.e., all detectors got a perfect score of one.

```
[
    "Theraphy.java/Theraphy/prepareListData:27",
    "UserSession.java/UserSession/getUserDetails:44",
    "UserSession.java/UserSession/getUserDetails:50"
]
```

Listing 5.3: Example of HMU detection before transformation

Before calculating these metrics, we must again address the line location problem explained previously in Section 5.2.1. If we encounter two patterns in the same method, the only distinguishing factor is the line number; this is an unavoidable limitation of our solution due to our approach to traceability. Listing 5.3 gives an example using the HMU detections shown before. In particular the method *getUserDetails* appears twice in lines 44 and 50.

Our solution here was to transform the string representation of the patterns. The transformation consists of removing the line number from each detection, counting the number of duplicates, appending that number to the end of each string, and finally, removing the duplicates.

```
[
    "Theraphy.java/Theraphy/prepareListData:1",
    "UserSession.java/UserSession/getUserDetails:2",
]
```

Listing 5.4: Example of HMU detection after transformation

Listing 5.4 is the transformed version of the previous example. The two instances of method *getUserDetails* get transformed to *...getUserDetails:2*. With the transformation of the results from each detector, we get sets of unique pattern detections, i.e., lists of unique strings without duplicates, ready to be compared using the different metrics.

5.2.3 Metric - Jaccard Index

The Jaccard Index [34, 24] is a statistic to compute the similarity between two sets. The similarity is represented by the ratio between the cardinality of the intersection over the cardinality of the union. It follows the following formula:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}, \tag{5.1}$$

where A and B are the two sets we want to compare. For our purposes, if both sets are empty, we consider them fully similar, with a Jaccard Index of 1.

To give a more tangible example of how the Jaccard Index works, suppose we have the two following sets:

$$A = [foo:1, bar:2]$$

$$B = [bar:2, baz:1]$$

The intersection of the two sets is the set $[bar:2]$, whose cardinality is one. The union is the set $[foo:1, bar:2, baz:1]$ with a cardinality of three. By applying equation 5.1 for Jaccard Index, we get:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{1}{3} \tag{5.2}$$

This result means that set A is 33.(3)% similar to set B. In reality, the result should be 50%, since we have two instances of *bar* and thus the total number of detections represented here is four.

We account for this by recalculating the cardinality based on the number of instances. The set $[bar:2]$ with cardinality of one gets a cardinality of two instead, since *bar:2* represents two instances. By doing this, the union also gets a cardinality of four, and the final score is the expected 50%.

The weak point of this metric comes from some decompilation situations where the decompiled code suffered small changes, e.g. *AIMSICD* becomes *ActivityC0001AIMSICD*, and the *Jaccard* treats these as completely different instances. These differences are acceptable overall, as we will see in Section 6, and make the *Jaccard* a more conservative metric that gives a worst-case scenario score.

5.2.4 Metric - Fuzzy Matching

With Fuzzy Matching [31, 15] we get a more optimistic score, which is an excellent complement to *Jaccard* and can more correctly score the previously stated decompilation problems. We use a library for Python called *TheFuzz*[2] to calculate the string differences. Given the higher degree of complexity involved, we will forego a detailed explanation of how it works and will instead apply it to the previous example. We tried three variations with Fuzzy Matching, using two methods from the *TheFuzz* library.

With the *fuzz.token_sort_ratio* method we first transform the set into a single string concatenated with a Pipe (|) symbol, e.g., $[foo:1, bar:2]$ becomes $"foo:1|bar:2"$. We call this method *FuzzyConcat*. When comparing the previous sets A and B, we get a score of 73%. In this example, we go above the expected 50% because of the similarities between *bar* and *baz*. Moreover, since we compare the whole strings, the repeated symbols and numbers *(:, 1, 2)* also increase the score. The benefit of this metric is more evident in the decompilation example we showed before. If we apply it to the sets $[AIMSICD:1]$ and $[ActivityC0001AIMSICD:1]$, we get a score of 0% with *Jaccard* and a more representative score of 58% with *FuzzyConcat*.

We also test the *process.extractOne* method from the *TheFuzz* library that, given a string, finds the best match in an array of strings. We iterate the smallest set, apply this method against the biggest set, sum each score and divide the final result by the length of the biggest array. We call this one *FuzzyProcess*. Applying it to these different sets, $["foo:1", 'bar:2', "AIMSICD:1"]$ and $["foo:1", "bar:2", "ActivityC0001AIMSICD:1"]$, we get a score of 97% which is quite close to the expected score of 100% (remember that AIMSICD is a decompilation edge case). The previous *FuzzyConcat* gets a score of 76% and *Jaccard* gets 60%.

The final metric we use is a combination of *Jaccard* and *FuzzyConcat*, which we call *FuzzyJaccard*. The Jaccard Index has many variations, including some with Fuzzy sets [45]. Our implementation is much simpler and aims to hopefully provide a score that averages its constituents; in

[2] https://github.com/seatgeek/thefuzz (Last Access: Aug 8, 2022)

situations where reports should be 100% similar but are not due to obfuscation, we can get a more correct score. It is calculated using the following formula:

$$FJ(A,B) = J(A,B) + (1 - J(A,B)) * F((A - B), (B - A)), \tag{5.3}$$

where J is the *Jaccard* method and F is the *FuzzyConcat* method. It also follows from this formula that, should the difference between sets be an empty set, we get a perfect score of one from *Jaccard*. On the other hand, if the intersection is empty, the resulting score is entirely commanded by *FuzzyConcat*. Using the previous example, we start by calculating the Jaccard score, which was 60%. We then take the differences between sets, i.e. $[''AIMSICD : 1'']$ and $[''ActivityC0001AIMSICD : 1'']$ and calculate the *FuzzyConcat* score which is 58%. We take the complement of *Jaccard*, 40%, multiply it by the *FuzzyConcat* score to get 23.2% and add the *Jaccard* score to a final score of 83.2%.

5.3 Summary

We need to test our detectors with a significantly sized dataset. We take the dataset used in [16] and prepare it for our testing purposes, resulting in 420 applications, each with source code and three *APK*s: Release, Debug, Debug(Unaligned).

We split our analysis into two phases: analysing the total number of detections and comparing the actual similarity between detections. When comparing the similarity between versions, we use two different methods, Jaccard Index and Fuzzy Matching, for a total of four metrics. We compare how these four metrics perform with the dataset.

In the next chapter, we present all the findings obtained from the testing and analysis explained in this chapter.

Chapter 6

Results and Discussion

We collected data from 420 applications, each with four versions: Source code, Release APK, Debug APK, Debug(U) APK. We analysed each version with our four implemented detectors (EMC, HMU, IG, MIM). After analysing the 420 applications, we organised all the data as a dataframe from the well-known Python library, Pandas.

Our main objective is to understand how the detection of patterns differs between Source code and APK, so we first filtered our results for applications where we detected at least one energy pattern with the following query:

$$total_s > 0 \text{ OR } total_d > 0 \text{ OR } total_{du} > 0 \text{ OR } total_r > 0,$$

where $\mathrm{total}_\mathrm{s}$, $\mathrm{total}_\mathrm{d}$, $\mathrm{total}_\mathrm{du}$ and $\mathrm{total}_\mathrm{r}$ are the total number of detections, per application, in the source code, debug, debug(unaligned) and release versions respectively.

We found that only 152 applications, 36% of the 420, reported at least one detection. This result is consistent with the study of Couto et al. [16] where we sourced the dataset; they reported only detecting patterns in roughly 40% of the applications.

As we explained in Chapter 5, we divided our analysis into two phases: i) comparing the total number of detections; ii) comparing the similarity between detections. The following sections contain the findings from the two analyses we conducted using these 152 applications.

6.1 Analysis over the total number of detections

We start our analysis by counting the number of detections that can be found in the four different versions of each of the 152 applications we considered. The results we obtained can be found in Figure 6.1.

We detected 947 patterns in the source code, an 11.50% and 10.44% difference to the *Release* and *Debug* respectively. The number of detections was precisely the same, 853, between the *Debug* and *Debug(Unaligned)* versions.

