

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Detecting Resource Leaks on Android with Alpakka

Gustavo Amorim Santos



Mestrado em Engenharia Informática e Computação

Supervisor: Prof. João Carlos Viegas Martins Bispo

Second Supervisor: Prof. Alexandra Sofia Ferreira Mendes

July 12, 2024

Detecting Resource Leaks on Android with Alpakka

Gustavo Amorim Santos

Mestrado em Engenharia Informática e Computação

Approved by:

President: Jácome Miguel Costa da Cunha

Referee: Tiago Diogo Ribeiro de Carvalho

Supervisor: João Carlos Viegas Martins Bispo

July 12, 2024

Resumo

Os dispositivos móveis tornaram-se parte intrínseca do nosso dia a dia, servindo uma multitude de propósitos, desde comunicação e entretenimento, à produtividade e muito mais. Estes dispositivos vieram transformar a forma como as pessoas interagem com o mundo digital, no entanto, sua utilidade está fundamentalmente ligada à duração da sua bateria. Desta forma, as aplicações direcionadas a este tipo de dispositivos devem priorizar a eficiência energética e não consumir bateria desnecessariamente. Em aplicações Android, os *resource leaks* são um culpado comum pelo consumo em excesso de bateria e por uma experiência de utilizador negativa. Estes *resource leaks* surgem de falhas de programação que resultam na alocação e desalocação ineficientes de recursos, incluindo memória, conexões de rede e identificadores de arquivos.

Trabalhos anteriores exploraram a deteção de *resource leaks* e a refatoração automática destes defeitos. A nossa abordagem propõe um compilador *source-to-source* para ficheiros APK para Android, permitindo análise e transformações, como detetar e corrigir *resource leaks*, a ser aplicados em aplicações de uma forma mais sustentável e simplificada. Esta estrutura alinha-se com as necessidades e prioridades de utilizadores finais e programadores, com o objetivo geral de melhorar a qualidade e performance de aplicações Android.

Nesta dissertação, exploramos de forma compreensiva a nossa metodologia para criar o Alpukka, o nosso compilador *source-to-source* para código smali. Ao trabalhar diretamente com código smali, uma representação de mais baixo nível do código de aplicação Android que pode ser descompilado a partir de um APK, tiramos partido da sua compatibilidade inerente com o processo de recompilação para um ficheiro APK, ao contrário de descompilar um APK para código Java. Para além disto, evitamos as complexidades associadas às diferentes linguagens e frameworks utilizados no desenvolvimento de aplicações Android, já que todos permitem compilar para o formato APK.

Adicionalmente, com esta dissertação iremos demonstrar a utilidade prática do Alpukka a melhorar o desenvolvimento de aplicações Android ao expor como pode ser usado para detetar *resource leaks* e aplicar correções automáticas.

Por fim, avaliamos a eficácia da nossa abordagem através de testes e benchmarking numa vasta gama de aplicações Android desenvolvidas com diferentes *frameworks*. Ao identificar, tratar e mitigar *resource leaks*, esperamos contribuir para o desenvolvimento de aplicações Android com maior eficiência energética.

Abstract

Mobile devices have become an integral part of our everyday lives, serving a multitude of purposes, from communication and entertainment to productivity and beyond. These devices have transformed the way people interact with the digital world. However, their utility is fundamentally tied to their battery life. As such, apps targeted for these devices should prioritize energy efficiency and not consume battery unnecessarily. In Android apps, resource leaks are a common culprit behind battery drain and a suboptimal user experience. These resource leaks arise from programming flaws that result in the inefficient allocation and deallocation of resources, including memory, network connections, and file handles.

Previous works have extensively explored the detection of resource leaks and the automatic refactoring of these defects. Our approach proposes a source-to-source compiler for Android APK files, allowing for analysis and transformations, like detecting and refactoring resource leaks, to be applied to applications in a maintainable and more streamlined manner. This framework aligns with the needs and priorities of both end-users and developers, with the overarching goal of enhancing the quality and performance of Android applications.

In this dissertation, we comprehensively explore our methodology for building Alpakka, our source-to-source compiler designed for smali code. By working directly with smali code, a lower-level representation of Android application code that can be decompiled from an APK, we are taking advantage of its inherent compatibility with the APK recompilation process, contrary to decompiling an APK into Java code. Additionally, we circumvent the complexities associated with different programming languages and frameworks used in Android development, as they all ultimately compile to an APK format.

Furthermore, this dissertation will demonstrate the practical applications of Alpakka in improving Android application development by showcasing how it can be used to detect resource leaks and apply automatic corrections.

Lastly, we evaluate the effectiveness of our approach through testing and benchmarking across a diverse range of Android applications built with different frameworks. By identifying, addressing, and mitigating resource leaks, we expect to contribute to the development of more energy-efficient Android apps.

Agradecimentos

Aos meus orientadores, Professor João Bispo e Professora Alexandra Mendes, pelo acompanhamento ao longo desta dissertação, em que foram sempre capazes de antecipar as minhas necessidades para oferecer apoio.

Ao Luís Sousa, a quem eu chateava quando algo não estava a funcionar.

Aos meus amigos, tanto de Engenharia Informática como de Engenharia Eletrotécnica, e a todos os que fizeram parte da minha vida académica, sem os quais a faculdade não teria sido o mesmo.

Aos meus pais e família, cujo constante apoio tornou tudo isto possível.

A todos os professores que deixaram uma marca na minha formação.

Gustavo Amorim Santos

*“You make it up to the top of a mountain,
and what’s left for me but lightning?”*

Friedrich Nietzsche

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	1
1.3	Objectives	2
1.4	Document Structure	3
2	Background	4
3	Related Work	9
3.1	Detecting and Refactoring Resource Leaks in Android	9
3.1.1	EcoAndroid	10
3.1.2	E-APK	11
3.1.3	Relda2	12
3.1.4	RelFix	13
3.1.5	PlumbDroid	13
3.1.6	Statedroid	14
3.1.7	VALA	15
3.2	Overview	16
4	Source-to-Source Compiler for Smali	19
4.1	Our Approach	19
4.1.1	The LARA Framework	20
4.1.2	Alpakka	21
4.1.3	The Application’s Flow	24
4.1.4	Detecting Resource Leaks	27
4.1.5	Correcting Resource Leaks	32
5	Results	34
5.1	Putting It To The Test	34
5.1.1	DroidLeaks Results	34
5.1.2	Comparing with others	38
5.2	Overview	40
6	Conclusion and Future Work	41
	References	43

List of Figures

2.1	Example of method overloading in smali by invoking two versions of the <i>overloadTest</i> method [14].	5
2.2	Android’s activity lifecycle [10].	7
2.3	A database cursor leak in Bankdroid ¹ in Java, detected through Alpakka’s analysis of its smali code.	8
3.1	A resource leak caused by variant lifecycles [18].	15
4.1	Alpakka’s workflow to detect and correct resource leaks.	19
4.2	Intermediate representation of a smali Android app with ClassNodes and Assets in Alpakka.	22
4.3	Comprehensive representation of statements in Alpakka.	23
4.4	Comprehensive representation of expressions in Alpakka.	23
4.5	The control flow graph of a function with a simple switch case, created with Alpakka.	25
4.6	The same <i>Landroid/database/Cursor</i> ; resource being tracked across two branches.	29
4.7	Two different resources of the same <i>Landroid/database/Cursor</i> ; type being released with only one operation.	31

