UNIVERSIDADE
BEIRA INTERIOR

Programming Languages and Static Analysis Techniques For Software Energy certification

(Versão final após defesa)

Délcio Kelson Alves Ferramenta

Dissertação para obetenção do grau de Mestre em

Engenharia informática

(2ºCiclo de estudos)

Orientador: Prof. Doutor Simão Patricio Melo de Sousa

Dezembro, 2022

Declaração de Integridade

Eu, Délcio Kelson Alves Ferramenta, que abaixo assino, estudante com número de inscrição M10536 do curso de Engenharia Informática da Faculdade de engenharia, declaro ter desenvolvido o presente trabalho e elaborado o presente texto em total consonância com o Código de Integridade da Universidade da Beira Interior.

Mais concretamente afirmo não ter incorrido em qualquer das variedades de Fraude Académica, e que aqui declaro conhecer, e que em particular atendi à exigida referenciação de frases, extratos, imagens e outras formas de trabalho intelectual, e assim assumo na íntegra as responsabilidades da autoria.

Universidade da Beira Interior, Covilhã 10/10/2022

(assinatura conforme Cartão de Cidadão ou preferencialmente assinatura digital no documento original se naquele mesmo formato)

Délcio K. A. Ferramenta

Acknowledgments

Despite the self-confidence provided by the knowledge acquired along my academic path, this work had great support, both direct and indirect, that allowed me to achieve the desired objectives. Thus, I show my sincere gratitude especially to:

Professor Doctor Simão Melo de Sousa was a great motivator and gave me many ideas on various aspects of the project, in addition to fulfilling his role as my supervisor.

My girlfriend, friends, and family motivated me to overcome the various obstacles throughout the project's development.

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.

Resumo

Os dispositivos móveis, particularmente os *smartphones*, tornaram-se uma extensão dos nossos corpos por uma variedade de razões, principalmente mobilidade, conveniência de comunicação e uma vasta gama de aplicações disponíveis. No entanto, várias limitações sugerem pela portabilidade destes dispositivos. A limitação comummente apontada pelos utilizadores é a duração limitada da bateria. Os fabricantes de *smartphones* tentam resolver este problema através da otimização do hardware e software, mas o problema persiste. É bem-sabido que as aplicações são um dos elementos que consomem mais energia nos *smartphones*, no entanto, a maioria dos desenvolvedores de aplicações não utiliza ou sequer considera a utilização de estratégias para minimizar o consumo de energia das suas aplicações. Uma forma prática para resolver este problema é forçar os desenvolvedores a se preocuparem mais sobre o consumo de energia das suas aplicações, criando um sistema de catálogo que classifica as aplicações com base no seu consumo de energia. Neste trabalho o nosso objetivo é desenvolver uma ferramenta capaz de fornecer uma métrica que poderá ser usado para fins de comparação entre as aplicações. A métrica que nos propomos em calcular é o *Worst-Case Energy Consumption (WCEC)*, que representa a energia consumida no caso mais extremo da execução de um programa. Esta abordagem é normalmente empregue em alguns sistemas embutidos, nos quais a duração da bateria deve ser rigorosamente calculada para evitar vários constrangimentos. Também ilustramos um cenário para demonstrar como a nossa ferramenta pode ser utilizada. O objetivo deste documento é explicar os princípios usados na nossa abordagem e descrever em pormenor os passos usados na implementação da nossa ferramenta.

Palavras-chave

Análise Estática, consumo de energia no pior caso, modelo energético, análise de recurso, interpretação abstrata.

Resumo Alargado

Os dispositivos móveis, particularmente os *smartphones*, tornaram-se uma extensão dos nossos corpos por uma variedade de razões, principalmente mobilidade, conveniência de comunicação e uma vasta gama de aplicações disponíveis. No entanto, várias limitações sugerem pela portabilidade destes dispositivos. A limitação comummente apontada pelos utilizadores é a duração limitada da bateria. Os fabricantes de *smartphones* tentam resolver este problema através da otimização do hardware e software, mas o problema persiste. É bem-sabido que as aplicações são um dos elementos que consomem mais energia nos *smartphones*, no entanto, a maioria dos desenvolvedores de aplicações não utilizam, ou mesmo consideram utilizar, estratégias para minimizar o consumo de energia das aplicações. Uma forma prática para resolver este problema é forçar os desenvolvedores a se preocuparem mais sobre o consumo de energia das suas aplicações, criando um sistema de catálogo que classifica as aplicações com base no seu consumo de energia. Neste trabalho o nosso objetivo é desenvolver uma ferramenta capaz de fornecer uma métrica que poderá ser usado para fins de comparação entre as aplicações. A métrica que nos propomos em calcular é WCEC, que representa a energia consumida no caso mais extremo da execução de um programa. Esta abordagem é normalmente empregue em alguns sistemas embutidos, nos quais a duração da bateria deve ser rigorosamente calculada para evitar vários constrangimentos. Na nossa abordagem, nós usamos as técnicas que são normalmente adotadas na análise WCEC para serem usadas em aplicações móveis. Mas a análise WCEC é um tópico relativamente pouco explorado, dificultando o estudo das técnicas normalmente implementadas neste contexto. Portanto, fizemos um estudo à análise Worst-Case Execution Time (WCET), na qual a análise WCEC normalmente se baseia. Esta análise, comparada com a análise WCEC, está melhor consolidada, e há mais estudos sobre a mesma, tornando-a mais fácil de compreender. Portanto, o estudo da análise WCET fornece uma base forte para uma melhor compreensão dos trabalhos sobre a análise WCEC.

Depois de um estudo profundo sobre as análises WCET e WCEC notamos que existem de fato várias semelhanças, principalmente nas técnicas que usam. As técnicas que são normalmente empregues em ambas análises são: análise estática e Implicit Path Enumeration Technique (IPET). A análise estática é uma técnica que permite a análise de programas sem a necessidade de executa-los. Esta análise é usada para extrair informações importantes sobre o programa, por exemplo, o valor de cada variável do programa. Estas informações podem afetar drasticamente os resultados das análises WCET e WCEC. O IPET é uma técnica que é exaustivamente usada pelas ferramentas WCET e WCEC. Esta técnica consiste na translação do problema do cálculo do WCET WCEC para um problema Integer Linear Programming (ILP), que pode ser resolvido por ferramentas já bem conhecidas. O IPET é usado na fase final da análise WCET WCEC para gerar o resultado. Na nossa abordagem nós tiramos partido da análise estática e IPET.

O objetivo deste documento é explicar os princípios usados na nossa abordagem e descrever em pormenor os passos usados na implementação da nossa ferramenta. Também ilustramos um cenário para demonstrar como a nossa ferramenta pode ser utilizada.

Abstract

Mobile devices, such as smartphones, have become an extension of our body for various reasons, mainly because of mobility, communication convenience, and an extensive range of provided apps. However, being portable, they have several limitations, and the most concerning for users is the limited battery life. Smartphone manufacturers are trying to address this problem by optimizing the hardware and the software, but the concern remains. Although it is well known that apps are one of the components of smartphones that consume the most energy, most app developers do not use or even consider using strategies to minimize their apps' energy consumption. Creating a labeling system that classifies apps based on their energy consumption is a possible solution to drive developers to be more conscious of their apps' energy consumption. This work aims to develop a technique to compute a metric for app energy certification. The metric we propose to calculate is *WCEC*, which represents the energy consumed in the most extreme case of program execution. Typically, this analysis is used in embedded systems, where the energy consumed by the apps must be rigorously determined to avoid inconveniences. Here we have reused the fundamentals of the WCEC analysis to use for Android apps. We address our solution to the Android platform since it has the largest worldwide mobile operating system market share. To perform the WCEC analysis, we take advantage of static analysis and the IPET, the techniques commonly used in this context. We have also created a test scenario to illustrate a case where our tool can be used. This document aims to explain the fundamentals present in our tool and describe its implementation details.

Keywords

Static analysis, worst-case energy consumption, Instruction-level energy model, resource analysis, abstract interpretation, energy consumption analysis.