The *Release* version had a similar the same number of detections, 844, which is 9 fewer detections, across three applications. We manually inspected all three applications and found the following:

- All three have the *minifyEnabled* property and showed signs of obfuscation.
- One of the applications had seven detections of HMU in *Source code* and *Debug* but zero in *Release* and, in fact, there was no presence of "HashMap" when performing a text search through the decompiled code.
- Another application was missing one out of three MIM detections in *Release*.
- The last application reported one extra EMC detection in the *Debug* version that was not present in *Source code*, while the *Release* was correct.

Overall, these differences could be attributed to obfuscation, but no real conclusion could be drawn from this information.

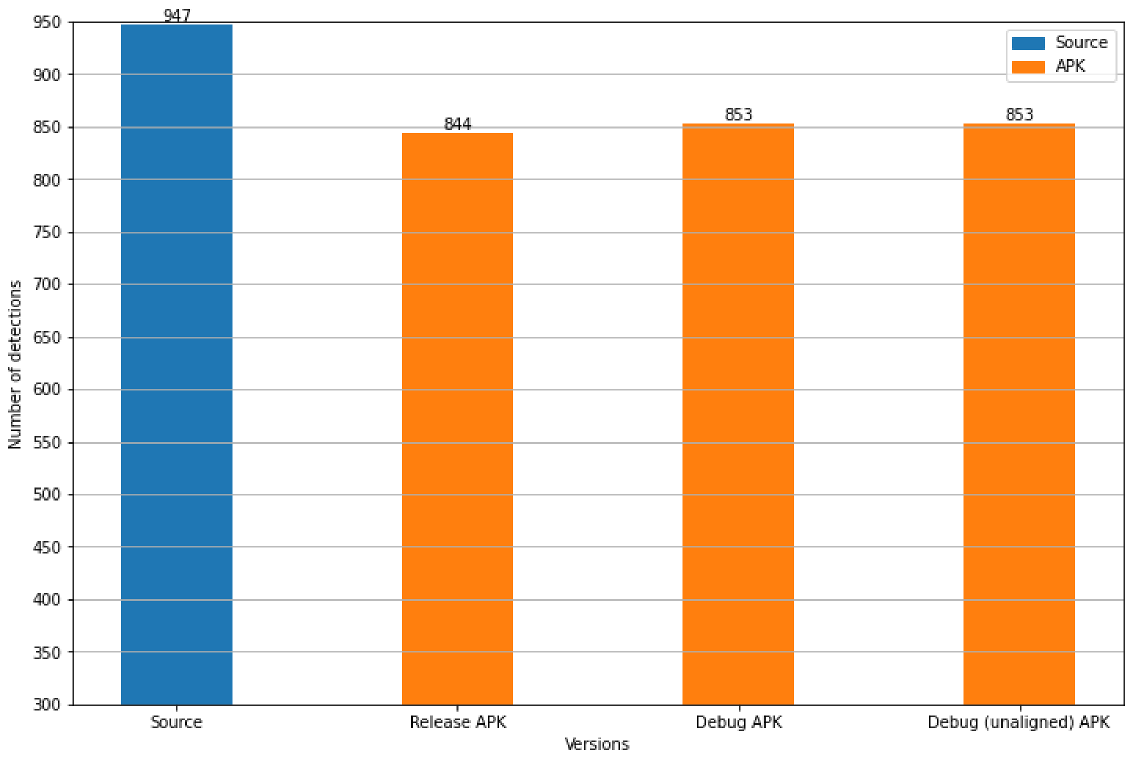


Figure 6.1: Total number of detections per version

We then tried to look further into the obfuscated applications. Of the 152 applications, only 8 were obfuscated, and 144 were non-obfuscated.

Table 6.1 contains descriptive statistics separating obfuscated and non-obfuscated applications. We show, for each version, the average number of detections, the standard deviation, the maximum number of detections in a single application and several quantiles.

	Mean	SD	25%	50%	75%	95%	max
				Non-Obfuscated			
Source	6.18	18.36	1.00	2.00	4.00	24.85	182
Release	5.58	18.16	0.00	1.00	3.00	23.95	182
Debug	5.58	18.16	0.00	1.00	3.00	23.95	182
Debug(U)	5.58	18.16	0.00	1.00	3.00	23.95	182
				Obfuscated			
Source	7.12	9.17	1.75	5.00	7.00	21.30	29
Release	5.00	9.49	0.00	1.50	4.25	19.95	28
Debug	6.12	9.60	0.75	3.50	5.50	21.30	29
Debug(U)	6.12	9.60	0.75	3.50	5.50	21.30	29

Table 6.1: Descriptive statistics for total number of detections in Obfuscated vs Non-Obfuscated applications

The standard deviation is relatively high in all situations, given the significant difference between the minimum and maximum values. The maximum number of detections in the obfuscated group is below the 95% quantile range of the non-obfuscated group, and no abnormalities were visible in the detectors when comparing the two groups.

In Figure 6.2 we can also see the same data points in a boxplot. The non-obfuscated group shows several outliers, but only one application is over the 100 detections range. The mean value in the boxplot does not include the outliers, therefore we can see a different between groups here.

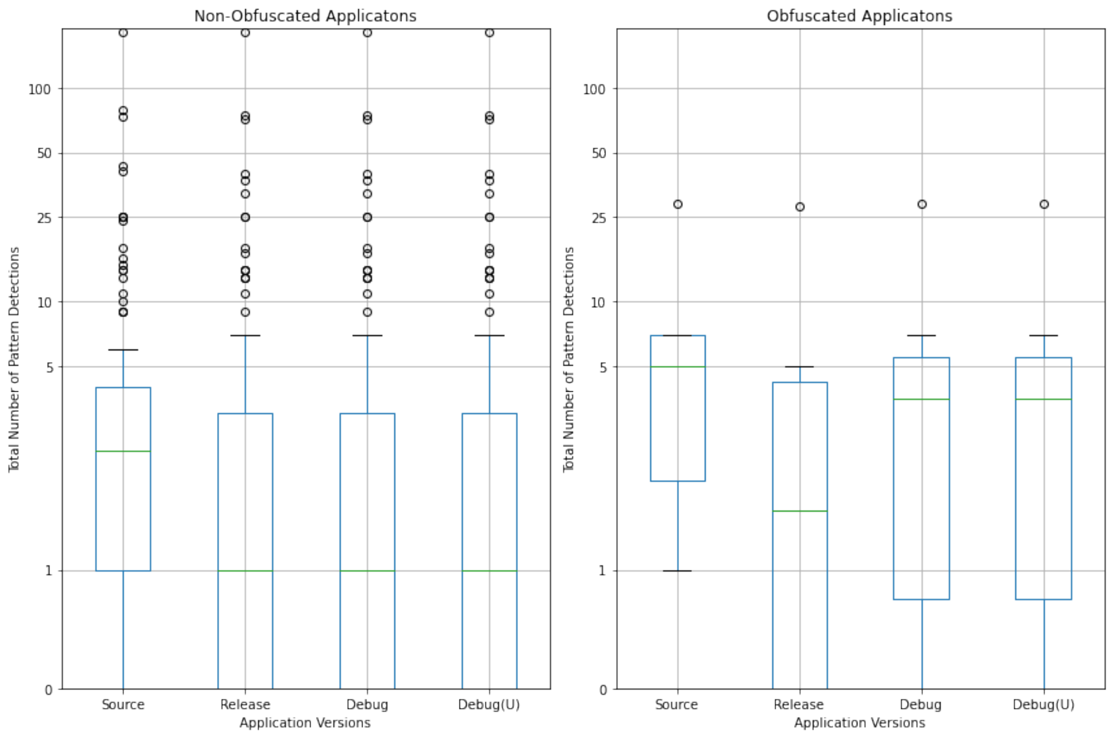


Figure 6.2: Number of detections by version

Overall, nothing conclusive on the impact of obfuscation was found. It would be interesting to test a bigger sample of obfuscated applications and compare the results to understand how obfuscation might impact our analysis.

An interesting finding related to obfuscation is the generation of invalid Java code in some circumstances. Prior to enabling the deobfuscation options in the decompiler Jadx, the analysis of five applications failed due to Spoon [44] (used internally by Kadabra) being unable to build their AST; this is another challenge of decompilation where we can have Dalvik bytecode that is incompatible with Java code.