List of Tables

3.1	Details on some of the available tools to detect and correct resource leaks in Android.	16
3.2	DroidLeaks' comparison of different resource leak detection tools, modified to include Extending EcoAndroid.	17
4.1	The Android resources analyzed with Alpakka and relevant information on each of them.	28
5.1	Resource leaks detected with Alpakka in the DroidLeaks dataset.	35
5.2	Detailed metrics for processed apps from the DroidLeaks dataset, including average number of nodes from different versions of the same app, average number of edges, average number of methods, average generation time, average detection time, average correction time, and average rebuild time.	37
5.3	Number of resource leaks automatically corrected with Alpakka.	38
5.4	DroidLeaks' comparison of different resource leak detection tools, modified to include Extending EcoAndroid and Alpakka.	39
5.5	Comparison of different resource leak detection tools executed on the DroidLeaks dataset.	40

Abreviaturas e Símbolos

AAB Android App Bundle

API Application Programming Interface

APK Android Package

ART Android Runtime

AST Abstract Syntax Tree

CFG Control Flow Graph

DEX Dalvik Executable

JAR Java Archive

SDK Software Development Kit

Chapter 1

Introduction

1.1 Context

Mobile devices have become indispensable companion gadgets in our everyday lives, with Android being the most prevalent mobile operating system, boasting a 70.5% market share [23]. Its most popular app store, Google Play, hosts close to 2.5 million apps [1].

System resources, such as sensors and device peripherals, serve as the fundamental building blocks that enable the diverse functionalities of mobile applications. Naturally, the consumption of these resources is not uniform, and certain components inherently demand more battery life, like the camera and GPS.

In Android apps, resources can be acquired by invoking the respective system API. Managing the use of these resources appropriately is of extreme importance in order to avoid unnecessarily depleting the battery. As such, when they are no longer needed, they should be released.

Resource leaks in Android applications occur when developers unintentionally retain resources even after they are no longer needed. This oversight can lead to degraded app performance, increased power consumption, and an overall diminished user experience.

However, ensuring timely resource release across all execution paths is not always a trivial task due to the intricacies of Android’s lifecycle model, which moves through different states, triggering different callback methods as an app evolves through its lifecycle, turning managing resource releases into a difficult challenge.

1.2 Motivation

Previous research has explored the topics of detection [20, 26, 18, 25] and automatic refactoring [16, 3] of resource leaks in Android. However, it remains a current problem in mobile apps.

In light of this, our work introduces a solution that facilitates a shift towards making resource leak detection and resolution a common and accessible practice in Android app development. This tool is not just limited to current needs but also opens the door to future enhancements and

additional functionalities. To our knowledge, there is no other solution that offers similar capabilities, particularly in enabling the analysis and transformation of Android applications by way of reusable scripts. By integrating it into the development workflow, we aim to foster improved app performance and enhance user experience across the mobile ecosystem.

With this platform, we aim to simplify and streamline the process of analyzing and transforming Android apps, applying it to effectively detect and rectify resource leaks.

By leveraging a source-to-source compiler, our approach makes working with Android apps significantly easier, regardless of whether access to the original source code is available or how the apps were originally constructed. This framework is designed to be both powerful and user-friendly, allowing developers to address problems with minimal effort and expertise at the smali [14] code level. Resource leak detection serves as an excellent demonstration of the platform’s capabilities, showcasing its potential for broader applications.

1.3 Objectives

With this work, we aim to introduce Alpakka, a novel framework capable of decompiling and re-compiling Android APK files, enabling analysis and transformations to be performed on Android applications without requiring access to the original codebase by way of decompiled smali code. Implemented using the LARA framework [21], Alpakka builds an intermediate representation of the smali code that allows to significantly enhance the experience of analyzing and manipulating smali code. These abstractions simplify the complexities of dealing with low-level Android code, making the framework both user-friendly and robust. Being the first time the LARA Framework is applied to a lower-level language such as smali, which is akin to an assembly language, we want to explore and address the following:

RQ1: *Can the LARA framework be taken advantage of to simplify working with lower-level languages?*

As a demonstration of the framework’s potential and versatility, we aim to facilitate the identification and automatic correction of resource leaks in Android apps. This use case demonstrates not only how Alpakka can be used for research purposes but also the framework’s ability to improve app reliability and performance by addressing a common and critical issue in mobile app development. Thus exploring the question:

RQ2: *Can a source-to-source compiler facilitate research and improvements in Android applications?*

Additionally, we will attempt to present empirical evidence supporting the effectiveness and efficiency of this approach in real-world Android applications. By focusing on detecting and correcting resource leaks, we will validate the utility of our framework in enhancing the development and maintenance of mobile applications. Our goal is to demonstrate that this framework not only streamlines the development process but also significantly contributes to producing higher-quality and more resilient Android applications.

1.4 Document Structure

Following this introduction, the subsequent parts of this dissertation are organized as follows:

- Chapter 2 offers some essential background knowledge on Android applications, the intricacies of their lifecycle model, and the issue of resource leaks in Android applications;
- Chapter 3 defines the problem at hand, providing an overview of the current research on the detection and mitigation of resource leaks in Android;
- Chapter 4 details our innovative approach using a source-to-source compiler, how we adopted the LARA Framework to build it, and how we detected and corrected resource leaks using it;
- Chapter 5 presents the outcomes of our experimentation, comparing the effectiveness and reliability of Alpakka with other existing tools and answers our research questions;
- Chapter 6 concludes the dissertation, offering an overview of our contributions and some final remarks on potential avenues for future development.

Chapter 2

Background

Android ¹ is an open-source, Linux-based operating system designed primarily for mobile devices. Developed by Google, it has become the most widely used mobile OS worldwide, powering a vast array of smartphones, tablets, and other smart devices.

Android applications are installed via Android Packages (APKs), a standard format used for this purpose. Despite Android app stores transitioning towards the use of Android App Bundles (AABs) for distribution [9], the core functionality of installing an application on an Android device still revolves around APK files. In this new approach, APK files are generated from the AABs at the time of installation, with the goal of optimizing the distribution process with device-specific APKs that don't need to contain all possible resources and configurations for various device types.

An APK file is a compressed archive (ZIP file) that encapsulates all the components necessary for an Android application to function. Among other things, it contains the application's code in Dalvik Executable (DEX) files. The Dalvik bytecode [11] in these DEX files is what is then compiled by the Android Runtime (ART) when a user installs an application.

For the purposes of analysis and manipulation, DEX files can be decompiled into either Java classes or smali code.

Smali [14] is an assembler specifically designed for Android's DEX format. Its syntax, seen in Figure 2.1, was inspired by the Jasmin assembler and supports the full functionality of the DEX format, making it a precise way to represent the compiled code.

The smali project also includes dexlib2, a library that provides functionality for reading, writing, and manipulating DEX files. Dexlib2 facilitates various operations on DEX files that make it an invaluable tool for developers and researchers who need to decompile, analyze, and recompile Android applications.

Since smali is decompiled directly from the DEX format, it allows us to work with any APK file, regardless of the original development framework or programming language used. Whether the app was developed in Java, Kotlin, or any other language that compiles down to DEX bytecode, decompiling to smali ensures that we have a consistent and accurate representation of the application's instructions and functionality.