Contents

Contents viii

List of Figures x

List of Tables xi

1 Introduction 1
1.1 Motivation and Objectives 1
1.2 Main Contributions 2
1.3 Document organization 2

2 Fundamental Concepts and Related Work 3
2.1 Android App Fundamentals 3
2.2 WCET Analysis 4
2.3 WCEC Analysis 6
2.4 Cost Model 8
2.5 Static Analysis 9
2.6 Bound Calculation 12
2.7 Conclusions 13

3 Proposed Energy Cost Model 14
3.1 Target Device Overview 14
3.2 Instruction Level Analysis 15
3.3 Instruction Level Model 15
3.4 Conclusion 16

4 Implementation Details 17
4.1 Phases of The Proposed Technique 17
4.2 Data Preparation 17
4.3 Static Analysis 19
Value Analysis 19
Loop-Bound Analysis 19
4.4 Bound calculation 20
Path-based Method 20
IPET 21
4.5 Conclusion 21

5 Technique Evaluation 22
5.1 Tests Setup 22
5.2 Results 23
5.3 Conclusions 23

6 Conclusions And Future Work 25

6.1 Future Work 25

Bibliography **27**

List of Figures

2.1 A typical Android app's build process [1]. 4
2.2 Phases of computation of Abint aiT tool [2]. 6
2.3 Phases of computation of EnergyAnalyzer tool [3]. 7
2.4 Signs abstract domain. 10
2.5 Example of a translation of a CFG to equations system. 12

4.1 Phases of our WCEC technique. 18
4.2 Interval abstract domain. 19

5.1 Test App layout. 23
5.2 Comparison between the relative energy of different apps. 23

List of Tables

2.1 WCET tools comparison. 6

3.1 Simplified part of the model. 16

5.1 Results of the tests and the corresponding total energy consumption of instruction included in the model. 24

Acrónimos

WCEC Worst-Case Energy Consumption

WCET Worst-Case Execution Time

CFG Control Flow Graph

BCET Best-Case Execution Time

IDE Integrated Development Environment

IPET Implicit Path Enumeration Technique

ICT Information and Communication Technology

ILP Integer Linear Programming

SPARK Soot Pointer Analysis Research Kit

CG Call Graph

DAG Direct Acyclic Graph

APK Android Application Package

DVM Dalvik Virtual Machine

RISC Reduced Instruction Set Computer

Chapter

1

Introduction

Currently, the use of mobile devices has become an everyday activity. However, the limited battery life of these devices affects the user experience. Smartphone manufacturers are trying to mitigate this limitation by optimizing the hardware and the software, but the problem remains. Several factors contribute to faster battery discharge, some of which are [4]: battery size, display, chipset, and other active and passive apps. Most of these components are physical and thus irreplaceable, limiting user mitigation of this issue. The apps are one of the elements that the user can easily replace to extend the device's battery life. These apps are necessary as they extend the functionality of the devices. Apps actively contribute to battery drain as most users spend their time on apps [5]. There are many apps options for a specific task, giving the user the power to select the one closest to meeting their needs. Nevertheless, in the app stores, there is no information like the apps' energy certificate that indicates how much energy the apps can drain to the battery.

In this thesis, aware of these situations, we propose a prototype that provides a metric that can be used in the energy certification of apps. This thesis is a part of the GreenStamp project, which aims to explore and build novel techniques for analyzing and cataloging the energy efficiency of mobile applications in the app store. This work complements other approaches being investigated and implemented concurrently by other researchers in this project.

1.1 Motivation and Objectives

In this work, we aim to develop a technique to produce a metric for energy certification of Android apps. The metric we propose to calculate is WCEC, as the name suggests, represents the energy consumed in the worst-case execution of a program. The goal is to rate Android apps by their energy certificate in a labeling system. As a result, developers will be driven to optimize their apps; otherwise, they will have a negative score on the catalog system, chasing away potential users. From the developers' point of view, being energy efficient will be a differential for an app because users can choose to use energy-efficient alternatives. That is an enormous incentive to create energy-efficient apps, given that this is a highly competitive market.

We address our solution to the Android platform since it has the largest share of the

worldwide mobile operating system market share. Currently, the Android operating system ranks first among the most used mobile operating systems, with 71.54% of the market share [6].

Information and Communication Technology Information and Communication Technology (ICT), which encompasses the smartphones, consume a considerable amount of global energy (estimated 4-6%) and is predicted to increase considerably by 2030 [7]. According to experts, ICT tasks can be executed using less energy if energetically optimized [7]. Therefore, it is essential to have metrics that accurately express the energy efficiency level of the target devices to help identify potential opportunities for improvement.

1.2 Main Contributions

There is ample opportunity for contributions because energy consumption analysis is a relatively little-explored topic, particularly when it comes to mobile apps. We faced several obstacles while working on this project, which led to the following contributions:

1. An instruction-level energy model for a modern Android smartphone, a Xiaomi Redmi note 9 pro;
2. A prototype to Compute the WCEC of Android apps. The prototype source code is available in `https://github.com/DelcioKelson/wcec-android`.
3. A scenario that shows the applicability of our WCEC analysis prototype, using the instruction-level model that have created;
4. A paper in the process of being written.

1.3 Document organization

The rest of the document is summarized as follows:

- In chapter 2, we describe the fundamental concepts applied in our WCEC analysis;
- In chapter 3, we describe the steps we follow to create an energy cost model, which we use further on in our WCEC analysis;
- In chapter 4, we explain in detail how we implemented our WCEC analysis prototype;
- In chapter 5, we describe experiments to show the usability of our prototype;
- In chapter 6, we describe our main conclusions and propose future work.

Chapter 2

Fundamental Concepts and Related Work

This chapter explains the main WCEC analysis fundamentals. Before getting into the concepts of WCEC, in section 2.1, we will give an overview of the Android app structure, the primary input of our analysis. Then, section 2.2 describes the fundamental concepts and works around the WCET analysis, an well-researched analysis in which WCEC inherits many characteristics. Compared to the WCEC analysis, the WCET analysis is better consolidated, and there are more studies on it, making it easier to understand. Therefore, the study of the WCET analysis provides a strong foundation for a better understanding of the WCEC analysis works. An essential ingredient in WCEC / WCEC analysis is a cost model. This element is discussed in section 2.4. In the section 2.5 and 4.4 are described, respectively, the common phases of WCET and WCEC analysis, static analysis and bound calculation phases. Finally, section 2.7 highlights the chapter's most important conclusions.

2.1 Android App Fundamentals

An Android app can be written using Java and Kotlin and, in some specific cases, can be written in C++. More than source code, an Android app is also made up of multiple resources, for example, the XML representation of the app layouts. An Android app can be composed by combining four different components, each with a specific purpose and lifetime that specifies how the component is created and destroyed [8]. The components are:

- Activities: handle the user interaction with the application;
- Services: runs in the background;
- Broadcast Receivers: handle the communication between the Android operating system and app [9];
- Content Providers: handle data and database administration concerns [9].

As seen in figure 2.1, the building process of an Android app is relatively complex. The Android build system compiles and packs the source code and other resources into an

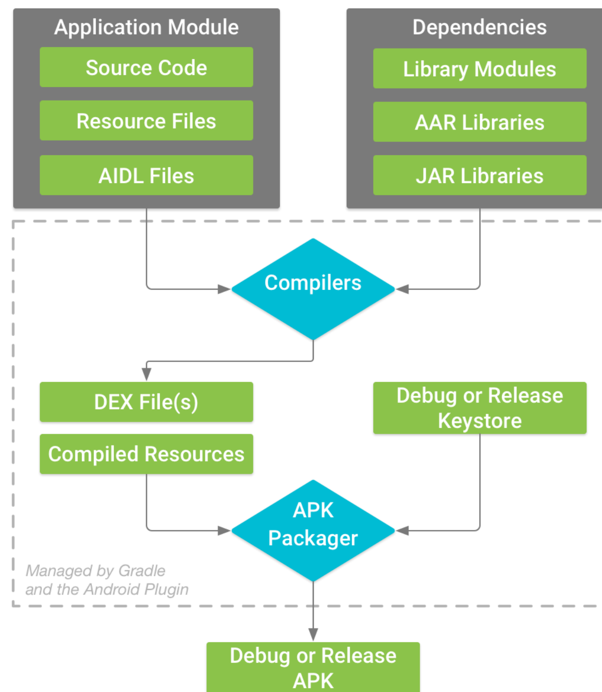

Figure 2.1: A typical Android app's build process [1].

Android Application Package (APK). In this build process, we highlight the translation of the source code into DEX (Dalvik Executable). These files interest us the most since they are the closest representation of the code that can be extracted from the app. After the build process, Android apps can be executed on a smartphone without many configurations. The apps run in an isolated virtual machine called Dalvik Virtual Machine (DVM).

2.2 WCET Analysis

The WCET represents the longest execution path of a program. Prediction of the WCET of tasks is commonly used in the embedded system context during the design phase to establish each activity's required hardware resources and time budget [10]. During implementation, critical system developers must ensure that the systems finish the tasks within the time constraints.

The resource consumption for the associated system will increase as the tasks WCET increase. It is vital to make safe estimations regarding the WCET, especially in real-time critical systems, as missing a time constraint would cause a catastrophic failure.

Different from WCET, there is another concept known as the Best-Case Execution Time (BCET), which indicates the shortest time possible during the execution of a program. The BCET is necessary when the quick termination of the program is a concern.

One important aspect to remember about the WCET is that it should not be underestimated but can be overestimated. The goal of the WCET estimation is to ensure that no execution time is longer than the time provided by the analysis. It is of the foremost importance that the result of a WCET is safe. Nevertheless, safety is not the only requirement

of the WCET analysis. The result should be as accurate as possible in order to be helpful. An exaggerated overestimation could be the safest way; however, it would require a more significant resource reserve, which would subsequently be unusable. On the other hand, underestimating WCET in critical systems might result in a disaster.