As is the case with non-obfuscated applications, bytecode generated directly from Java can be translated back safely. However, should any transformation happen to the bytecode, e.g., obfuscation, some form of preprocessing might be required to decompile it safely. Notwithstanding, after enabling the deobfuscation options, all five applications that failed previously were successfully analysed.

We proceeded with our comparison using only the source code and the *Release APK* since it is the primary *APK* candidate as explained in Section 5.2.1. If we ignore the obfuscated applications, no difference was found between the *APK* versions. We also kept the obfuscated applications in the remaining analyses but disregarded any possible impact they may have due to their small sample size.

In Figure 6.3 we can see how each detector performed comparatively in the source code and the release *APK* and in Table 6.2 we show the difference in percentage between the two scenarios, calculated by dividing the absolute difference of both values by their average.

	EMC	HMU	MIM	IG
Source	194	170	151	432
Release APK	190	148	106	400
% Difference	2.08%	13.84%	35.02%	7.69%

Table 6.2: Percentage Difference between Source Code and Release APK

Without exception, the number of detections was higher in the source code, with the most significant difference coming from the *MIM* pattern at 35.02%. Judging only by this value, the *MIM* pattern could be the one suffering the most from the compilation and decompilation processes. Meanwhile, the *IG* pattern has visibly more detections than the rest.

As discussed before in Section 4.3.6, we considered the *IG* pattern to be a likely candidate for compiler optimisations, e.g., inlining. However, the results show the opposite; they reinforce the notion that perhaps this pattern is handled by the JIT or AOT compiler during the installation phase in the smartphone.

Our next step was to look at the differences across applications to get a more in-depth understanding of the results. Figure 6.4 is a histogram of the differences in total detections found between Source code and the Release *APK*. From the 152 applications, we found the same number of detections between the Source code and the Release *APK* in 64 applications. Besides those, 67

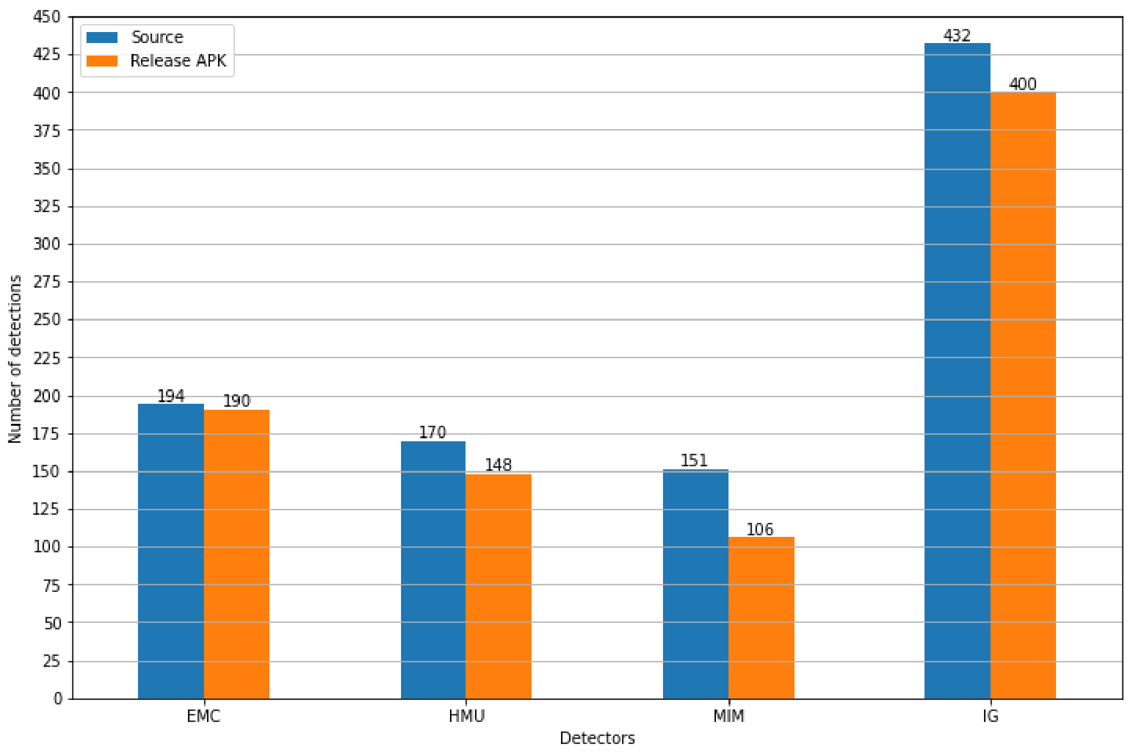


Figure 6.3: Total number of detections per detector

showed more detections in the Source code, with 47 of those having only one more detection, and 21 showed more in the Release *APK*.

With this information, it is important to recall the initial set of 420 applications. If we redo the previous comparison with the set of 420, we find that 332 applications have the same number of detections. In Table 6.3 we can see the different ratios for: same number of detections; Source having more than *APK*; Source having less than *APK*.

In the set of 420 applications, given the significant percentage of 79.05% where we did not find any differences between Source and *APK*, we can claim with some certainty that the compilation and decompilation processes are not systematically introducing detections; this, in turn, reinforces the potential of the type of static analysis we performed.

	Set of 152	Set of 420
Source = APK	42.11%	79.05%
Source > APK	44.08%	15.95%
Source < APK	13.82%	5.00%

Table 6.3: Ratios of equal, higher and lower number of detections in Source vs APK

To get a different perspective on the data we collected and understand how each detector behaves in situations where the number of detections does not match, we isolated the two groups, *Source* > *APK* and *Source* < *APK* and analysed them. Figure 6.5 shows the two groups with each

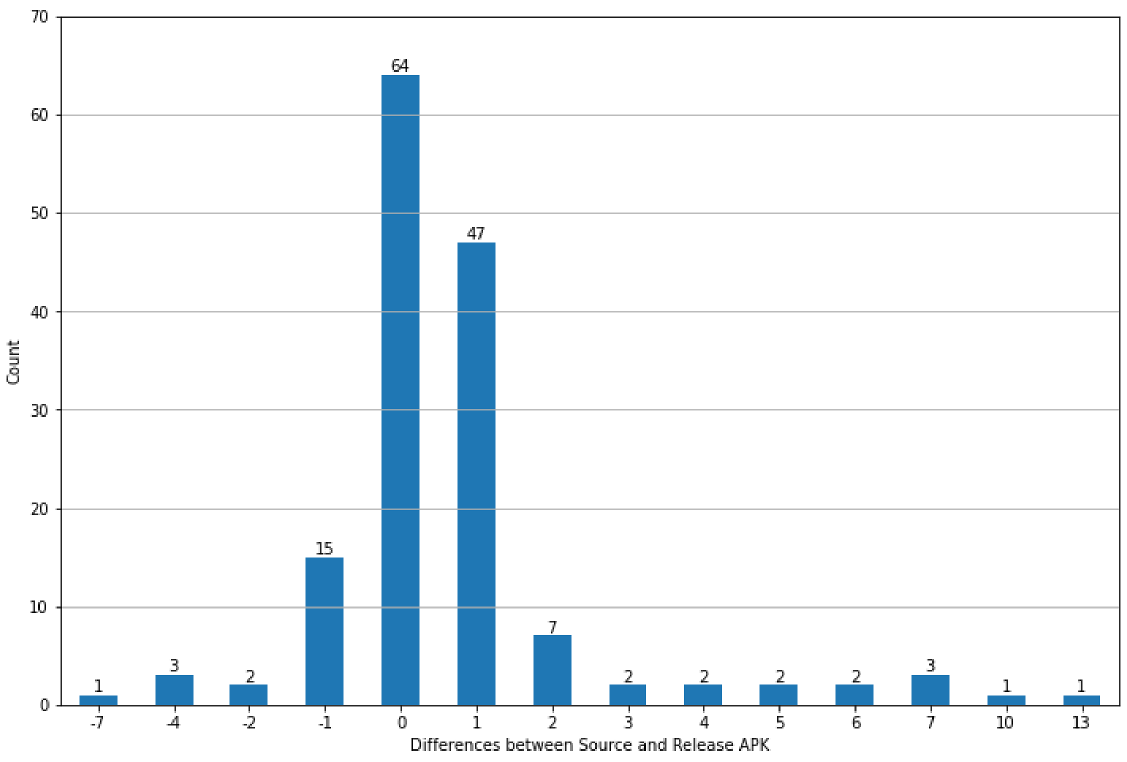


Figure 6.4: Histogram of detection differences between Source and Release

detector represented individually in a stack for each application, sorted by the difference in total number of detections. Positive values represent more detections on source code compared to APK, and negative values represent the opposite. As a side note, the detectors are colour-coded, and the same colours will be used in the next section for consistency and ease of comparison.