¹<https://www.android.com/>

```
1 .class public LMain;
2 .super Ljava/lang/Object;
3
4 .method public constructor <init>()V
5     .registers 1
6     invoke-direct {p0}, Ljava/lang/Object;-><init>()V
7     return-void
8 .end method
9
10 .method public static main([Ljava/lang/String;)V
11     .registers 4
12
13     sget-object v0, Ljava/lang/System;-.out:Ljava/io/PrintStream;
14
15     new-instance v1, LMain;
16     invoke-direct {v1}, LMain;-><init>()V
17     invoke-virtual {v1}, LMain;->overloadTest()Ljava/lang/String;
18     move-result-object v2
19
20     invoke-virtual {v0, v2}, Ljava/io/PrintStream;-.println(Ljava/lang/
        Object;)V
21
22     invoke-virtual {v1}, LMain;->overloadTest()I
23     move-result v2
24
25     invoke-static {v2}, Ljava/lang/Integer;-.toString(I)Ljava/lang/
        String;
26     move-result-object v2
27
28     invoke-virtual {v0, v2}, Ljava/io/PrintStream;-.println(Ljava/lang/
        Object;)V
29
30     return-void
31 .end method
32
33
34 .method public overloadTest()Ljava/lang/String;
35     .registers 1
36
37     const-string v0, "returning a string"
38     return-object v0
39 .end method
40
41 .method public overloadTest()I
42     .registers 1
43
44     const v0, 42
45     return v0
46 .end method
```

Figure 2.1: Example of method overloading in smali by invoking two versions of the *overloadTest* method [14].

Also included in an APK is the application's manifest, a critical XML file that guides the Android OS in managing an application and orchestrating its components. The manifest provides a comprehensive overview of the application's structure, which is crucial to effectively detect resource leaks in Android apps.

In this manifest, a developer outlines the various components that make up the app and how they interact within the Android framework, providing insights into an app's entry points and how its components operate together.

In Android development, a typical app contains multiple entry points, each corresponding to different types of components that handle various tasks and interactions. These include:

- **Activities** — An activity serves as the entry point for user interaction in an Android app, typically implementing one screen;
- **Services** — A service is a background process without a user interface. They handle long-running operations or background tasks such as playing music or downloading files;
- **Content Providers** — A content provider offers an app a standardized interface to access and modify data stored by itself or by other apps;
- **Broadcast Receivers** — A broadcast receiver enables an app to be notified of system-wide broadcast announcements or messages even when it is not currently running.

Each of these component types within the Android framework follows its own lifecycle that defines how it is created and destroyed. As a component changes its lifecycle state, different callback methods are invoked, offering developers precise control over the component's behavior at various stages.

As an example, let us analyze an activity's lifecycle as illustrated in Figure 2.2. Its lifecycle encompasses a series of callback methods, each corresponding to a distinct phase in the lifecycle: *onCreate()*, *onStart()*, *onResume()*, *onPause()*, *onStop()*, and *onDestroy()*.

An activity's life begins upon its creation with the *onCreate()* method being invoked, where the activity is initialized, and essential setup occurs. As the activity comes into the foreground and becomes visible to the user, *onStart()* is invoked, where a developer should initialize the code to maintain the UI. Later, *onResume()* is called, allowing for user interaction with the app. When an event that takes focus away from an activity occurs, the system calls the *onPause()* method, where a developer can pause ongoing processes or save crucial data. Notably, an activity might still be visible while being in a *Paused* state by virtue of being in multi-window mode or having a semi-transparent window on top of it and a user can return to it, leading it back to a *Resumed* state. When the activity is no longer visible, the *onStop()* callback method is invoked, where the app should adjust the work being done accordingly. From a *Stopped* state, an activity can be restarted, when a user navigates back to it, or finish entirely, leading to the *onDestroy()* method being called, allowing for cleanup and the release of any remaining resources.

Resource management across the lifecycle of a component is a critical aspect of Android app development, essential for ensuring efficient performance and optimal user experience.

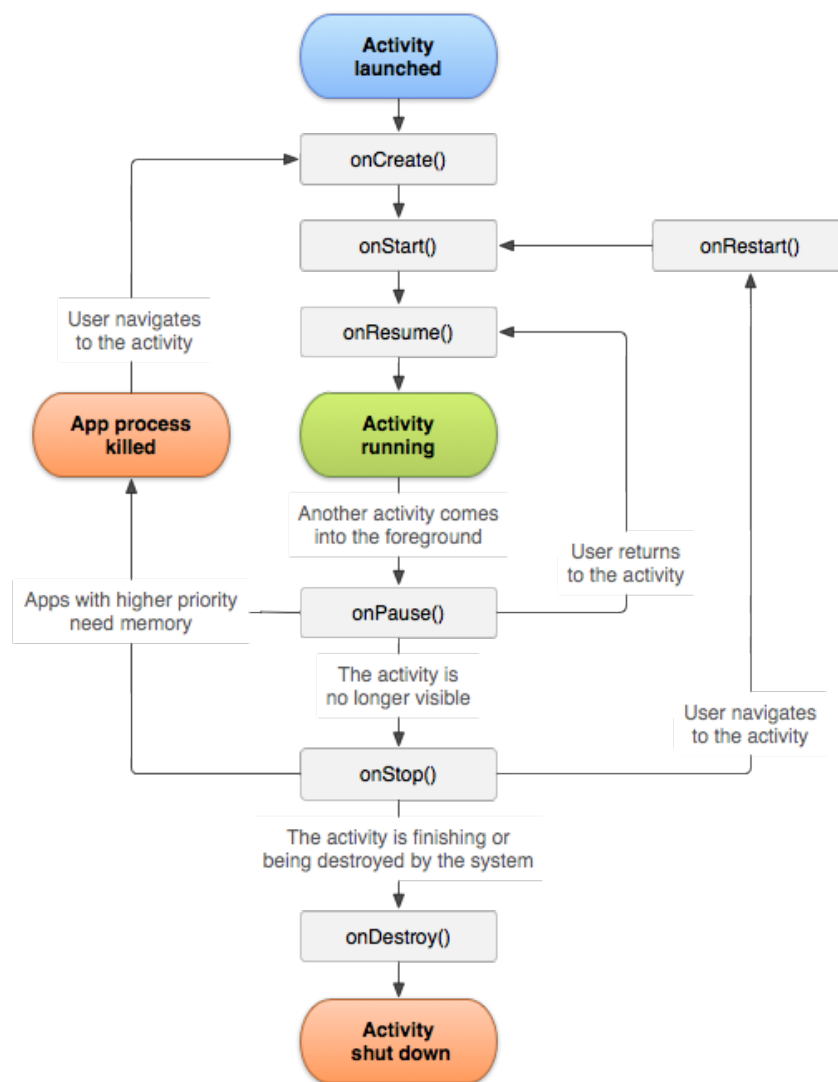


Figure 2.2: Android's activity lifecycle [10].

Improper management of resources throughout a component's lifecycle can lead to issues such as resource leaks. These leaks occur when an application acquires a resource but fails to release it properly.

For Android applications to acquire system resources, developers must use the appropriate resource-acquiring APIs provided by the Android platform. Once the resources are no longer needed, developers should invoke the corresponding release methods to dispose of the resources properly. This practice is essential because it enables the garbage collector to efficiently reclaim memory and other resources, ensuring that the application runs smoothly and does not encounter issues related to resource exhaustion.