Unfortunately, the complexity of WCET analysis is increasing over the years because some hardware updates, such as memory system updates, processor design updates, and the introduction of multiprocessing in recent years, have significantly increased [11]. Another difficulty in WCET analysis is the timing anomaly, which affects most capable microprocessors and occurs when the local WCET does not imply the global worst-case [12].

The solutions addressed to the WCET estimation are limited. In general, it is just possible to estimate a precise WCET for constrained programs (for instance, without recursion or with no non-deterministic loops). The exact estimation of WCET can be tricky since it includes solving the intractable halting problem. However, real-time systems only utilize a kind of programming that ensures that programs always end [13].

The WCET can be done using one of the following approaches: measurement-based or static analysis [2]. Measurement-based consists of running the program several times with various inputs and recording the execution time of each test. The measurement-based cannot ensure safety. Since it is not feasible to consider all possible inputs, measurement-based does not guarantee that in all execution paths the WCET will not be exceeded. Different from the static analyses that can produce valid results for all possible inputs [14]. It is one of the reasons that static analysis has been used more than measurement-based.

Most well-known WCET tools compute the WCET in 3 phases: Control Flow Graph (CFG) building, static analyses, and bound calculation. One example is the aiT WCET Analyzer, whose calculation phases are shown in figure 2.2. aiT is a WCET tool from Absint, a company reference in developing programs for critical systems. In CFG[1] building phase, as the name suggests, a CFG is created. It is essential to have a CFG that represents as accurately as possible all the feasible execution paths of the program, to perform a complete analysis of the program. The other phases will be explained in the coming chapters.

There are some open-source tools and academic research prototypes that calculate WCET, such as:

- Chronos [15]: tool used to do timing analysis on embedded software, developed by the National University of Singapore (NUS) primarily for the academic research community. Chronos is built on the SimpleScalar, a processor architecture simulator [16]; it allows the modeling of various processor models;

- Heptane [17]: modular and flexible tool developed at IRISA/University of Rennes to estimate the upper bounds of execution times on simple Reduced Instruction Set Computer (RISC) [2] architectures;

[1] A graph representation of the program paths.
[2] A microprocessor architecture that uses a tiny, highly efficient set of instructions [18].

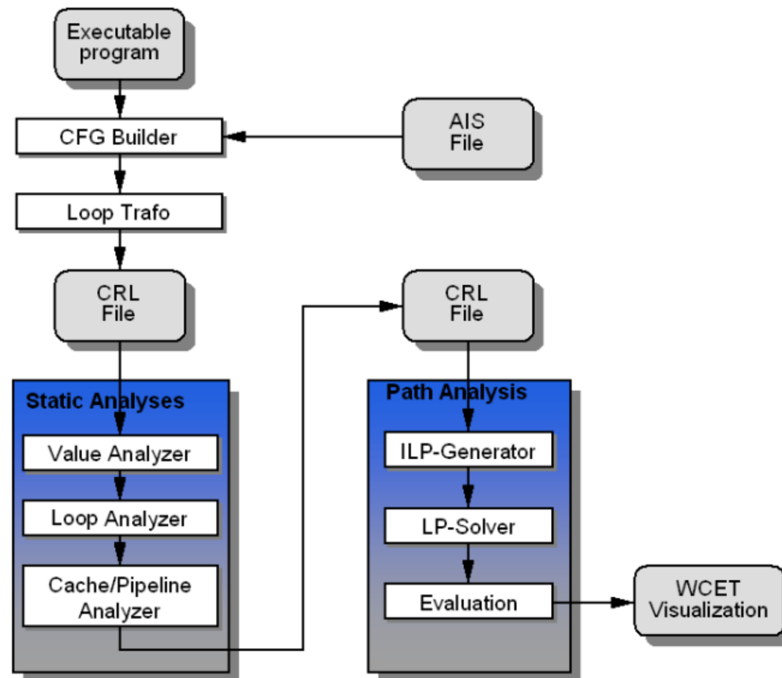

Figure 2.2: Phases of computation of Abint aiT tool [2].

Tool	Flow facts extraction	Bound calculation
aiT	Static analysis	IPET
Chronos	Static analysis	IPET
Heptane	Static analysis	IPET
Otowa	static analysis	IPET
SWET	Static analysis	IPET
PTA	...	path-based

Table 2.1: WCET tools comparison.

- Otowa [19]: extensible tool, which enables the implementation of new static analyses for WCET computation;
- SWEET [20]: uses program path analysis to compute not just the WCET but also BCET;
- PTA [21]: a timing analyzer which aims to analyze straight-line code (without loops and recursive function calls).

The tools present below share almost the sames techniques, as we can see in the table 2.1. In this table we can notice that almost all tools implements the IPET. This fact can be justified by the fact that the IPET method allows dealing with more complex program flow.

2.3 WCEC Analysis

Many characteristics of WCEC analysis are inherited from WCET analysis. The similarities can be noticed by comparing the computation phases of Absint aiT (figure 2.2) and

Absint EnergyAnalyzer [3] (figure 2.3), a WCEC analysis tool. Since the analyses are very similar, in this section, we will focus on the differences between these two analyzes and the additions of WCEC analysis. Besides WCEC and WCET analysis being similar in their approach, in many cases, execution time and energy consumption are unrelated [22].

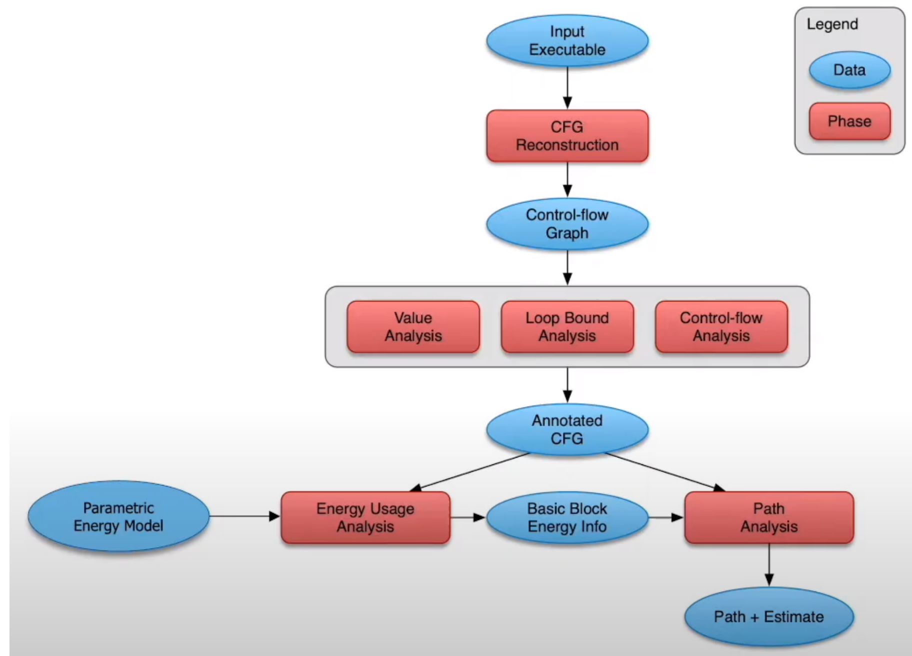

Figure 2.3: Phases of computation of EnergyAnalyzer tool [3].

Most of the works around the WCEC are addressed to battery-operated embedded systems, mainly for those that become useless or cause problems when they run out of battery. Energy is a restricted resource in many embedded real-time systems, but it is essential. Therefore, a computation of a metric similar to WCET but energy aware instead of time (the WCEC) is also needed in task scheduling to ensure that the device will meet the energy constrain [22]. Battery technology is not advancing fast enough to meet the rising performance demands of mobile embedded systems [23], situations that the smartphone industry is also facing.

The research of WCEC analysis is at a relatively early stage. Besides this there are some works around this topic. In [23], Chang *et al.* propose a static analysis approach for estimating a task's WCEC of microarchitectures. In contrast to [24], this work attempts to determine the upper bound (worst-case) rather than relying on the developer's specification. This approach first computes the WCEC of each basic block and then utilizes the results to determine the WCEC of the entire program. Charalampos *et al.* [25] take a similar approach, focusing on basic blocks, with the app's overall energy consumption determined by a combination of these blocks' results.

Some works focus on contexts beyond embedded systems. For example, Couto *et al.* [26] address a WCEC technique for software product lines[3]. Their technique integrates static analysis and well-known techniques used in WCEC analysis to analyze all products in a software product line.

[3]Collection of software systems designed to address the needs of a specific market or task [27].

Few works focus on energy consumption analysis for mobile applications. Daniel *et al.* [28] propose a static analysis approach to optimize the energy efficiency of graphics-intensive apps, such as games, which claimed to save up to 44% of device energy consumption. This work concentrates on Graphics Processing Unit (GPU) operations, with the rationale that most graphics-intensive apps use GPU rather than CPU to draw graphics quicker. The paper presents a set of software analyses focused on removing energy bugs and reducing needless GPU usage. One problem with this approach is that it allows false positives and negatives, leaving it up to the user to determine what might be real.