The differences can be attributed to different situations. Positive values can point to cases where the compiler optimised the pattern and removed it, or it could be when the decompilation process fails to reconstruct the Java code accurately. Negative values can result from artefacts introduced by the decompilation process.

The catalogues we mentioned in Section 3.1 tested these patterns before and after refactoring them and proved them to be more energy-efficient. Therefore it is unlikely that the compilation process would introduce harmful energy patterns, and safe to assume that the compiler is not responsible for these negative values. We also did not extensively test the detectors, and some situations can be unaccounted for; This can also result in positive or negative differences. Further research is required in this area.

Overall, the results point favourably towards static analysis of decompiled code being viable and lend credence to our RQ. It is important to note that these results come from the total values for each detector and do not account for the similarity of detections. That is discussed in the next section.

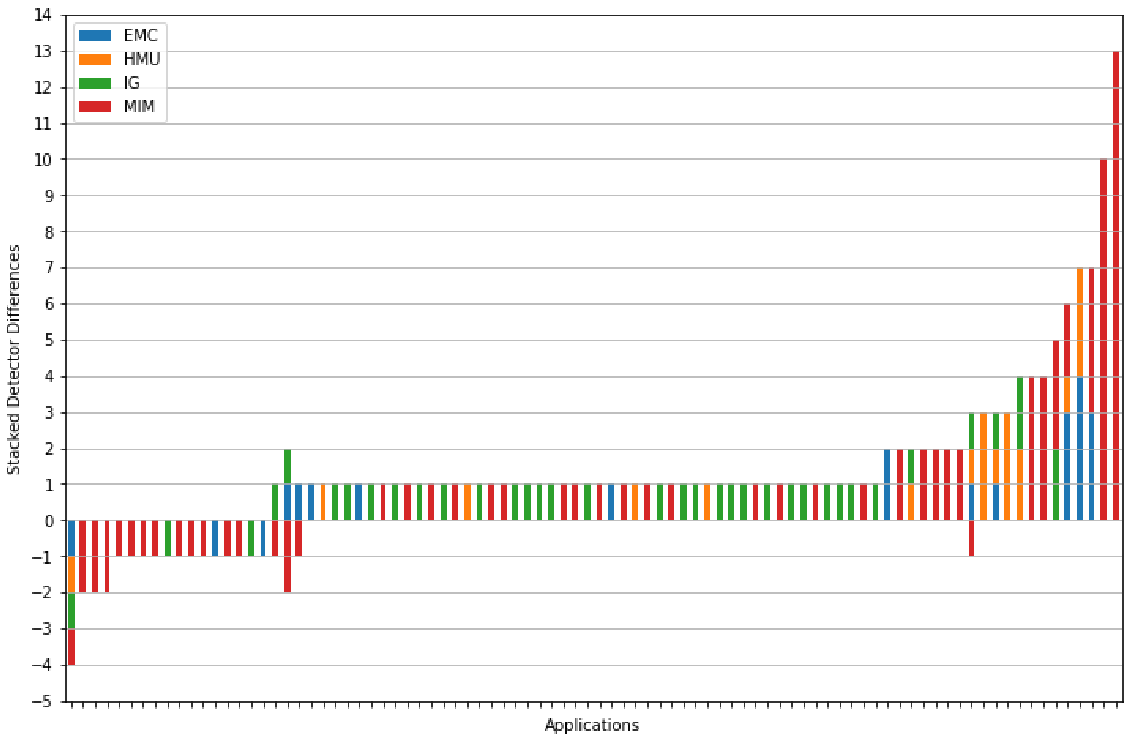


Figure 6.5: Detector Differences per Application between Source and APK

6.2 Analysis of similarity between detections

In this section, we complement the previous analysis by verifying the similarity score between detections, i.e., given a report from Source code and another from the Release *APK*, how similar are they?

We quantify the similarity using different metrics as demonstrated in Chapter 5 and we also maintain the same set of 152 applications in this analysis.

We start with the Jaccard Index, where the similarity score ranges from zero to one. Each detector has its own score, and cumulatively, an application can have a score ranging from zero to four. Figure 6.6 contains the histograms for our four detectors, and a logarithmic scale is used to visualise the smaller indexes more clearly.

In all detectors, the number of applications with a similarity score of one is almost an order of magnitude higher than the rest, i.e., for most applications, their reports contain the same number of patterns in the same locations. The second highest category in all detectors is where we have a similarity score of zero, i.e., the two reports, from Source and APK, do not have a single matching detection.

The results above explore the detectors by themselves, but to understand how they work together, we must look at the cumulative score. The 130 applications in detector *EMC* are not necessarily the same as the 131 applications from detector *HMU*. In Figure 6.7 we can get a better perspective of each application's similarity across all four detectors; in fact, only 54 applications,

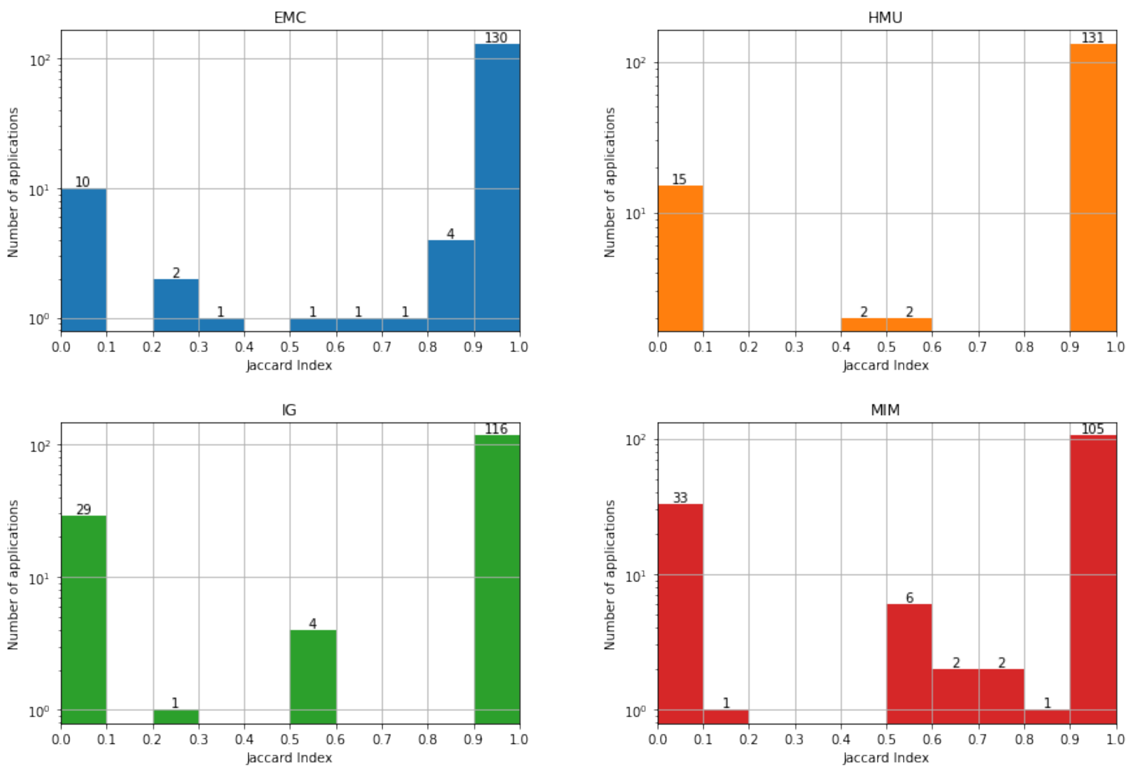


Figure 6.6: Jaccard Index per Detector between Source and Release APK

35.53%, have a similarity score of 4, contrary to what one might believe from Figure 6.6. However, 136 applications, 89.47%, have a score equal to or higher than three, i.e., the reports in three detectors are entirely similar, and only two applications got a score of zero.