Resource leaks can result from various issues, such as exceptions interrupting the normal execution flow or simply overlooking a resource release in certain execution paths. These leaks can persist until the app is fully closed by a user, causing a range of detrimental effects, directly

impacting:

- Performance: as an app consumes more resources than necessary.
- Stability: leading to crashes when an app exhausts system resources.
- Energy efficiency: as unreleased resources can continue to draw power unnecessarily, draining the device's battery more quickly.

Consider the example presented in Figure 2.3. The `Cursor` class provides `isBeforeFirst()` and `isAfterLast()` methods to check the current position of a cursor relative to a result set returned by a database query. In this snippet, the cursor is closed inside a conditional clause that verifies the cursor is not null, is not closed, and is positioned between the first and last rows of the query result.

Under normal circumstances, a cursor can only be either before the first row or after the last row, but not both simultaneously. However, there is a specific scenario where both `isBeforeFirst()` and `isAfterLast()` can return true: when the cursor is empty, meaning it contains no rows. In this case, the cursor is technically positioned before the first row and after the last row simultaneously because there are no rows to navigate to, yet it remains open.

```
1      Cursor c = db.fetchTransactions(...);
2      if (!(c == null || c.isClosed() || (c.isBeforeFirst() && c.
          isAfterLast())) {
3          ...
4          c.close();
5      }
6 +   if (c != null && !c.isClosed()) {
7 +       c.close();
8 +   }
```

Figure 2.3: A database cursor leak in Bankdroid² in Java, detected through Alpakka's analysis of its smali code.

Despite there being previous attempts at fixing this cursor leak manually, it remained present in this execution path. Using our source-to-source compiler Alpakka, we were able to detect it and automatically correct it with the solution presented in Figure 2.3. In this solution, we check if the cursor is not null and is still open, then close it after it is no longer necessary.

This scenario represents a partial resource leak, as it only manifests within a specific execution path. In contrast, a complete resource leak would occur if there were no attempts to release the resource across any of the possible execution paths.

²<https://github.com/liato/android-bankdroid/blob/f4fbbfd966a25a9c2e4b0a5aca381b47c2f36ac1/src/com/liato/bankdroid/BankFactory.java#L201>

Chapter 3

Related Work

When developers overlook releasing resources after they are no longer needed or fail to adequately handle exceptions related to resource management, it can lead to increased memory and resource consumption, which can degrade the performance of the app. Over time, this can exhaust system resources, causing the app to crash. Additionally, failing to release resources such as sensors can unnecessarily drain the device's battery. Proper resource management is essential to ensure the efficiency and stability of applications.

In this section, we will delve into some of the previous research concerning detecting and automatically refactoring resource leaks in the context of Android applications.

3.1 Detecting and Refactoring Resource Leaks in Android

While the subject of detecting and correcting resource leaks in Android apps is not new, it remains a current problem within the mobile application development landscape. Resource leaks persist as a critical concern impacting battery life and user experience, prompting the need for ongoing exploration.

In the existing landscape, we can find:

- **Detection tools** — Primarily focused on detecting these defects, offering insights into the presence of resource leaks within code or an APK file;
- **Refactoring tools** — A subset of tools that actively address these leaks, incorporating automated solutions for fixing them.

Detection tools are primarily aimed at developers, arming them with the knowledge to resolve these defects during the development process. Refactoring tools, while also targeted at developers aiming to streamline the correction process, may also prove beneficial to savvy users looking to improve the performance of apps on their devices, provided the barrier of entry is not too high.

Some of these tools work at the source code level, requiring access to the original codebase to detect and correct resource leaks, while others take a broader approach in which they decompile an APK to detect and fix the resource leaks on the resulting decompiled code. By taking this last

approach, these tools become agnostic to the original language or framework used to develop an app, as all frameworks can compile to an APK format that can be installed on any compatible device.

DEX files in Android APKs are commonly decompiled into either Java classes or smali code for easier analysis.

Decompiling an APK to Java allows developers to take advantage of Java's popularity and the plethora of existing tools for code analysis and manipulation. However, this process can sometimes result in broken or missing code due to the potential inaccuracies in the translation from DEX bytecode to Java source code. These issues can make it impossible to recompile the code back into a functional APK.

It may seem unintuitive that decompiling to Java can result in such losses, especially since many Android apps are originally created in Java. However, once the Java code is compiled to DEX bytecode, various optimizations are applied that complicate the reverse translation process. Additionally, the use of obfuscators by developers to protect their code can further complicate this task.

In contrast, decompiling an APK to smali code avoids these translation losses. Smali code retains a direct representation of the original DEX bytecode, ensuring that all functionality and instructions are preserved accurately.

The decompilation to smali is typically done using Apktool [24], a widely-used reverse engineering tool for Android applications. By employing Apktool to decompile an APK to smali, we obtain an accurate version of the application's code that can be effectively used for analysis and transformation. Smali provides a low-level, precise view of the code, preserving the exact sequence of instructions and enabling detailed examination and manipulation that might be lost in higher-level decompilations.

However, decompiling an APK file with Apktool is not without its challenges. Beyond converting the bytecode to smali, the tool also needs to decode various assets contained within the APK, such as XML files and PNGs. Occasionally, Apktool may encounter issues during this asset decoding process, leading to failures that can make recompiling the application impossible. Despite this, ApkTool remains a critical tool used in most projects for detecting and correcting resource leaks.

It's important to note that decompiling lawfully acquired APK files is generally legal [15, 7], even if it can go against the terms of service of some applications. However, while the act of decompiling itself is permissible, the original owner of the application retains all rights to their intellectual property.

The next subsections present some of the existing tools for the detection and correction of resource leaks on Android apps.

3.1.1 EcoAndroid

Originally designed to identify and automatically apply energy patterns in Java source code, EcoAndroid [22, 20] was later expanded to incorporate resource leak detection. This open-source

Android Studio plugin, built on top of IntelliJ’s Program Structure Interface (PSI), serves as one of the precursors to our current work, distinguishing itself as a tool that operates at the source code level. Even though it only works with Java and not Kotlin or other frameworks used in Android development, EcoAndroid poses as a valuable tool for developers looking to improve their application’s energy efficiency.

In order to understand the application’s flow, EcoAndroid builds a Control Flow Graph from which it detects leaks related to four critical Android resources:

- Camera;
- Cursor;
- SQLiteDatabase;
- WakeLock.

To build this Control Flow Graph, EcoAndroid compiles the source code to an APK and uses FlowDroid [2], a powerful taint analysis tool designed for Android applications. FlowDroid’s primary purpose is to analyze the potential paths of sensitive data through an application, helping researchers and developers identify security vulnerabilities, such as information on leaks or unauthorized data access. FlowDroid’s knowledge of Android components’ lifecycles and its specific callbacks allows it to trace the flow of data across different methods and components, providing insights into how information propagates within the app’s codebase and allowing a Control Flow Graph that accurately represents Android’s lifecycle to be generated.

To benchmark its efficacy, EcoAndroid underwent a comparative analysis against other detectors, particularly focusing on the DroidLeaks [17] dataset, comprised of manually verified resource leaks in 32 open-source Android apps.