Computing a WCET or WCEC analysis requires an element that relates the software's effects on the hardware; this element is known as the cost model. In the next section, we explain the cost models' general ideas and which suite is better for us.

2.4 Cost Model

A cost model is essential parameter in every WCET and WCEC analysis. These models aim at a specific target device since the energy/time spent by the program relies strongly on the hardware and software of the device executing it. However, creating models is time-consuming and error-prone, and the target device is required, making it a complex process to scale and automate for new devices.

For wort-case analyses is required an instruction-level model that relates the aimed resource (energy/time) to instructions. The measurement of these instructions in the embedded system domain is performed on assembly instructions since the lower the level, the more accurate the results.

To create an instruction-level energy model is necessary to measure the energy consumed by the execution of the instructions. The works found about instruction-level energy models are focused on embedded systems. [29] [30] are works that describe methods to the creation of instruction-level energy models for microcontrollers[4]. These works are very similar to the approach used to measure the energy per instruction. The employed approach is simple and boils down to iterating the instruction and calculating average values. Yakun *et al.* [32] on their work, follow a similar approach, but on more complex hardware, the Intels Xeon Phi processor. The Volkmar *et al.* [33] follow a different approach to measure the instructions energy cost. In their approach, they ran a set of benchmarks B on the target platform and its measured the energy consumption (E_b) by each benchmark b. To calculate the energy per instruction (EPI), they used the following equation system:

$$\begin{bmatrix} IC_{1,1} & IC_{1,2} & ... & IC_{1,n} \\ IC_{2,1} & IC_{2,2} & ... & IC_{2,n} \\ . & . & . & . \\ . & . & . & . \\ . & . & . & . \\ IC_{m,1} & IC_{m,2} & ... & IC_{m,n} \end{bmatrix} \begin{bmatrix} EPI_1 \\ EPI_2 \\ . \\ . \\ . \\ EPI_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ . \\ . \\ . \\ E_m \end{bmatrix} \quad (2.1)$$

[4]Small integrated circuit that controls a single function in an embedded system [31].

where the $IC_{b,i}$, the instruction count of the instruction i for the benchmark b, n the number of instructions, and m the number of benchmarks.

There are not many works that aim at smartphone modeling. The works found are focused on smartphone modeling at the hardware level, analyzing component by component. A hardware-level model is typically a formula that represents the relationship between the different components associated with system variables, considering the energy they consume, as the model described in [34]. Usually, the smartphone components considered in these models are the CPU, WiFi interface, GPS, Audio interface, and display. Commonly, the method used to build hardware-level consists of the measurement of the energy consumption by the smartphone with components in different states, then establishing a relationship, for example, with linear regression. There are some works that present tools to build these models automatically. In [35] [34] are presenting tools that use similar approaches to create models. These tools are smartphone applications that use built-in sensors to monitor power consumption while running test scenarios on the smartphone, with components in varying states. The data gathered is posterior used in the model elaboration.

The model creation is a task that must be a priority and taken with the most rigorously as possible since the soundness of the analysis relies strongly on the model accuracy [33]. Therefore, the proper functioning of the analysis phases depends on the model. For a good understanding of WCET and WCEC analysis and how the model can affect the analysis, in the following sections, we will explain two typical phases applied in both analyses, static analysis and bound calculation.

2.5 Static Analysis

Static analysis is a technique that allows extracting facts from programs without executing them. This technique has been used for program optimization, security, and safety analysis and recently successfully predicted the program's time bounds. The static analysis as a phase of WCET / WCEC analysis comprises a bunch of analyses to extract relevant information, as can be seen in figure 2.2.

Static analysis received more attention in the computer science community when Patrick and Radhia Cousot introduced the theory of safe approximation named abstract interpretation in 1977, presenting a formal mathematical approach to static analysis [36]. The abstract interpretation allows the exploration of all possible ramifications of the program's flow of control; moreover, it describes how each part of the program can be combined to produce an approximate behavior for the entire program [36]. Thus, static analysis is limited to approximations of program behavior. The approximation must be suited to the addressed problem. In some domains, such as the critical system's domain, the approximation must be the safest as feasible.

Static analysis allows going through all possible paths. It means it considers paths regardless of whether they are executed during runtime. Because of this propriety, static analysis allows in-depth analysis and produces more accurate results.

Static analysis has several practical applications. With static analysis, given specifications, it is possible to prove that no software behavior violates them (formal verification). Furthermore, it is possible to find paths that lead to unwanted behavior (bug-searching). Also, static analysis techniques are commonly implemented on compilers to perform automatic optimizations and on Integrated Development Environment (IDE) to code issues identifying assistance.

Static analysis allows for finding many bugs in the source code, some of which cannot be found with tests. For example, some of the bugs that can be found through static analysis are uninitialized variables, illegal operations (such as division by zero and overflow/underflow in arithmetic expressions), resource leaks, and buffer overflow.

In order to understand how static analysis works, consider that we want to perform a sign analysis. This analysis aims to determine the possible sign for each variable in a given program. An important ingredient for all static analysis is an abstract domain that, in short, defines the groups in which the values could be categorized. For the sign analysis, we have the abstract sign domain, represented as lattice[5] in figure 4.2. This lattice have as elements (-), (+), 0, $\top$ and $\bot$. Where $\top$ represent a set of (-), (+), or/and 0. The $\bot$ indicates the empty set. Then, the behavior of the programs (semantics) is written in our target language. The semantic is used as a guide for the static analysis design. Now, we need to implement an abstract interpreter, similar to a concrete interpreter[6]. An abstract interpreter performs an abstract execution, determining the abstract value of each variable. In the following example, we compare the abstract execution with the concrete execution. The last abstract state of each variable in abstract execution makes up the result of the analysis.

	Concrete execution	Abstract execution
	$\{x \to \mathbb{Z}, y \to \mathbb{Z}\}$	$\{x \to \top, y \to \top\}$
x = 5	$\{x \to \{5\}, y \to \mathbb{Z}\}$	$\{x \to \{+\}, y \to \top\}$
y = 17	$\{x \to \{5\}, y \to \{17\}\}$	$\{x \to \{+\}, y \to \{+\}\}$
x = x * (-5)	$\{x \to \{-25\}, y \to \{17\}\}$	$\{x \to \{-\}, y \to \{+\}\}$
y = x * (-3)	$\{x \to \{-25\}, y \to \{51\}\}$	$\{x \to \{-\}, y \to \{+\}\}$

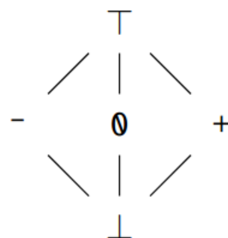

Figure 2.4: Signs abstract domain.

Based on the above example, we can intuitively state that a static analyzer is an abstract interpreter, which the function can be defined as

[5] A partially ordered set with a least upper bound and a greatest lower bound for each two-element subset [37].

[6] A program that executes instructions written in a programming or scripting language without compiling them previously. Being concrete means that the values resulting from the execution of this interpreter are the actual ones.

$$analyzer(P, \alpha_{pre}) = \alpha_{pos} \quad (2.2)$$

where P is a program, α_{pre} is the initial abstract, and α_{pos} is the final abstract state.

The sign analysis is a simple one but can have real applications. In addition to signal analysis, the other most common static analysis are:

- Live variables analysis: Try to determine at a certain program point if the value of a certain variable could be read further in the program execution.
- Constant propagation analysis: Determine which variables have constant values for each program point.
- Very busy Expressions analysis: Check which expressions are evaluated more than once before changing the expression value.

There is an alternative to performing program analysis, the dynamic analysis, in contrast to the static analysis, requires the program execution on the target hardware of the program. When examining the source code, static analysis is best as it examines all possible execution paths and variable values but at the cost of several approximations. The dynamic analysis is precise by definition but just related to a specific execution scenario [38] since it just examines paths used during execution. Nevertheless, considering all scenarios is necessary to consider all input combinations that sometimes are impractical. Also, dynamic analysis is generally more computationally expensive since it requires running the program. The characteristics of static analysis make it more suitable for WCEC analysis. However, the analysis must be sound [7] to guarantee that found result is never an underestimation. Typically, a static analysis approach is subject to the following pattern [39]:

- Be automatic;
- Produce reliable (sound) results;
- Generally, incomplete because they are incapable of representing all program properties.

A typical static analysis used in the context of WCEC (consequently WCET) is the value analysis, which attempts to identify the values stored in each variable for each program point. It usually cannot identify these values precisely but only determines safe lower and upper bounds[2]. These bounds are used for further analysis, such as loop bound analysis, as the name says, consists of determining the bounds of loops. It is not always possible to determine the loop bounds. In the case of aiT, the unknown loop bounds depend on the annotations provided by the user [2].

[7] Property that conditions the analysis in case of uncertainty always results in the most cautious result possible.