The Jaccard Index can quantify and sort the applications by how similar they are between versions. In the previous section, in Figure 6.5, we filtered and sorted the applications that differed in the total number of detections. Figure 6.8 redos the same analysis but this time using the cumulative Jaccard Index as a filter (cumulative Jaccard Index < 4) and sorting value.

The difference between both figures is the eight applications with a value of zero, i.e., cases where we have the same number of detections in all detectors, but they are not entirely similar. Given that only eight applications went undetected, purely looking at the total number of detections, as we did in the previous section, is a somewhat reliable method of comparing versions and provides an accurate profile of the collected data.

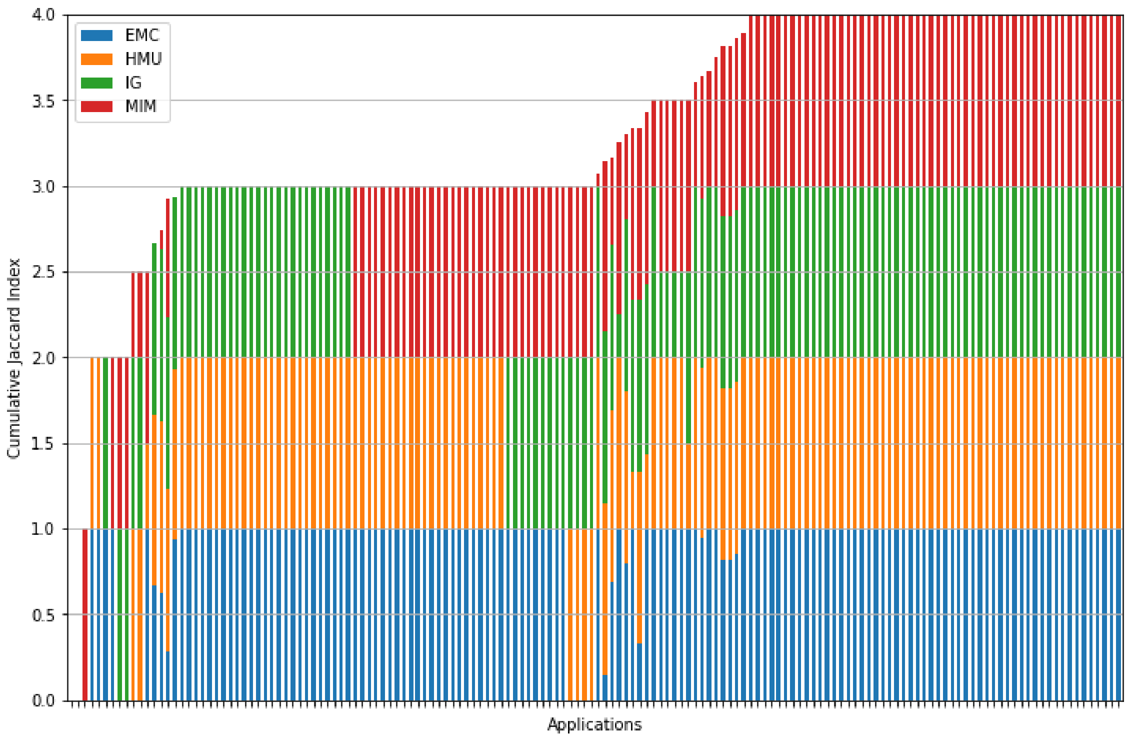


Figure 6.7: Cumulative Jaccard Index per Detector per Application between Source and APK

We also wanted to verify and reinforce the adequacy of testing similarity scores with the Jaccard Index. We introduced Fuzzy Matching as an alternative metric alongside Jaccard Index in Chapter 5, and tested three variants: *FuzzyConcat*, *FuzzyProcess* and *FuzzyJaccard*.

In Figure 6.9 we can see the scores per detector, as we did for the Jaccard Index in Figure 6.6. The differences are: a slightly smaller number of applications in the extreme bins, similarities of zero and one; the values in between are also slightly shifted to the right, towards one. Overall the results are almost indistinguishable, which is true for all three variants.

In Table 6.4 we can see how the four metrics compare numerically and confirm our initial predictions. All metrics show a similar result and help reinforce the Jaccard Index as the more conservative, worst-case scenario metric. Meanwhile, Fuzzy Matching, in particular, *FuzzyConcat* serves as a more optimistic metric.

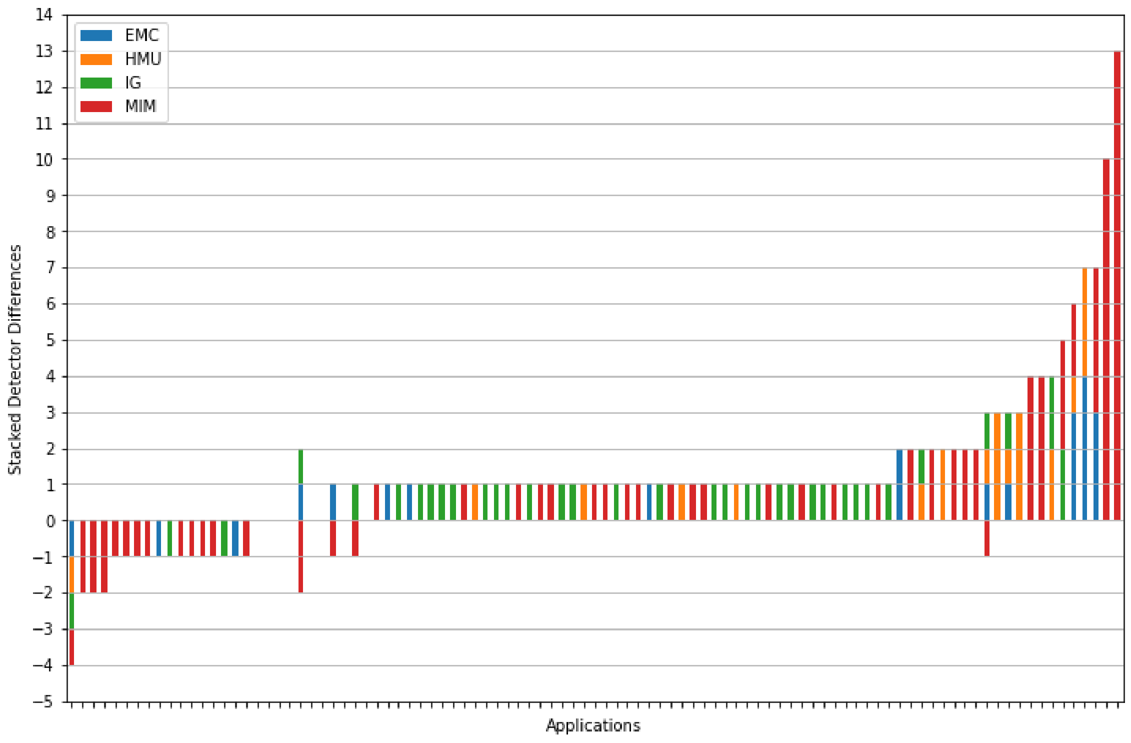


Figure 6.8: Detector Differences per Application between Source and APK

	Mean	SD	25%		Mean	SD	25%
EMC				**IG**			
Jaccard	0.90	0.28	1.00	**Jaccard**	0.79	0.40	1.00
FuzzyConcat	0.93	0.23	1.00	**FuzzyConcat**	0.80	0.39	1.00
FuzzyProcess	0.93	0.23	1.00	**FuzzyProcess**	0.79	0.40	1.00
FuzzyJaccard	0.93	0.23	1.00	**FuzzyJaccard**	0.79	0.40	1.00
HMU				**MIM**			
Jaccard	0.88	0.32	1.00	**Jaccard**	0.75	0.41	0.50
FuzzyConcat	0.92	0.26	1.00	**FuzzyConcat**	0.76	0.40	0.74
FuzzyProcess	0.92	0.25	1.00	**FuzzyProcess**	0.76	0.40	0.59
FuzzyJaccard	0.91	0.27	1.00	**FuzzyJaccard**	0.78	0.40	0.58

Table 6.4: Descriptive statistics for similarity metrics

To conclude this section, we also did some statical analysis within the limits of our sample data. At best, we can calculate the range of values that are likely to contain the similarity score of the population, i.e., other applications, based on our population sample.