EcoAndroid was executed on nearly the entirety of the non-exhaustive DroidLeaks dataset, being executed in 107 out of 137 APK files due to errors executing their analysis. In this analysis, in which multiple versions of the same 32 apps are present, potentially harboring the same resource leak multiple times, EcoAndroid identified 127 unique leaks. Out of these, 86 were confirmed as true positives, 28 were identified as false positives, and the other 13 could not be evaluated.

3.1.2 E-APK

E-APK [13], another precursor to our current work, extends Kadabra [5] to detect energy patterns in Android applications by decompiling APK files to Java code and analyzing the result. The overarching goal is to understand if there is a difference in applying this detection to decompiled code as opposed to the original source code.

Kadabra, as used in E-APK, is built on top of the LARA framework [21], a Java-based framework for developing scripts that perform complex code analysis and transformations. This framework has previously been used to successfully build fully working source-to-source compilers like

Kadabra and Clava [4]. These tools allow developers to write scripts that query an input program, dynamically apply modifications to each element, and produce an altered output program.

For E-APK, multiple detectors were built in the form of JavaScript scripts in order to locate energy patterns in the Java code, whether decompiled from an APK or original Java source code. These detectors leverage the capabilities of the LARA framework to navigate and analyze the code structure effectively.

In a comprehensive evaluation to test its original research question of whether it is viable to detect energy patterns in decompiled APK files versus the original source code, E-APK underwent testing against a substantial dataset comprising 420 open-source Android applications. Of these applications, 151 reported at least one energy pattern detection. However, since the evaluation focused on the viability of the detection process, the accuracy of the detections was not considered.

In the analysis of the source code of these 151 applications, E-APK identified 927 energy patterns, whereas when analyzing the corresponding release APK files, E-APK detected a total of 823 energy patterns, highlighting the feasibility of detecting energy patterns in decompiled APK files when access to the original source code is not an option.

3.1.3 Relda2

Relda2 (REsource Leak Detection for Android) [25] is a static analysis tool designed for detecting resource leaks in the Dalvik bytecode of Android applications. By taking an APK file as input, it builds a Function Call Graph to establish call relations among methods and perform inter-procedural analysis.

Relda2 is capable of providing information on leaks related to 45 Android resources, initially identified by mining Android's documentation for resource-related keywords.

The tool provides detailed trace information on potential resource leaks within the analyzed application, aiding developers in pinpointing and addressing these issues.

Relda2 is built using Androguard [8], a tool that allows for disassembling, decompiling, and analyzing Android APKs to extract valuable information about their structure and behavior.

Relda2 has two forms of operation: a flow-insensitive, which simplifies the analysis, and a flow-sensitive one, where the resource summaries are generated based on a Control Flow Graph (CFG) built using Androguard.

To assess Relda2's effectiveness, the authors conducted an evaluation against 103 real-world Android apps, some being closed source. From these, Relda2 detected that 76 applications had potential resource leaks. Using the flow-insensitive technique, Relda2 detected 69 potential resource leaks, while the flow-sensitive approach detected 121 leaks. The authors estimate the flow-insensitive technique detected 47 real resource leaks, while the flow-sensitive method detected 67 real leaks.

3.1.4 RelFix

RelFix [16] is a refactoring tool from the same developers behind Relda2 [25]. It addresses resource leaks in Android applications by decompiling APK files to smali code using Apktool [24], a popular reverse engineering tool for Android APK files. RelFix constructs a Function Call Graph that maps out the methods invoked within an Android Activity and forms an Activity Asynchronous Graph linking each asynchronous callback method to its corresponding Activity. With this information and the resource leak report from the Relda2 detector, it is able to pinpoint every location where a resource is requested but never released and apply the corresponding release operation in the appropriate Android lifecycle step to the smali code.

By working with APK files and smali code, RelFix is taking a more agnostic approach that can be applied to most applications, no matter how they were built or if a user has access to the app's source code.

To validate their tool's effectiveness, the developers conducted an evaluation on 427 real-world applications, with 165 applications initially flagged for potential resource leaks. However, upon manual examination, only 12 were conclusively confirmed to exhibit resource leaks, totaling 26 detected instances. This discrepancy between the initially flagged cases and the confirmed instances was attributed to false positives of the static analysis and failure to explore all execution paths of the app.

Notably, the evaluation extended beyond mere detection to assess the impact of automatic patches on the size and performance of the applications. Additionally, they used Android's Monkey [12] test generation tool, which generates pseudo-random click events, to evaluate RelFix's possible impact on the overall integrity and stability of the applications under scrutiny. This comprehensive testing ensured that the automated fixes applied by RelFix did not introduce new issues, thereby maintaining the overall functionality and reliability of the applications.

3.1.5 PlumbDroid

PlumbDroid [3] takes a similar approach to RelFix in which it decompiles an APK file to smali code using Apktool and builds a Control Flow Graph using Androguard with the app's resource usage. This enables PlumbDroid to detect resource leaks and insert necessary resource release operations at appropriate points in the code.

PlumbDroid is currently capable of detecting leaks related to 13 types of Android resources, 9 of which are non-aliasing, meaning their usage typically does not introduce multiple references that alias one another, such as:

- `AudioRecorder;`
- `BluetoothAdapter;`
- `Camera;`
- `LocationListener;`

- MediaPlayer;
- Vibrator;
- WakeLock;
- WifiLock;
- WifiManager.

Taking a step further, PlumbDroid introduces a validation phase that scrutinizes the proposed changes. This phase not only verifies the effectiveness of the introduced fixes for the identified resource leaks but also ensures compatibility by checking for potential conflicts with other resource usages. The validation step is designed to guarantee that the resource is released only after it has served its purpose, avoiding premature releases or conflicts with other parts of the application.

In a comprehensive evaluation against the DroidLeaks dataset, PlumbDroid was tested on 17 applications with 26 known resource leaks, of which PlumbDroid successfully detected and generated corrections for 50 resource leaks. However, 6 instances of the resource leaks were marked inconclusive, and 5 fixes were deemed invalid during the automatic validation step.

Interestingly, some of the detected leaks were found in third-party libraries integrated into the applications. This presents a significant challenge for app developers, as managing and fixing leaks in external libraries is inherently more complex and often beyond their direct control. Moreover, the widespread use of these third-party libraries across many Android apps raises the question of whether the uniqueness of the resource leaks detected accounted for this overlap.

When pitted against Relda2 in 17 apps, PlumbDroid detected 70 true positive leaks out of 78 detections, while Relda2 identified 46 true positives in 85 leak detections. Additionally, PlumbDroid's automated repair capabilities were compared against RelFix in 4 applications. In this scenario, PlumbDroid detected 11 leaks, with 9 deemed true positives. It generated 11 fixes, 2 of which were automatically labeled as invalid. RelFix, in the same scenario, detected and generated fixes for 11 resource leaks, but only 7 were confirmed as true positives.

3.1.6 Statedroid

Statedroid [26] is another noteworthy detector in the domain of resource leak detection for Android applications. Its methodology involves taking an APK file as input and employing FlowDroid to construct a Control Flow Graph, enabling static analysis to identify resource leaks and other energy-related issues by mapping out the application's control flow and tracking resource usage patterns.

Statedroid was assessed through extensive testing against a dataset comprising 100 open-source Android applications. This evaluation uncovered a total of 102 reported energy leaks across 18 of the applications. However, an examination of the results revealed that 19 of these reported cases were identified as false positives.