2.6 Bound Calculation

As the name suggests, the bound calculation, also called path analysis, is the phase where the energy (upper) bound is calculated. Two main methods to estimate the worst-case value are path-based and IPET. The path-based method uses well-known graph algorithms to search in the CFG for the path that leads to the longest execution time. This search can be exhaustive since the control flow in a complicated problem might be extensive even if the program stops.

The IPET method boils down to the transformation of a CFG to an ILP problem, a type of optimization problem consisting of a set of variables constraints and an objective function [40]. The worst-case result is the maximization of the objective function 2.3, where x_i is the number of times the block/node B_i is executed, c_i is the energy cost of the block, and N is the number of existing blocks. The value of each x_i is found in solving the equations, and the c_i is determined by calculating each block energy. The system of equations is solved using an ILP solver specifically designed to handle this type of problem. There are several different ILP solvers available to choose from.

Based in the objective function formulation from [41], we have:

$$\sum_{i}^{N} c_i * x_i \quad (2.3)$$

The rest of the equations are derived considering that each x_i is equal to the number of times control enters the node (inflow) and equal to how many times the control leaves the node (outflow) [41]. The figure 2.5 shows an example of a translation of a CFG to an equations system using this technique.

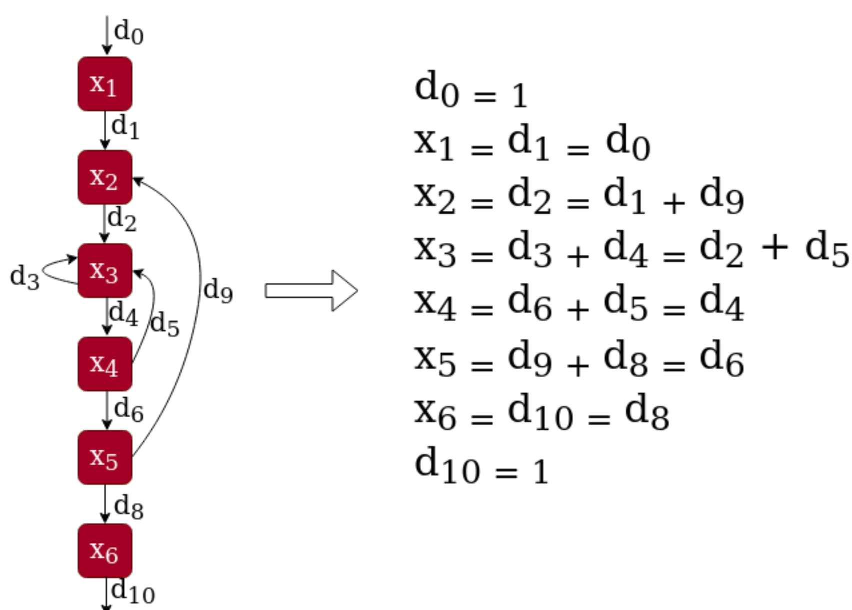

Figure 2.5: Example of a translation of a CFG to equations system.

Usually, in WCEC/WCET analysis approaches, use IPET to perform the bound calculation instead path-method. IPET has the advantage since it can handle more complex graphs, unlike path-based, which can only handle well-structured graphs [13]. We just found one tool that uses the path-based method, the PTA. Besides using the path-based method, the PTA delivers what it promises since it just works with simple programs (without loops on theirs CFG), where the path-based method is the optimal option.

2.7 Conclusions

An important conclusion here is that most of the WCET analysis tools currently on the market use the same techniques, such as static analyis and IPET, and as expected, the WCEC tools follow the same role. In fact, WCEC is similar to WCET in differences aspects. The main difference between both is that the WCET analysis focuses on operations that significantly impact the CPU. In contrast, the WCEC analysis focuses on those that impact the battery. Aside from the fact that WCEC and WCET analyses are similar in concept, execution time and energy consumption are frequently unrelated.

Chapter
3

Proposed Energy Cost Model

In this chapter, we explain the approach used to create a simple model that relates energy cost to each instruction of a set of instructions; this type of model is known as the instruction-level energy model. The model is created for a specific target device since the energy (and time) spent by the instructions depends on the device's hardware and software. Therefore, in section 3.1, we describe some relevant specifications of our target device to understand its behavior better. Also, we describe some actions we applied in the device that reduces discharging energy rate, consequently the noise in the analysis. In section 3.2, we explain the approach we use to calculate the energy consumed by the instructions. In section 3.3, we explain how all information obtained by instruction level analysis is used to compose a model.

3.1 Target Device Overview

In this project, we use as the target device a modern smartphone, the Xiaomi Redmi Note 9 Pro (XRN9), a smartphone released in 2020. This smartphone was used because it has a hardware architecture and a system that represent the typical smartphone nowadays. This smartphone has the main specifications: 6 GB of ram and 64 GB of internal storage, Snapdragon 720G CPU (Octa-core), Android 11th version, and IPS LCD display.

In a smartphone, the component that most affects the battery life is the display [4]. In the case of the XRN9, we are dealing with an IPS LCD display, which generally consumes more energy than the other display options, such as OLED [42]. Another essential element to consider is the device system because it plays a crucial role in power management since it dictates how the hardware resources will be used. The XRN9 uses Android, a system continuously being improved with new features to reduce the consumption of the device.

Modern phones have multiple factors that increase the discharging rate of the battery, such as processes running in the background. A feature found on XRN9 and other modern smartphones help us to reduce the noise in our measurement process, the ultra battery saver mode. As described in the XRN9, this mode implies dark theme, background activity restriction and restriction of power-consuming activity (e.g., GPS, vibration, always-on display).

In the ultra battery saver mode, we can chose witch can run without restriction. Therefore we select our app to run normally. In addition, we applied other actions to reduce the noise in our measure, such as reducing to minimum the brightness level, taking the SIM card to avoid any mobile telephony services, and ensuring that there is no background process by set the background process option's limit to "no background process".

3.2 Instruction Level Analysis

In this section, we explain how we could calculate the power of each instruction and the time it spends. Getting the exact power/time cost per instruction is hard, and it is not feasible for most complex architecture, such as the existing one in modern android smartphones. Therefore, we used a model composed of approximated values that characterize the energy consumption of instructions relative to another[22], known as a relative energy model. Typically, the precise model just can be provided by the device manufacturer. However, even this information may be inaccurate [33].

In order to create a new energy model, we need to measure the energy consumed by each instruction i of the set of instructions I. We represent this as EPI_i (Energy per instruction i). The *EPI* is given by:

$$EPI = P * T \quad (3.1)$$

where P is the average power a device uses to perform a certain instruction, and T represents the instruction's execution time. For the power, we use the instantaneous battery current since it can represent the spent at a given time. To measure the T and P, we use a method explained in [33] [29], which consists of executing a micro-benchmark for each instruction and calculating the energy and time used by it. However, as the values of the T and P are very tiny is impossible to measure these values alone. To handle this, a simple approach is execute multiple instances of the instruction i multiple times [33] [29].

We built a simple app that repeatedly executes fifty instances of the same instruction. In every 100 milliseconds, the instantaneous battery current in microamperes is gathered. Also is measured the time to execute these fifty instructions. To obtain the instantaneous battery current, we use a system-level service, the battery service, that allows querying the battery and charging (and discharging) properties. To obtain the most reliable values possible, it is necessary to analyze each instruction under the same conditions. Thus, we perform an analysis of different instructions with the settings explained in the section 3.1.

3.3 Instruction Level Model

In this work we do not seek to give exact instruction's energy consumption in our model since it is impractical; instead, we provide abstract values that may be used to compare different instructions regarding energy consumption. To create a new model, we apply the approach explained in section 3.2 for a set of instructions. As shown in table 3.1, our

Instructions	**Power**	**Time**
getUserStyle()	4.5575	0.0095
getAppTaskThumbnailSize()	8.5036	0.5815
isRequestPinAppWidgetSupported()	62.309	58.1663
getConcurrentCameraIds()	3.7969	0.0065
hasVibrator()	9.3821	0.1403
stopBluetoothSco()	10.7528	0.3121
getConcurrentCameraIds()	3.7969	0.0065
hasVibrator()	9.3821	0.1403
getAllProviders()	9.2062	0.0951

Table 3.1: Simplified part of the model.

model consists of a collection of instructions with its respective execution time T and Power P, stored in a TXT file.

In an Android app's development, a large number of instructions types can be used, making infeasible to test all of them. In our test, we found that there are many types of simple instructions, such as arithmetic, that their execution does not significantly drain the battery energy. In contrast, we have noticed that many instructions that involved method call drain significantly the battery energy. We pick a set of instructions with a method call because they have a more significant energy expenditure. To be more specific, we test methods of the system services [1], because they show more expressive energy cost.

3.4 Conclusion

Determining the exact energy (power and time) consumed by an instruction execution in a smartphone is not feasible since it is a very complex platform. Nevertheless, it is possible to calculate a metric that can be used to compare the energy spent by instruction to another. This type of model is know as relative energy model.

[1] Services that allow communication with the indicated hardware [43].