In Table 6.5 we can see the 95% confidence interval for the mean of each detector and their cumulative mean, for *Jaccard* and *FuzzyConcat*. What we can gather from these intervals is that for the worst performing detector, *MIM*, we get a 95% confidence level that the true value of similarity between Source code and a Release *APK* is at least 0.69 when calculated by the more

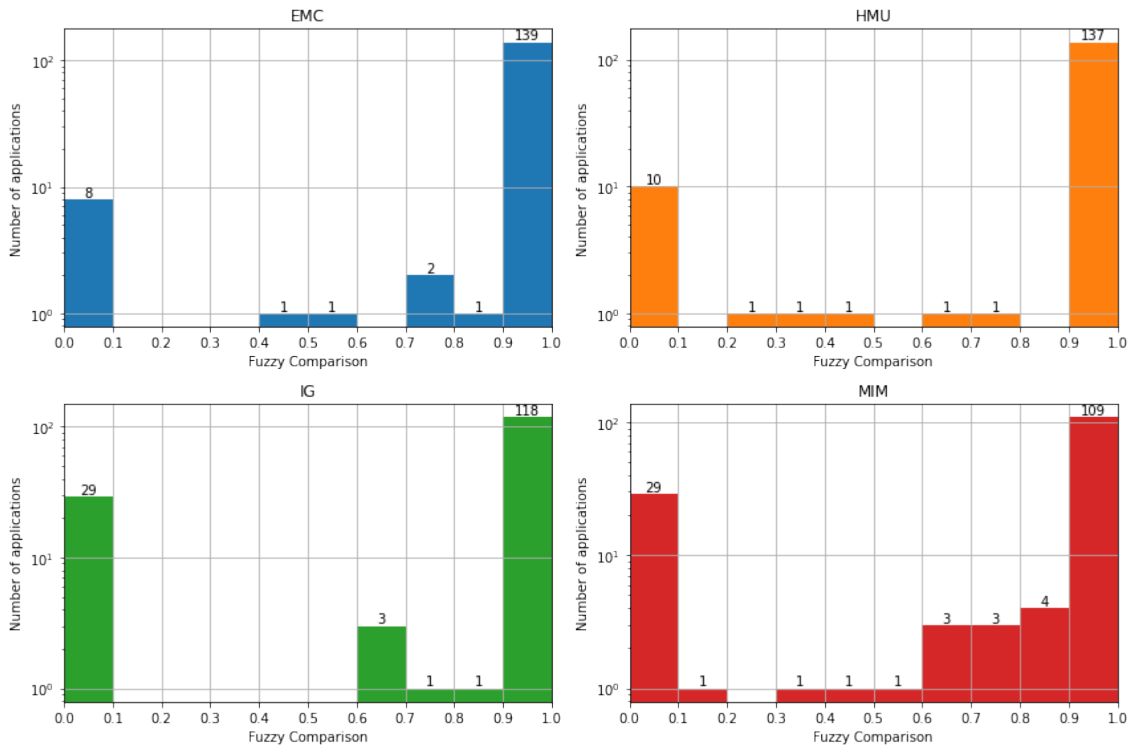


Figure 6.9: Fuzzy Matching per Detector between Source and Release APK

conservative Jaccard Index.

As for the true value of cumulative similarity from all four detectors, we can say that it is between 3.26 and 3.38. These values might, of course, be dependent on other variables that were not accounted for in our tests, and the usage of more detectors can skew the results in either direction, but overall we got positive results supporting our RQ.

6.3 Summary

We tested 420 applications, 152 of which reported at least one detection. The number of detections in Source code was, in all detectors, superior to the *APK* versions, with an average difference of 11% between the two. When analysing the impact of obfuscation, due to the small sample size, results were inconclusive, and further testing with a bigger sample is required.

Our analysis of the total number of detections primarily showed that the compilation and decompilation process is not systematically introducing detections, with 79.05% of the 420 applications reporting an equal number of detections between Source and Release *APK*. We also compared the similarity of detections between those two versions and got reinforcing results. From the group of 152 applications, 34.53% got a perfect similarity score of four, with 89.47% getting a score of three or more out of four. Meanwhile, only two applications got completely different reports with a score of zero.

	Lower Bound	Upper Bound
	Jaccard	
EMC	0.84	0.96
HMU	0.82	0.94
IG	0.73	0.85
MIM	0.69	0.81
Cumulative	3.26	3.38
	FuzzyConcat	
EMC	0.87	1.00
HMU	0.85	0.98
IG	0.74	0.86
MIM	0.71	0.84
Cumulative	3.36	3.49

Table 6.5: 95% Confidence Interval for the Mean

We calculated the 95% confidence intervals for each detector and found that for the worst performing detector, *MIM*, the true value of the similarity score in this population is between 0.69 and 0.81 with the Jaccard Index. We did the same for the cumulative value of all four detectors and got an interval between 3.36 and 3.49.

The two analyses we performed confirm and reinforce themselves. We know the detections between Source code and Release *APK* are reasonably similar. We know that even comparing the total number of detections is, by itself, a somewhat reliable mechanism to evaluate the presence of energy patterns. Given all this, we can safely claim that static analysis of decompiled applications is indeed viable, as we proposed in our RQ and we believe this area should be further explored.

A replication package is available in a zenodo repository [1]. It contains instructions for preparing the dataset used, executing Kadabra and E-APK, and reproducing the analysis results and graphs.

In the next and final chapter, we present our overall conclusions for this work, the main contributions achieved and what can be done in the future to expand the work done so far.

[1]`https://doi.org/10.5281/zenodo.7083540` (Last Access: Aug 16, 2022)

Chapter 7

Conclusions and Future Work

Energy efficiency is an important consideration when developing mobile applications. The current literature provides catalogues of good energy practices and automatic detection and refactoring tools for developers. However, there is a noticeable requirement among these tools: they need access to an application's source code. Besides the original developer, other stakeholders, most notably App Stores, that want to perform an independent energy analysis of applications lack the means to do so.

To bridge this gap, we took 420 open-source Android applications to understand if energy patterns present in their source code can be detected by decompiling and analysing the resulting code. We developed a module capable of detecting Android energy patterns in Java source code. To analyse APKs directly, we also extended Kadabra to transparently decompile the Dex bytecode from *APK*s into Java source code.

We comparatively analysed source code against decompiled code and found that approximately 79.05% of the applications reported no difference in the number of detections. We proceded with a more in-depth analysis, trying to correspond each detection instance individually in the source code and the decompiled code.

A similarity score from 0 to 4 was attributed to each application, with 34.53% of all applications getting a perfect similarity score of 4, and 89.47% getting a score of three or more out of four. Meanwhile, only two applications got completely different reports with a score of zero.

We also calculated the 95% confidence intervals for each detector and found that for the worst performing detector, *MIM*, the true value of the similarity score in this population is between 0.69 and 0.81 with the Jaccard Index. We did the same for the cumulative value of all four detectors and got an interval between 3.36 and 3.49.

The execution time per application was acceptable, ranging from 10 seconds to over a minute in some cases, but cumulatively, it took over three hours. We accelerated the analysis process by taking advantage of Kadabra's *runParallel* feature to analyse each version in parallel. The total execution was roughly reduced by a factor of four to less than an hour. Furthermore, we can also parallelise each detector for even faster times; this opens the possibility of testing of even larger datasets and facilitate a more iterative analysis.

Overall, our results support our RQ: static analysis techniques, typically used in source code, might be viable to inspect *APK*s when access to source code is restricted. We believe this area should be further explored, and in Section 7.2, we expand upon potential research paths that can complement our work.