In a comparative analysis against Relda2, Statedroid detected 16 energy leaks in 15 apps out of a set of 55, with only 1 false positive. Relda2, when applied to the same dataset, detected 39 energy leaks on the same 15 apps with 4 false positives.

3.1.7 VALA

Vala [18] is a tool designed to detect resource leaks and other bugs caused by variant lifecycles in Android applications. Like other advanced analysis tools, Vala [18] employs FlowDroid to build a Control Flow Graph, which is instrumental in tracing the flow of resources and identifying misuse patterns that can lead to leaks.

VALA holds information on 51 pairs of acquisition/release operations gathered from previous studies.

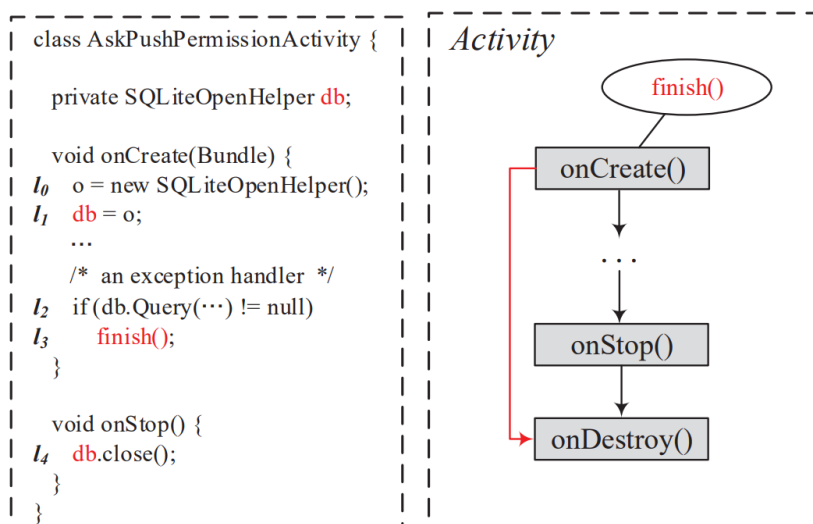


Figure 3.1: A resource leak caused by variant lifecycles [18].

To understand the impact of variant lifecycles, consider the example illustrated in Figure 3.1, a situation where an activity calls the finish method in *onCreate()*. In this case, the *onDestroy()* method will be invoked immediately, something the developer did not take into account, releasing resources only in *onStop()*, leading to a resource leak.

In their exploration, the Vala developers gathered a dataset of 287 open-source apps. However, for experimental purposes, they focused on a subset of 35 apps characterized by variant lifecycles, which are particularly prone to resource leaks due to unconventional transitions between lifecycle states. In a comparative analysis with Relda2, Vala successfully detected 6 resource leaks, manually confirmed to be true positives. In contrast, Relda2, when subjected to the same dataset, identified a substantially higher number of resource leaks, detecting a total of 177 instances. However, the manual confirmation process revealed that only 12 of these detections were accurate.

3.2 Overview

The landscape of resource leak detection tools for Android applications encompasses a variety of solutions, each with unique strengths and limitations tailored to different stages of application analysis and development. These tools can be broadly categorized based on their input types and their approaches to analyzing and refactoring code, as seen in Table 3.1.

EcoAndroid operates directly at the source code level, utilizing IntelliJ’s Program Structure Interface to identify and automatically apply energy patterns in Java source code.

Other tools, such as Statedroid and Vala, accept APK files as input. Statedroid focuses on detecting general resource leaks, while Vala specifically addresses issues caused by variant lifecycles, such as resources not being released when an activity’s lifecycle deviates from the standard sequence.

E-APK also accepts APK files but takes a different approach by decompiling them into Java code. This allows E-APK to leverage Kadabra’s AST model for energy pattern detection.

RelFix and PlumbDroid introduce an extra layer of versatility by decompiling APK files into smali code, a lower-level representation of Android application code. By working with smali code, RelFix and PlumbDroid can directly manipulate the bytecode to implement resource release operations.

All these tools share a common methodology of building some form of a Control Flow Graph to perform static analysis and identify resource misuses. A CFG provides a structured representation of all possible paths that can be taken through a program’s execution, highlighting the flow of control between various code blocks, allowing these tools to trace resource allocation and deallocation paths and pinpoint potential leaks.

Table 3.1: Details on some of the available tools to detect and correct resource leaks in Android.

Name	Primary Purpose	Input Type	Analysis Technique	Underlying Technologies	Evaluated Dataset	Detection Results	Automated Fix Capabilities
EcoAndroid	Energy pattern and resource leak detection	Java source code	Control Flow Graph	FlowDroid	DroidLeaks (29 apps)	127 unique leaks, 86 true positives, 28 false positives	No
E-APK	Energy pattern detection	APK files / Java source code	Javascript detectors	Kadabra, LARA	420 open-source apps	927 patterns in source code, 823 in APKs	No
Relda2	Resource leak detection	APK files	Function Call Graph	Androguard	103 real-world apps	Flow-insensitive: 69 leaks, 47 TP, Flow-sensitive: 121 leaks, 67 TP	No
RelFix	Resource leak correction	APK files	Function Call Graph	Apktool	427 real-world apps	26 TP leaks in 165 apps flagged for possible leaks	Yes
PlumbDroid	Resource leak detection and correction	APK files	Control Flow Graph	Apktool, Androguard	DroidLeaks (17 apps)	78 leaks, 70 true positives	Yes
Statedroid	Energy pattern and resource leak detection	APK files	Control Flow Graph	FlowDroid	100 open-source apps	102 leaks, 83 true positives	No
VALA	Resource leak detection	APK files	Control Flow Graph	FlowDroid	35 open-source apps	6 true positives in 35 apps with verified problems	No

Additionally, in the research for DroidLeaks, their authors also included comparisons on the detection performance of several popular tools, including some that are not specifically aware of Android component’s lifecycles, as seen in Table 3.2.

In their analysis, they ran these tools on a subset of the DroidLeaks dataset. They then compared the tools’ bug detection rates, which were calculated based on the proportion of known bugs the tools were able to detect in the experimented faulty apps, and their false alarm rates, calculated by how many of these bugs the tools detected in the corresponding patched apps relative to the known bugs present in the faulty apps.

Table 3.2: DroidLeaks’ comparison of different resource leak detection tools, modified to include Extending EcoAndroid.

Detector	# Experimented Bugs	# Detected Bugs (Bug detection rate)	# False Alarms (False alarm rate)
Code Inspection	89	61 (68.5%)	47 (52.8%)
Infer	72	40 (55.6%)	16 (22.2%)
EcoAndroid Extended	48	9 (18.8%)	1 (2.1%)
Lint	40	12 (30.0%)	0 (0.0%)
FindBugs	38	6 (15.8%)	0 (0.0%)
Relda2 Flow-Sensitive	15	13 (86.7%)	10 (66.7%)
Relda2 Flow-Insensitive	15	9 (60.0%)	4 (26.7%)
Elite	8	7 (87.5%)	5 (62.5%)
Verifier	8	4 (50.0%)	3 (37.5%)

Some of the tools previously mentioned were not available at the time of the DroidLeaks research, which is why they are not included in the comparison table. Despite this, the authors of Extending EcoAndroid followed the same procedures, successfully detecting nine resource leaks with EcoAndroid while generating only one false alarm result.