Chapter 4

Implementation Details

This chapter describes how we implemented our technique in a prototype for Android apps WCEC analysis. To give a better spectrum of our prototype functioning, in section 4.1, we explain the phases of our analysis. Following, we highlight the main phases of our analysis in the following order: inputs preparation in section 4.2, static analysis in section 4.3, and bound calculation in section 4.4.

4.1 Phases of The Proposed Technique

Our technique consists of multiple phases. The phases are designed based on the principles discussed in chapter 2. The different phases of our technique are represented in figure 4.1. Each phase represents a module, making our prototype modular. Being modular means that each module can be modified independently. Also, it allows flexibility in the development process and enables more straightforward improvements for each module without affecting the overall functioning of the prototype. In the next section, we describe the implementation details of the main modules.

To perform an analysis, our prototype requires two inputs: an Android app in APK format and an instruction-level cost model. The prototype takes an Android app in APK format as input, extracts relevant data, and performs static analysis to infer loop bounds. If the loop bound analysis is successful, the loop bound is annotated to the data in the analysis. Otherwise, the loop bound is annotated with a value provided by the user. The prototype then uses the annotated data to calculate the energy per method. Finally, it uses all relevant data to perform a bound calculation or path analysis, producing the WCEC value as output.

4.2 Data Preparation

The data preparation is the starting point of our analysis. The output of this phase is the CFG and the files representing the app methods implementation, which will be analyzed in the static analysis phase. To perform this phase, we write a shell script that makes it easy to carry out specific tasks we desire, such as file manipulation and executing different programs.

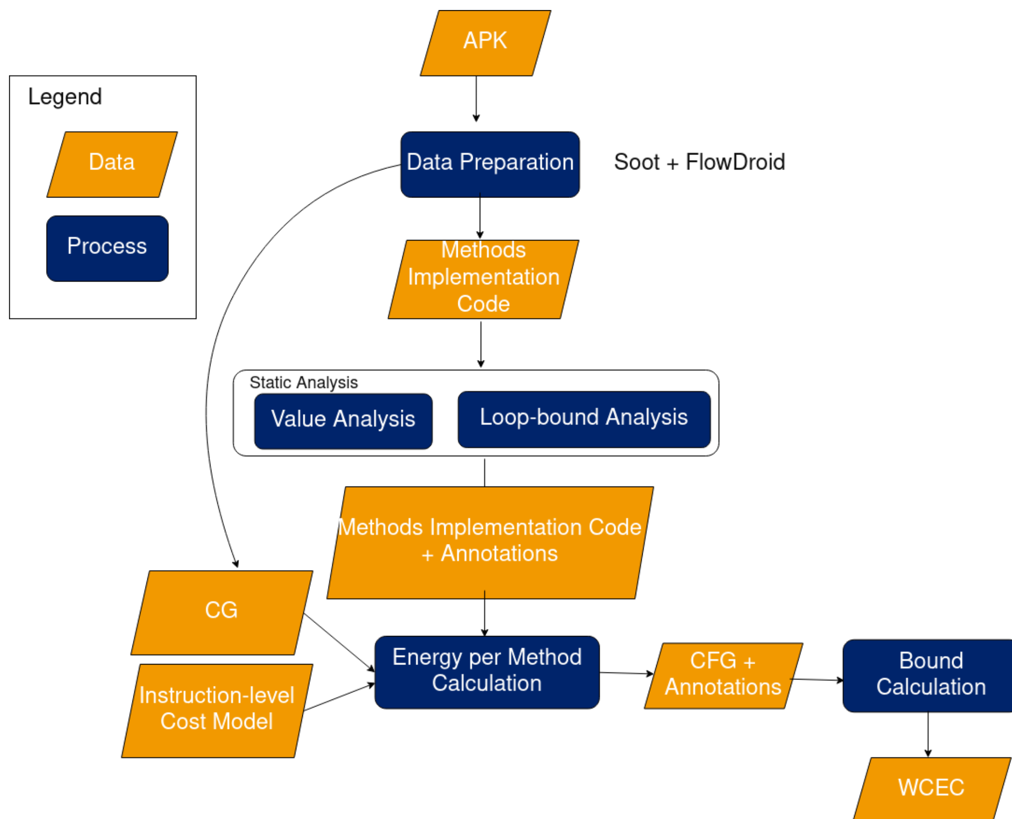

Figure 4.1: Phases of our WCEC technique.

The files that represent the implementation of the methods are extracted from the app using Soot, an open-source framework for Java and Android apps analysis[44]. We use the Soot as a stand-alone tool, executing it in the data preparation script; we provide the path of the app. The Soot comes with some intermediate languages, the main one is Jimple, a language designed to facilitate the analysis of Java programs and android apps [45].

To be more precise, we produce a Call Graph (CG), a CFG that represents the calling relationships between its methods at runtime. To construct CFG's, we took advantage of FlowDroid, an open-source tool built on top of Soot that allows static taint analysis of Android apps [46]. To generate the call graph, the FlowDroid uses the Soot Pointer Analysis Research Kit (SPARK), a framework capable of building CFGs, using points-to analysis [47], a static analysis that determines information on the values of pointer variables or expressions [48].

In general, constructing a CFG requires the specification of an entry point. In languages such as C, a program has the primary function as the entry point, but Java programs/Android apps are more complex. Since there are many potential entry points, such as static initializers and finalizers, the problem is significantly more difficult[47]. The SPARK tries to find a suitable entry point by modeling the app execution. Also, it is essential to refer that the FlowDroid build CFGs with reachable methods; it performs an analysis to identify the potential runtime methods that could be accessible. To use FlowDroid, we implemented a java class, setting our desired settings. Ultimately, we need to provide the path of the target APK. Some apps have many nodes, consequently a large CFG. To reduce the size of the CFG, we set the FlowDroid to ignore some Android packages that

are automatically added, such as the Java package.

4.3 Static Analysis

The static analysis in this context extracts relevant facts from the program before performing the path analysis. We use the static analysis to perform a value analysis to calculate the loops' bounds. We wrote the static analysis phase, and therefore the core of our tool, in Ocaml because it allows us to express type definitions and operations on abstract syntax trees succinctly and straightforwardly [39].

Value Analysis

Value analysis aims to identify the value of all program variables. Usually, the commonly used approach is to determine the interval the variable value may be. However, in many cases is not possible to determine the exact variable values. Furthermore, the analysis result is frequently exaggerated. However, as with existing works, we overcome this limitation by asking the user to provide a bound for the loops. Thus, in this analysis, we try to find reasonable results in as many cases as possible, but in case we are unable to use the user loop specification. The value analysis outcomes are employed in the loop-bound analysis. As stated in chapter 2, to carry out a static analysis is mandatory to specify an abstract domain. In our value analysis, we use the abstract interval domain, in which the elements are represented in I set, where I is defined as

$$\{[a,b] | a,b \in \mathbb{Z} \cup \{-\infty,\infty\}\} \cup \{\perp\}$$

Then, after we understood the Jimple program's semantics, we could implement an abstract interpreter.

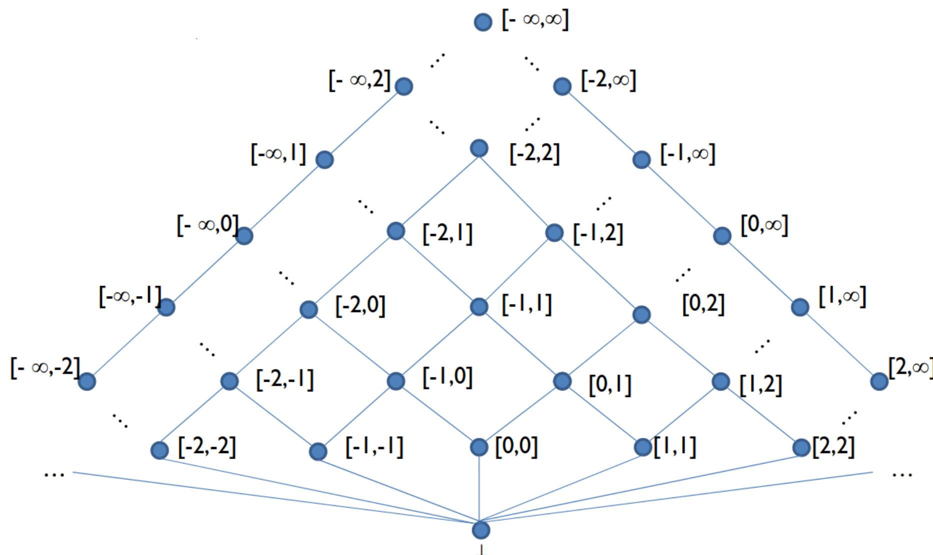

Figure 4.2: Interval abstract domain.