7.1 Main Contributions

Here we present the most significant milestones we achieved as a way to synthesise the work:

- Integration of an *APK* decompiler, Jadx, into Kadabra to transparently decompile *APK*s and analyse their decompiled code.
- Development of a collection of scripts for Kadabra, E-APK, to detect energy code patterns.
- Large-scale testing, with over 400 Android applications, of Kadabra and our E-APK scripts.
- Assessment of the viability of energy code pattern detection in *APK*s as a method of performing independent energy analysis.
- Publication of a conference paper [29] with partial findings of our work:
  N. Gregório, J. Bispo, J. P. Fernandes and S. Q. Medeiros (2022). E-APK: Energy Pattern Detection in Decompiled Android Applications. 26th Brazilian Symposium on Programming Languages, 2022.

Additionally, our work also resulted in:

- Identifying a feature inconsistency in the decompiler Jadx, and creating a pull request [1] that was accepted by Jadx's team.
- Deployment of our solution in other research done within the GreenStamp project.

7.2 Future Work

Future work includes performing a complementary analysis of the bytecode generated from the compilation process to possibly identify false positives and artifacts from the decompilation tool; this comes with its own challenges however, since it requires implementing detectors for bytecode and matching the resulting detections between source code and bytecode.

Due to limitations in the dataset we used, the impact of obfuscation was not explored. Reproducing the analysis we did but with pairs of application, obfuscated and non-obfuscated, could provide insights into the impact of obfuscation on the detection of patterns; this could be done while testing different decompilers and deobfuscation tool for a clear understanding of the current capabilities and limitations of state of the art decompilation and deobfuscation tools.

[1]`https://github.com/skylot/jadx/pull/1467` (Last Access: Sep 14, 2022)

Another important aspect is the possibility of generalising this type of analysis to other languages. Our dataset was limited to Android applications written in Java, but Kotlin is growing in popularity [2] and should also be studied. Our detectors have some flexibility here since the LARA framework has multi-language support and can be extended to also work with Kotlin, and the detections scripts might not even require changes.

Other efforts worth looking into are:

- The impact of compilation and optimisation flags and how often developers take advantage of them.
- Extending the available detectors to increase pattern coverage and understand which patterns are more susceptible to interference from compilation/decompilation.
- Repeat the experiment with a different dataset with older or newer applications targetting different Android versions.

7.3 Acknowledgments

This work was financed by FEDER (Fundo Europeu de Desenvolvimento Regional), from the European Union through CENTRO 2020 (Programa Operacional Regional do Centro), under project CENTRO-01-0247-FEDER-047256 – GreenStamp: Mobile Energy Efficiency Services.

[2]https://blog.jetbrains.com/kotlin/2021/09/the-actual-number-of-kotlin-developers-or-who-our-a
(Last Access: Sep 09, 2022)

References

[1] Simone Aonzo, Gabriel Claudiu Georgiu, Luca Verderame, and Alessio Merlo. Obfuscapk: An open-source black-box obfuscation tool for android apps. *SoftwareX*, 11:100403, 2020.

[2] Yauhen Arnatovich, Hee Beng Kuan Tan, Sun Ding, Kaiping Liu, and Lwin Khin Shar. Empirical comparison of intermediate representations for android applications. In *SEKE*, pages 205–210, 2014.

[3] Yauhen Leanidavich Arnatovich, Lipo Wang, Ngoc Minh Ngo, and Charlie Soh. A comparison of android reverse engineering tools via program behaviors validation based on intermediate languages transformation. *IEEE Access*, 6:12382–12394, 2018.

[4] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, dec 1994.

[5] Vivek Balachandran, Sufatrio, Darell J.J. Tan, and Vrizlynn L.L. Thing. Control flow obfuscation for android applications. *Computers & Security*, 61:72–93, 2016.

[6] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. EnergyPatch: Repairing Resource Leaks to Improve Energy-Efficiency of Android Apps. *IEEE Transactions on Software Engineering*, 44(5):470–490, May 2018.

[7] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, page 27–38, New York, NY, USA, 2012. Association for Computing Machinery.

[8] Richard Baumann, Mykolai Protsenko, and Tilo Müller. Anti-proguard: Towards automated deobfuscation of android apps. In *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*, SHCIS '17, page 7–12, New York, NY, USA, 2017. Association for Computing Machinery.

[9] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, page 343–355, New York, NY, USA, 2016. Association for Computing Machinery.

[10] João Bispo and João M.P. Cardoso. Clava: C/c++ source-to-source compilation using lara. *SoftwareX*, 12:100565, 2020.

[11] Martin Brylski. Android smells catalogue. https://martinbrylski.github.io/android_smells/index.html, 2013.

[12] Huaqian Cai, Ying Zhang, Zhi Jin, Xuanzhe Liu, and Gang Huang. DelayDroid: Reducing Tail-Time Energy by Refactoring Android Apps. In *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, Internetware '15, pages 1–10, New York, NY, USA, November 2015. Association for Computing Machinery.

[13] João Cardoso, José Coutinho, Tiago Carvalho, Pedro Diniz, Zlatko Petrov, Wayne Luk, and Fernando Gonçalves. Performance-driven instrumentation and mapping strategies using the lara aspect-oriented programming approach. *Software: Practice and Experience*, 46, 12 2014.

[14] Antonin Carette, Mehdi Adel Ait Younes, Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. Investigating the energy impact of Android smells. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 115–126, February 2017.

[15] M Cayrol, H Farreny, and H Prade. Fuzzy pattern matching. *Kybernetes*, 1982.

[16] Marco Couto, João Saraiva, and João Paulo Fernandes. Energy Refactorings for Android in the Large and in the Wild. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 217–228, February 2020.

[17] Luis Cruz and Rui Abreu. Catalog of energy patterns for mobile applications. *Empirical Software Engineering*, 24(4):2209–2235, August 2019.

[18] Luis Cruz and Rui Abreu. Improving Energy Efficiency Through Automatic Refactoring. *J. Softw. Eng. Res. Dev.*, 2019.

[19] Anthony Desnos and Geoffroy Gueguen. Android: From reversing to decompilation. *Proc. of Black Hat Abu Dhabi*, 01 2011.

[20] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, XiaoFeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. In Raheem Beyah, Bing Chang, Yingjiu Li, and Sencun Zhu, editors, *Security and Privacy in Communication Networks*, pages 172–192, Cham, 2018. Springer International Publishing.

[21] David Ehringer. The dalvik virtual machine architecture. *Techn. report (March 2010)*, 4(8):72, 2010.

[22] Iffat Fatima, Hina Anwar, Dietmar Pfahl, and Usman Qamar. Detection and Correction of Android-specific Code Smells and Energy Bugs: An Android Lint Extension. In *QuA-SoQ@APSEC*, 2020.

[23] Daniel Feitosa, Luís Cruz, Rui Abreu, João Paulo Fernandes, Marco Couto, and João Saraiva. Patterns and Energy Consumption: Design, Implementation, Studies, and Stories. In Coral Calero, Mª Ángeles Moraga, and Mario Piattini, editors, *Software Sustainability*, pages 89–121. Springer International Publishing, Cham, 2021.

[24] Sam Fletcher, Md Zahidul Islam, et al. Comparing sets of patterns with the jaccard index. *Australasian Journal of Information Systems*, 22, 2018.

[25] Abderraouf Gattal, Abir Hammache, Nabila Bousbia, and Adel Nassim Henniche. Exploiting the progress of oo refactoring tools with android code smells: Randroid, a plugin for android

studio. In *Proceedings of the 36th Annual ACM Symposium on Applied Computing*, SAC '21, page 1580–1583, New York, NY, USA, 2021. Association for Computing Machinery.

[26] Anton B. Georgiev, Alberto Sillitti, and Giancarlo Succi. Open source mobile virtual machines: An energy assessment of dalvik vs. art. In Luis Corral, Alberto Sillitti, Giancarlo Succi, Jelena Vlasenko, and Anthony I. Wasserman, editors, *Open Source Software: Mobile Open Source Technologies*, pages 93–102, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[27] Olivier Le Goaër. Enforcing Green Code With Android Lint. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 85–90, September 2020.