Based on the data presented in PlumbDroid’s paper and the corrections information the authors provide publicly, we estimate it would be capable of being experimented on 15 bugs present in the DroidLeaks dataset. Out of these 15 bugs, PlumbDroid would likely detect approximately 7 successfully, yielding a bug detection rate of 47%. Additionally, given the methodology and accuracy described, its false alarm rate would likely be very low.

This data is not an exact representation of the full capabilities of these tools, as some can detect types of resources not included in the DroidLeaks dataset, or they may be able to identify other bugs of the same resource types that were not documented in the dataset. Despite this limitation, the data serves as a valuable point of comparison, given that all the tools were tested on the same set of subjects.

It is also worth noting only a subset of these tools, such as EcoAndroid, have been made publicly available and are ready to be used in a practical manner in Android research. This limited availability restricts the widespread adoption and application of these tools across various studies and projects focusing on Android application analysis and improvement.

Our solution adopts a methodology similar to existing tools by building a Control Flow Graph to perform static analysis and identify resource leaks. The distinguishing feature of our approach, however, lies in the use of a source-to-source compiler, detailed in the following section. By leveraging a source-to-source compiler, our solution offers a more reusable and versatile framework, allowing for broader applicability and easier adaptation across different projects and environments.

Chapter 4

Source-to-Source Compiler for Smali

Resource leaks continue to be a pervasive challenge in Android application development, primarily due to the inherent difficulty in consistently identifying and rectifying them. The complexity of modern Android applications and the variability in user interactions make it challenging to eradicate resource leaks entirely, consequently contributing to an overall diminished user experience manifested in the form of degraded performance or increased power consumption. The question arises: Can the process of detecting and correcting resource leaks be streamlined, offering a more efficient and effective solution to this pervasive issue?

4.1 Our Approach

With our solution, we aim to provide actionable information about the location of resource leaks within an application. Additionally, we want to use this information to offer automatic corrections, presenting an improved application as the final output. We strive for a generic approach that is not tied to a specific language or framework. Therefore, we chose to work with APK files, which can be decompiled to smali code, making this approach the most fitting for our goals.

To that end, we present *Alpakka*, a source-to-source compiler for smali code built on top of the Lara framework. An overview of the process for analyzing and correcting resource leaks in Android applications using Alpakka is illustrated in Figure 4.1. Using a LARA Framework Weaver, we first decompile the APK file to smali using Apktool, build an AST representation of the smali code, detect and correct resource leaks using a Control Flow Graph built with one of Alpakka’s APIs and finally rebuild the modified smali code to an APK format.

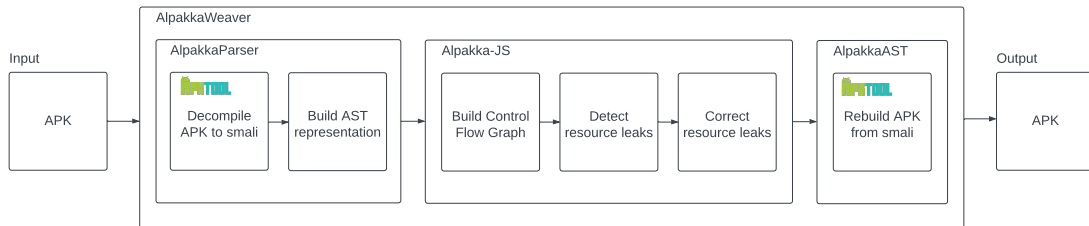


Figure 4.1: Alpakka’s workflow to detect and correct resource leaks.

4.1.1 The LARA Framework

The LARA Framework is a robust and flexible solution designed to simplify the development of source-to-source compilers. It provides a comprehensive set of tools and APIs that facilitate the creation of fully working compilers with ease, allowing developers to focus on the core aspects of their analysis and transformations without getting bogged down by the underlying complexities.

With LARA, developers can build a *Weaver* for a specific language using the framework's extensive toolkit. This *Weaver* acts as a bridge between the source code and the scripts. It allows the framework to interact with the source code by exposing its structure and semantics in a way that can be analyzed and manipulated by the scripts. Once the *Weaver* is established, scripts can be crafted to perform detailed analysis and apply precise transformations to a source program. This modular approach not only enhances reusability and maintainability but also ensures that it is possible to extend the compiler without modifying the compiler itself, since new compiler passes are libraries of interpreted scripts. This allows us to accommodate new requirements in a straightforward and efficient manner.

In essence, the LARA Framework empowers developers to leverage a powerful, language-agnostic platform for creating sophisticated source-to-source compilers, streamlining the process of program analysis and transformation.

Clava, which was specifically tailored for the C and C++ programming languages, serves as a prime example of the LARA Framework's versatility. It leverages LARA's capabilities to perform a variety of sophisticated tasks, like benchmarking to identify bottlenecks and performance issues or automatically improving software by applying known optimization patterns.

Clava's effectiveness lies in its integration with the LARA Framework, which facilitates the creation of highly customizable scripts for code analysis and transformation.

Alpakka adopts a similar approach to Clava by leveraging the LARA Framework but with a unique focus on the smali language. This makes Alpakka the first tool to extend LARA's capabilities to a lower-level language like smali. By focusing on smali code, Alpakka aims to bring these benefits to Android application development.

By leveraging the LARA framework with Alpakka, we adopt a modular approach that allows us to create specialized JavaScript libraries tailored for specific tasks. In this case, our focus is on the detection and correction of resource leaks within Android applications. This modularity facilitates the development of reusable, maintainable components that can be easily integrated and extended for various purposes.

Alpakka has significantly benefited from recent upgrades to the LARA Framework, which have made it compatible with Node.js. This compatibility allows Alpakka to be distributed as a Node.js module, facilitating its integration into various Android research and development projects. This modular approach enhances accessibility and usability, making it easier for developers and researchers to leverage Alpakka's capabilities without extensive setup or configuration.

Our Control Flow Graph API is designed for seamless integration with any project that utilizes Alpakka, providing a robust foundation for analyzing and manipulating smali code. Moreover, our

libraries for detecting and correcting resource leaks are designed to be user-friendly and straightforward, requiring only they be imported into a project.

Alpakka's use of the LARA Framework makes it a versatile and powerful tool for Android research and development. Leveraging the LARA Framework, Alpakka enables a modular approach to the analysis and transformation of Android applications, allowing developers to craft reusable JavaScript libraries and scripts for various purposes.

To our knowledge, no other publicly available tool offers such a comprehensive solution for the detailed analysis and transformation of Android applications through reusable scripts. This makes Alpakka an invaluable resource for developers and researchers working with Android.

4.1.2 Alpakka

Alpakka derives its name from the Icelandic word for package, "Pakka", following in the footsteps of smali and Dalvik, which also have Icelandic roots. As the name suggests, it accepts APK files as input, which it decompiles into smali code using Apktool.

Once the APK is decompiled into smali code, Alpakka leverages smali's own ANTLR parser to parse the application's smali files. ANTLR [19] is a powerful parser generator that facilitates the creation of parsers for reading, processing, and executing structured text. By utilizing smali's ANTLR parser, Alpakka can accurately interpret the smali code. From there, it constructs an Abstract Syntax Tree (AST) with our own Java objects, derived from the output of smali's ANTLR parser.

Generally, in our representation, seen in Figure 4.2, an Android application is made up of smali *Class Nodes* and various *Assets*, including XML files like the *Manifest*, PNG resources, and other necessary files that make up an Android application. This structure allows for detailed analysis and manipulation of both the code and the resources contained within the APK.