Loop-Bound Analysis

Knowing the loop bounds is very important for a WCEC analysis because a large part of the energy expenditure is attributed to the repetition of the instruction in the loops. We

can proceed to the loop bound analysis with the value analysis results. To perform the loop bound analysis, we apply the method introduced in [49], defined as a technique that uses program slicing to determine the loop bounds in addiction value analysis. Program slicing consists in eliminating/ignoring some specific parts of the code. In this context, it s used to eliminate parts that do not affect the loop termination. Thus, we will analyze the code parts that affect the loop behavior in the loop bound determination. To a better understanding, we will explain how this technique works for the following code example:

```
int i = 0
int j = 10
while (i <= j) {
    i = i + 2
}
```

By the result of the value analysis, we know that after the loop, the value of i will be in the interval [0,10]. Also, we can see that the increment factor (inc) of i is two because, in each interaction, two is added to i. Knowing that to compute the loop bound, we need to determine the cardinality[1] of the set A, where A is defined as

$$\{0,..,10\} \cap \{x \mid x \in \mathbb{Z} \text{ and } x \% \text{ inc} = 0\}$$

In the example, the cardinality of A is 6.

As explained in the chapter 2, it is expected that the static analysis approaches do not always find good answers. In our case, this statement means that there are many cases where our tool will not be capable of determining a realistic bound for the loop. Thus, like many other WCEC and WCET, we allow the user to choose the bound of the loops, which our tool can give realistic bounds.

4.4 Bound calculation

As is expected, the bound calculation is the final phase of our WCEC analysis. In this phase, we already have what we need to determine the upper bound energy of the app, an annotated CFG. This annotated CFG has information about the relation between the app methods and the energy expenditure caused by each. As described in the chapter 2, the primary methods to perform the bound calculation are the path-based method and IPET. In order to compare the two methods, we apply the two methods. Following, we present our thoughts about the two methods.

Path-based Method

Considering our CFG a weighted graph G, where V is the set of the vertices and E is the set of edges, each with a weight that is the energy cost to execute the destination node. This method consists of the resolution of the graph's longest (heaviest) path problem. To solve the longest path problem, we follow the approach explained in [50], allowing solving it at a

[1] Property that represents the size of a set

linear time (O(V + E)). This approach consists of the transformation longest path problem into the shortest path problem by inverting the signal of the weights (-G).

To solve the shortest path problem, we first order the vertices using the topological sort to arrange them so that all of their edges point from a vertex earlier to a vertex later in the order. Each vertex will be linked to an enormous value (in theory, this value can be infinity) that will be updated as the weight to reach it is better (smaller) than the actual weight. An iterative process will be carried out, traversing the entire graph, starting at each node of the order and passing through all reachable vertices. In the end, we will use the topological order and the total weight to reach any vertex. Now we can multiply all values by -1 and take the biggest one, which is the worst-case since this is the biggest total weight to reach. We can also take the worst-case path by taking steps back from the vertex with the biggest weight, passing by the vertex's biggest weights to reach.

Despite being a high-speed method, it is not very useful for our context since it just works if the graph is a Direct Acyclic Graph (DAG)[2], but it is usual for a graph from a call graph of an Android app to have loops. These loops in the graph can result from various factors, such as recursive functions.

IPET

As explained in the chapter 2, this method consist in elaboration a ILP problem from graph. In practice, we create a new graph, replacing the instructions' names with sample names and assigning a name for each edge. Now we apply straightforwardly the technique explained in [41]. With the ILP equations formed, we write them into a file, which will serve as input of an ilp ILP solver. To solve the ILP equations, the lp_solve is used, a well-known free ILP solver.

4.5 Conclusion

Our tool structure is modular, meaning any component can be improved without making too many changes to other components. The methods used in part of the projects are commonly used in WCET WCEC analysis, making it easier to implement our tool since we can reuse the ideas used in the other tools. The part we need to recreate more is the static analysis since this part depends strongly on the language analyzed.

[2]A graph with no directed cycles.

Chapter 5

Technique Evaluation

In this chapter, we conducted experiments with our prototype to highlight the efficiency of our technique. We present a scenario in which we want to compare different energy consumption apps to rank them. First, section 5.1 outlines the main components employed in our experiments, such as the created apps. Then, in section 5.2, we show the results of our experiments and discuss them in detail.

5.1 Tests Setup

To run the WCEC tool is necessary a computer with the required dependencies. In addition to the Ocaml and some necessary modules, it is required to have the lp_solve to solve the ILP generated problem, and the Dune, an open-source build system for Ocaml projects [51]. To run our tests successfully, we use a portable computer equipped with an Intel Core I7 (8th generation) processor and 8GB of RAM. For the sake of the current tests, we built four simple Android apps in Java with different features, all with layouts represented in figure 5.1. The intent of running tests with simple apps that share the same structure is that it is possible to easily predict the results and compare them with the actual results produced by the tool. The apps created for the tests are:

- **App1** : We consider that app as our base case. That app is an empty app that does not perform any special task ;
- **App2**: This uses the audio system service[1] to play audio when the start button is clicked and stop when the stop button is clicked;
- **App3**: In contrast with the App2, this app uses the vibrator system service to vibrate the device when is the start button clicked;
- **App4**: This app joins the features of App2 and App3. Thus, play audio and vibrate the device when clicking the start button.

[1] A system service that allows accessing to the audio Hardware interface.

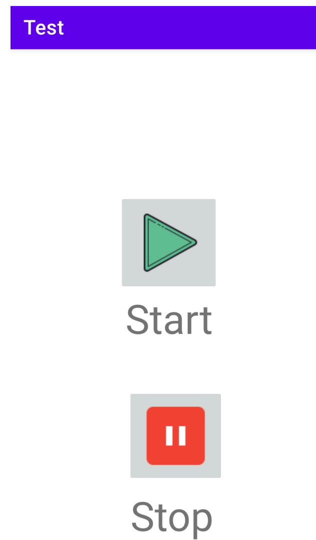

Figure 5.1: Test App layout.

5.2 Results

Here, the results of our analysis will be presented. The table 5.1 relates the results of WCEC analysis for each test app with the total energy consumption of the known (included in the model) instructions. Figure 5.2 explicitly compares the WCEC results of the test app, presenting as expected the App4 with the highest worst-case execution consumption and App1 with the lower.

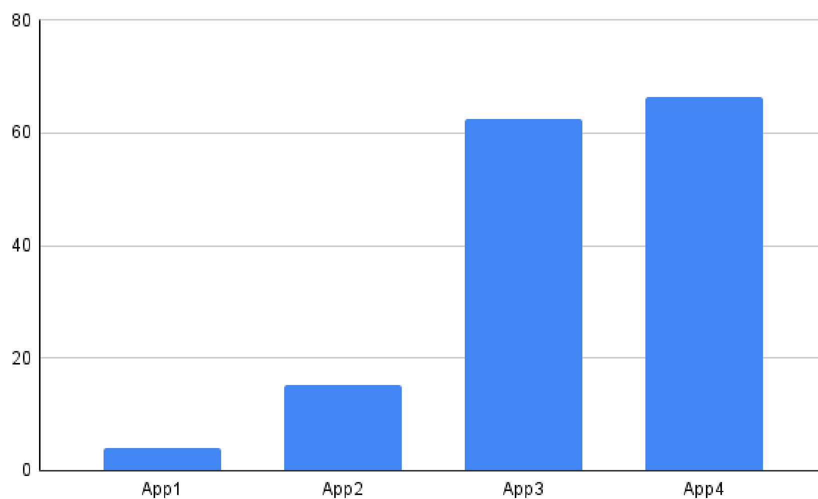

Figure 5.2: Comparison between the relative energy of different apps.

5.3 Conclusions

This chapter presents the results of simple tests performed on our tool. The results of our analysis can be used to compare different apps. Providing precise results requires an

App	WCEC	Total energy by used instructions
App1	4.000	
App2	15.143	3.0943
App3	62.309	58.1663
App4	66.403	61.260

Table 5.1: Results of the tests and the corresponding total energy consumption of instruction included in the model.

accurate cost model. However, our cost model is not composed by exacts measurement values; thus, we do not guarantee precise results. However, creating a model with exacts for a very complex architecture is not feasible. Although the tool does not produce exact results, we show a scenario in which our tool can be helpful (for app comparisons).

Chapter
6

Conclusions And Future Work

Our primary goal in this work was to develop a tool to perform WCEC analysis on Android apps using static analysis techniques. This document describes the fundamentals and the decisions made to achieve this goal. We prioritize model generation because the soundness of the analysis relies on model correctness. It is not feasible to derive a model with precise values for a complex platform such as the android smartphone. Besides, it is possible to generate a relative model, a model with abstract values that represent the difference in instruction's energy consumption to others. It is not feasible to ensure accurate boundaries with a relative model, but it is still useful to compare programs based on their energy consumption.

We follow a similar approach to the existing works, but different from them, we aim to analyze Android apps instead of simple embedded system programs. In our tool, we use static analysis and the IPET, the common methods used in the WCEC and WCET analysis. To show what we can do with our tool, we evaluate it by comparing different apps.