[28] David Gonzalez. Aeon: Automated android energy-efficiency inspection. https://plugins.jetbrains.com/plugin/7444-aeon-automated-android-energy-efficiency-inspection, 2017.

[29] Nelson Gregório, João Bispo, João Paulo Fernandes, and Sérgio Queiroz Medeiros. E-apk: Energy pattern detection in decompiled android applications. In *2022 26th Brazilian Symposium on Programming Languages*, 2022.

[30] Sarra Habchi, Naouel Moha, and Romain Rouvoy. Android Code Smells: From Introduction to Refactoring. *arXiv:2010.07121 [cs]*, October 2020.

[31] Patrick A. V. Hall and Geoff R. Dowling. Approximate string matching. *ACM Comput. Surv.*, 12(4):381–402, dec 1980.

[32] Geoffrey Hecht, Romain Rouvoy, Naouel Moha, and Laurence Duchien. Detecting Antipatterns in Android Apps. In *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, pages 148–149, May 2015.

[33] Emanuele Iannone, Fabiano Pecorelli, Dario Di Nucci, Fabio Palomba, and Andrea De Lucia. Refactoring Android-specific Energy Smells: A Plugin for Android Studio. In *Proceedings of the 28th International Conference on Program Comprehension*, pages 451–455. Association for Computing Machinery, New York, NY, USA, July 2020.

[34] Paul Jaccard. The distribution of the flora in the alpine zone. 1. *New phytologist*, 11(2):37–50, 1912.

[35] Heejun Jang, Beomjin Jin, Sangwon Hyun, and Hyoungshick Kim. Kerberoid: A practical android app decompilation system with multiple decompilers. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 2557–2559, New York, NY, USA, 2019. Association for Computing Machinery.

[36] Hao Jiang, Hongli Yang, Shengchao Qin, Zhendong Su, Jian Zhang, and Jun Yan. Detecting energy bugs in android apps using static analysis. In Zhenhua Duan and Luke Ong, editors, *Formal Methods and Software Engineering*, pages 192–208, Cham, 2017. Springer International Publishing.

[37] Jozef Kostelanský and L'ubomír Dedera. An evaluation of output from current java bytecode decompilers: Is it android which is responsible for such quality boost? In *2017 Communication and Information Technologies (KIT)*, pages 1–6, 2017.

[38] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. In *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, volume 15, 2011.

[39] Guangtai Liang, Jian Wang, Shaochun Li, and Rong Chang. Patbugs: A pattern-based bug detector for cross-platform mobile applications. In *2014 IEEE International Conference on Mobile Services*, pages 84–91, 2014.

[40] Yu Lin, Semih Okur, and Danny Dig. Study and Refactoring of Android Asynchronous Programming (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 224–235, November 2015.

[41] Noah Mauthe, Ulf Kargén, and Nahid Shahmehri. A large-scale empirical study of android app decompilation. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 400–410, 2021.

[42] Rodrigo Morales, Rubén Saborido, Foutse Khomh, Francisco Chicano, and Giuliano Antoniol. EARMO: An Energy-Aware Refactoring Approach for Mobile Apps. *IEEE Transactions on Software Engineering*, 44(12):1176–1206, December 2018.

[43] Man D. Nguyen, Thang Q. Huynh, and T. Hung Nguyen. Improve the Performance of Mobile Applications Based on Code Optimization Techniques Using PMD and Android Lint. In Van-Nam Huynh, Masahiro Inuiguchi, Bac Le, Bao Nguyen Le, and Thierry Denoeux, editors, *Integrated Uncertainty in Knowledge Modelling and Decision Making*, Lecture Notes in Computer Science, pages 343–356, Cham, 2016. Springer International Publishing.

[44] Renaud Pawlak, Martin Monperrus, Nicolas Petitprez, Carlos Noguera, and Lionel Seinturier. Spoon: A library for implementing analyses and transformations of java source code. *Softw. Pract. Exper.*, 46(9):1155–1179, sep 2016.

[45] Matej Petkovic, Blaz Skrlj, Dragi Kocev, and Nikola Simidjievski. Fuzzy jaccard index: A robust comparison of ordered lists. *CoRR*, abs/2008.02216, 2020.

[46] Pedro Pinto, Tiago Carvalho, João Bispo, Miguel António Ramalho, and João M.P. Cardoso. Aspect composition for multiple target languages using lara. *Computer Languages, Systems & Structures*, 53:1–26, 2018.

[47] Jan Reimann, Martin Brylski, and Uwe Assmann. A Tool-Supported Quality Smell Catalogue For Android Developers. *Softwaretechnik-Trends*, January 2014.

[48] Luís Reis, Joao Bispo, and João Cardoso. Compilation of matlab computations to cpu/gpu via c/opencl generation. *Concurrency and Computation: Practice and Experience*, 32, 06 2020.

[49] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, page 49–61, New York, NY, USA, 1995. Association for Computing Machinery.

[50] Ana Ribeiro, João F. Ferreira, and Alexandra Mendes. Ecoandroid: An android studio plugin for developing energy-efficient java mobile applications. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*, pages 62–69, 2021.

[51] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press, 1999.

[52] Zhiwu Xu, Cheng Wen, and Shengchao Qin. State-taint analysis for detecting resource bugs. *Science of Computer Programming*, 162:93–109, 2018. Special Issue on TASE 2016.

[53] Geunha You, Gyoosik Kim, Seong-je Cho, and Hyoil Han. A comparative study on optimization, obfuscation, and deobfuscation tools in android. *J. Internet Serv. Inf. Secur.*, 11(1):2–15, 2021.

Appendix A

Energy Pattern table

Energy Pattern	**Also known as**
Draw Allocation [16, 30]	Init OnDraw
Wakelock [16, 17, 47, 11]	Resource Leak, Durable Wake-Lock, Race-to-idle
Recycle [16]	
Obsolete Layout Parameter [16]	
View Holder [16, 47, 11]	Uncached Views
HashMap Usage [16, 42, 30]	
Excessive Method Calls [16]	
Member Ignoring Method [16, 47, 11, 30]	
Dark UI Colors [17]	
Dynamic Retry Delay [17]	
Avoid Extraneous Work [17]	
Push over Poll [17]	
Power Save Mode [17]	
Power Awareness [17]	
WiFi over Cellular [17]	
Suppress Logs [17]	
Batch Operations [17]	
Cache [17]	
Decrease Rate [17]	
User Knows Best [17]	
Inform Users [17]	
Enough Resolution [17]	
Sensor Fusion [17]	
Kill Abnormal Tasks [17]	
No Screen Interaction [17]	

Avoid Extraneous Graphics and Animations [17]	
Manual Sync, On Demand [17]	
Bulk Data Transfer On Slow Network [47, 11]	
Data Transmission Without Compression [47, 11, 17]	Reduce Size
Debuggable Release [47, 11]	
Early Resource Binding [47, 11, 17, 42]	Binding Resources too early, Open Only When Necessary
Inefficient Data Structure [47, 11]	
Inefficient SQL Query [47, 11]	
Inefficient Data Format And Parser [47, 11]	
Internal Getter/Setter [47, 11, 42]	Private getters and setters
Leaking Inner Class [47, 11, 30]	
Leaking Thread [47, 11]	
Nested Layout [47, 11]	
No Low Memory Resolver [47, 11, 30]	
Overdrawn Pixel [47, 11, 30]	UI Overdraw
Prohibited Data Transfer [47, 11]	
Rigid AlarmManager [47, 11]	
Set Config Changes [47, 11]	
Slow Loop [47, 11]	
Unclosed Closable [47, 11]	
Blob [42]	
Lazy Class [42]	
Long-parameter list [42]	
Refused Bequest [42]	
Speculative Generality [42]	
Unsupported Hardware Acceleration [30]	
Unsuited LRU Cache Size [30]	
Lifetime Containment [22]	
Bitmap Format Usage [22]	
Heavy AsyncTask [22, 40]	
Heavy Service Start [22, 25]	
Heavy Broadcast Receiver [22]	
Vacuous Background Services [22, 12, 6, 27]	Sensor Coalesce
Immortality Bug [6]	
Battery-Efficient Location [27]	

Table A.1: Energy Patterns