6.1 Future Work

Analyzing Android apps to determine whether WCEC is not a trivial task since Android smartphones and apps are very complex. Although our meaningful achievement, we know that our WCEC tool needs considerable effort to reach our expectations. The main tasks that could be performed to improve our WCEC tool are:

- Extension and refinement of the energy model. There are a large number of instructions that need to be tested and included in the power model. Thus, it could be beneficial to understand which sets of instructions have a higher energy expenditure than average and need attention. Also, the accuracy of the analysis relies directly on the model. Thus, improving the model leads to an increase in accuracy;
- Improve the value analysis. In the worst case, we can guarantee that our value analysis provides an exaggerated over-approximation of the actual values. The worst case is reached when the analysis cannot find a concrete result. Thus, the analysis needs refinements to find concrete results most of the time;

- Study static analysis that could be useful in our analysis. In general, WCEC analyzers have the static analysis phase, extracting facts about the code without executing the program. This phase is essential to extract as much information as possible to provide more accurate results. Thus, there is an opportunity to improve the accuracy of the analysis by performing a new static analysis.

Bibliography

[1] Configure your build. [Online] https://developer.android.com/studio/build. accessed: 2022-09-30.

[2] Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. 2004.

[3] AbsInt Angewandte Informatik GmbH. Report on early prototype of energyanalyser d4.2, 2018.

[4] Joe Hindy. A definitive guide to everything that affects smartphone battery life. [Online] https://www.androidauthority.com/smartphone-battery-life-drain-causes-1071423/. accessed: 2022-09-05.

[5] Sarah Perez. Mobile users are now spending 4-5 hours per day in apps. [Online] https://techcrunch.com/2022/08/03/mobile-users-now-spend-4-5-hours-per-day-in-apps-report-says/. accessed: 2022-09-02.

[6] Mobile Operating System Market Share Worldwide. [Online] https://gs.statcounter.com/os-market-share/mobile/worldwide. accessed: 2022-09-27.

[7] Aimee Ross and Lorna Christie. Energy consumption of ict, 2022.

[8] Application Fundamentals. [Online] https://developer.android.com/guide/components/fundamentals. accessed: 2022-09-17.

[9] Android - Application Components. [Online] https://www.tutorialspoint.com/android/android_application_components.htm. accessed: 2022-09-17.

[10] P. Puschner. A tool for high-level language analysis of worst-case execution times. In *Proceeding. 10th EUROMICRO Workshop on Real-Time Systems (Cat. No.98EX168)*, pages 130–137, 1998.

[11] Andreas Ermedahl and Peter Puschner. Preface to the special issue on worst-case execution-time analysis. *Journal of Systems Architecture*, 57(7):675–676, 2011. Special Issue on Worst-Case Execution-Time Analysis.

[12] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *WCET*, 2006.

[13] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenstrom. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst.*, 7, 01 2008.

[14] C Ferdinand, R Heckmann, M Jersak, F. Martin, and K Richter. Integrating System-Level and Code-Level Timing Analysis for Dependable System Development. In *Embedded Real Time Software and Systems (ERTS2008)*, toulouse, France, January 2008.

[15] Chronos. [Online] `https://www.comp.nus.edu.sg/~rpembed/chronos/` accessed: 2022-02-02.

[16] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos version 2 . 0 user manual. 2007.

[17] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. The Heptane Static Worst-Case Execution Time Estimation Tool. In *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*, volume 8 of *International Workshop on Worst-Case Execution Time Analysis*, page 12, Dubrovnik, Croatia, June 2017.

[18] RISC? [Online] `https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/whatis/index.html`. accessed: 2022-10-05.

[19] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. Otawa: an open toolbox for adaptive wcet analysis. pages 35–46, 10 2010.

[20] Björn Lisper. Sweet-a tool for WCET flow analysis (extended abstract). volume 8803, pages 482–485, 10 2014.

[21] Friedhelm Stappert and Peter Altenbernd. Complete worst-case execution time analysis of straight-line hard real-time programs. *Journal of Systems Architecture*, 46:339–355, 02 2000.

[22] Peter Wägemann, Tobias Distler, Timo Hönig, Heiko Janker, Rüdiger Kapitza, and Wolfgang Schröder-Preikschat. Worst-case energy consumption analysis for energy-constrained embedded systems. In *2015 27th Euromicro Conference on Real-Time Systems*, pages 105–114, 2015.

[23] R. Jayaseelan, T. Mitra, and Xianfeng Li. Estimating the worst-case energy consumption of embedded software. In *12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'06)*, pages 81–90, 2006.

[24] Francisco Bueno Carrillo. A framework for verification and debugging of resource usage properties. In *ICLP 2010*, 2010.

[25] Charalampos Marantos, Konstantinos Salapas, Lazaros Papadopoulos, and Dimitrios J. Soudris. A flexible tool for estimating applications performance and energy consumption through static analysis. *SN Comput. Sci.*, 2:21, 2021.

[26] Marco Couto, Paulo Borba, Jácome Cunha, João Paulo Fernandes, Rui Pereira, and João Saraiva. Products go green: Worst-case energy consumption in software product lines. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A*, SPLC '17, page 8493. Association for Computing Machinery, 2017.

[27] Linda Northrop. Software product lines essentials. *Software Engineering Institute Carnegie Mellon University*, 46, 2008.

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

[29] Vivek Tiwari and Mike Tien-Chien Lee. Power analysis of a 32-bit embedded microcontroller. In *Proceedings of the 1995 Asia and South Pacific Design Automation Conference*, ASP-DAC '95, page 23es, New York, NY, USA, 1995. Association for Computing Machinery.

[30] C. Chakrabarti and D. Gaitonde. Instruction level power model of microcontrollers. In *1999 IEEE International Symposium on Circuits and Systems (ISCAS)*, volume 1, pages 76–79 vol.1, 1999.

[31] Ben Lutkevich. microcontroller (MCU). [Online] `https://www.techtarget.com/iotagenda/definition/microcontroller`. accessed: 2022-10-07.

[32] Yakun Sophia Shao and David Brooks. Energy characterization and instruction-level energy model of intel's xeon phi processor. In *International Symposium on Low Power Electronics and Design (ISLPED)*, pages 389–394, 2013.

[33] Volkmar Sieh, Robert Burlacu, Timo Hönig, Heiko Janker, Phillip Raffeck, Peter Wägemann, and Wolfgang Schröder-Preikschat. An end-to-end toolchain: From automated cost modeling to static wcet and wcec analysis. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, pages 158–167, 2017.

[34] Lide Zhang, Birjodh Tiwana, Robert P. Dick, Zhiyun Qian, Z. Morley Mao, Zhaoguang Wang, and Lei Yang. Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *2010 IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 105–114, 2010.

[35] Wonwoo Jung, Chulkoo Kang, Chanmin Yoon, Donwon Kim, and Hojung Cha. Devscope: A nonintrusive and online power analysis tool for smartphone hardware components. CODES+ISSS '12, page 353362, New York, NY, USA, 2012. Association for Computing Machinery.

[36] Patrick Thomson. Static analysis: An introduction: The fundamental challenge of software engineering is one of complexity. *Queue*, 19(4):2941, aug 2021.

[37] Lattice. [Online] `https://encyclopediaofmath.org/wiki/Lattice`. accessed: 2022-10-07.

[38] Dynamic Analysis vs. Static Analysis. [Online] `https://www.intel.com/content/www/us/en/develop/documentation/inspector-user-guide-linux/top/getting-started/dynamic-analysis-vs-static-analysis.html` year = 2021, accessed: 2022-02-02.

[39] Xavier Rival and Kwangkeun Yi. *Introduction to Static Analysis An Abstract Interpretation Perspective*. Addison-Wesley, 2020.

[40] Mark M. Meerschaert. Chapter 3 - computational methods for optimization. In Mark M. Meerschaert, editor, *Mathematical Modeling (Fourth Edition)*, pages 57–112. Academic Press, Boston, fourth edition edition, 2013.

[41] Y.-T.S. Li, S. Malik, and A. Wolfe. Efficient microarchitecture modeling and path analysis for real-time software. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 298–307, 1995.

[42] Kristoffer Bonheur. Comparison: IPS vs. AMOLED, 2019. [Online] `https://www.profolus.com/topics/comparison-ips-vs-amoled-display-technologies/`. accessed: 2022-09-25.

[43] Android Architecture. [Online] `https://source.android.com/docs/core/architecture`. accessed: 2022-09-20.

[44] Patrick Lam, Eric Bodden, Ondvrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. 10 2011.

[45] Eric Bodden. Soot - A framework for analyzing and transforming Java and Android applications. [Online] `https://soot-oss.github.io/soot/`. accessed: 2022-09-15.

[46] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, page 259269, New York, NY, USA, 2014. Association for Computing Machinery.

[47] Ondrej Lhoták and Laurie J. Hendren. Scaling java points-to analysis using spark. In *CC*, 2003.

[48] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, 2(1):1–69, 2015.

[49] Andreas Ermedahl, Christer Sandberg, Jan Gustafsson, Stefan Bygde, and Björn Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. In *WCET*, 2007.

[50] Robert Sedgewick and Kevin Wayne. *Algorithms, 4th Edition*. Addison-Wesley, 2011.

[51] Dune. [Online] `https://dune.build/`. accessed: 2022-09-27.