Smali *Class Nodes* serve as the fundamental building blocks of the APK file's code representation. Each *Class Node* includes crucial information about the class itself, such as:

- **Class Specification:** Details about the class, including its name and access modifiers.
- **Superclass Specification:** Information about the superclass that this class extends.
- **Implemented Interfaces:** Specifications of any interfaces that the class implements.

Additionally, they can contain as children *Method Nodes*, *Field Nodes*, and *Annotations*.

Field Nodes contain details about the fields of the class, including their access modifiers, type descriptors, and initial values if provided.

Method Nodes include information on its specification, such as the method's access list, name, and prototype, providing a clear understanding of the method's signature and access level. As children, a *Method Node* will have a sequence of *Statements* representing the method's body.

These statements, seen in Figure 4.3, can be:

- **Labels:** Used to mark positions within the code, often as targets for jump instructions.

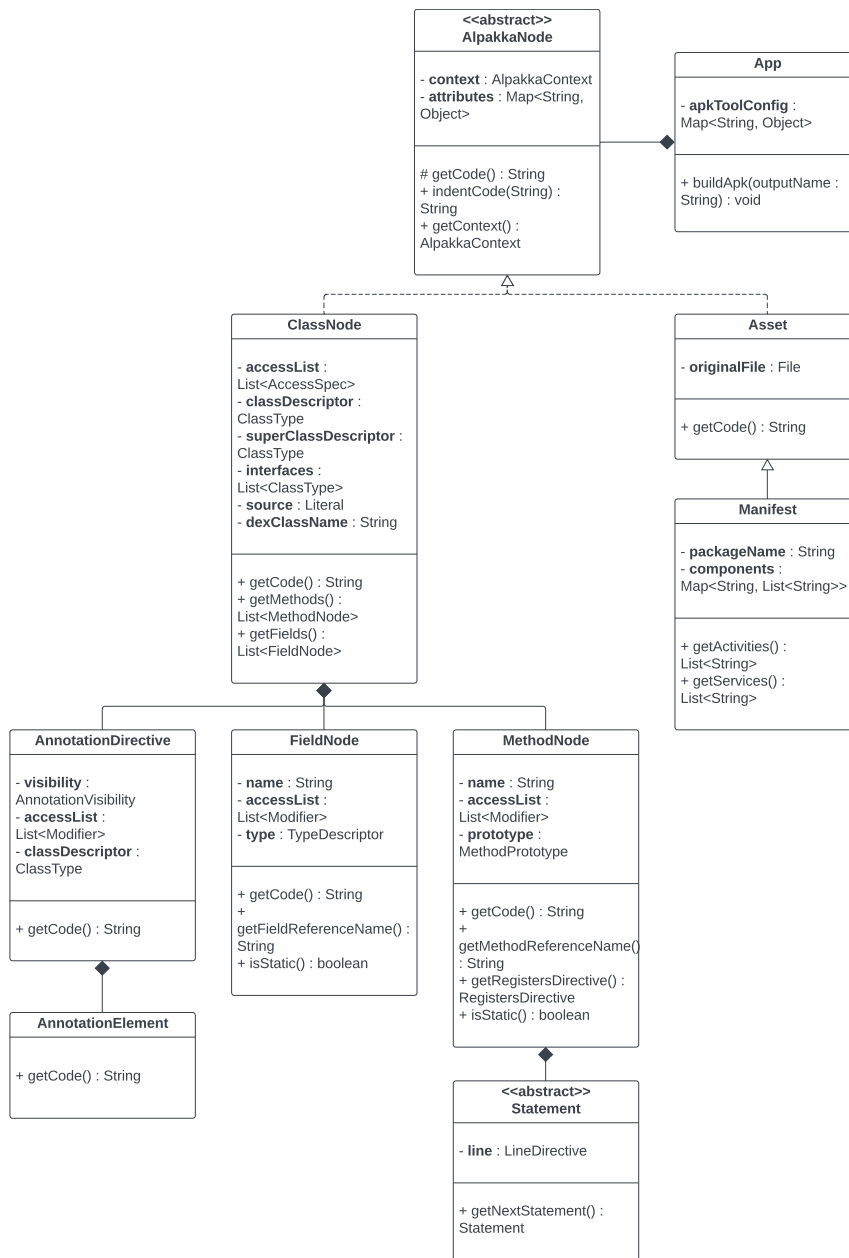


Figure 4.2: Intermediate representation of a smali Android app with ClassNodes and Assets in Alpakka.

- Directives: Such as the registers directive, which indicates the number of registers used by the method.
- Instructions: Actual operations to be performed, such as a Goto instruction. Instructions are characterized by an opcode, the operation code specifying the instruction type, and *Expressions*, as seen in Figure 4.4.

The AST serves as a representation of the application's code structure. It captures the syntactic elements and their relationships, providing a detailed map of the code's composition.

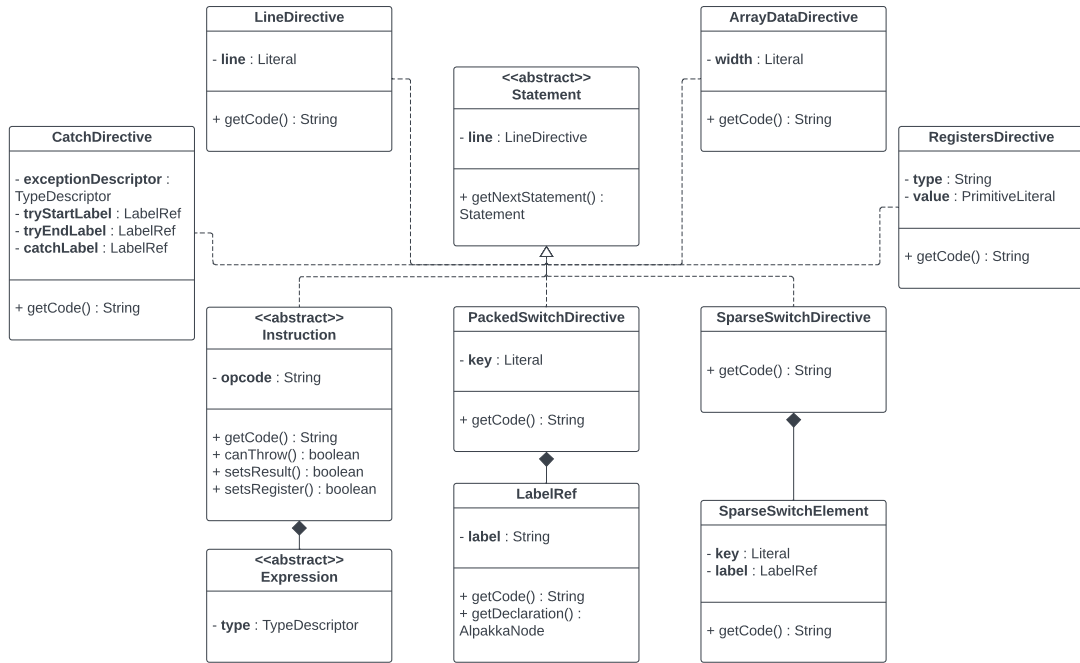


Figure 4.3: Comprehensive representation of statements in Alpakka.

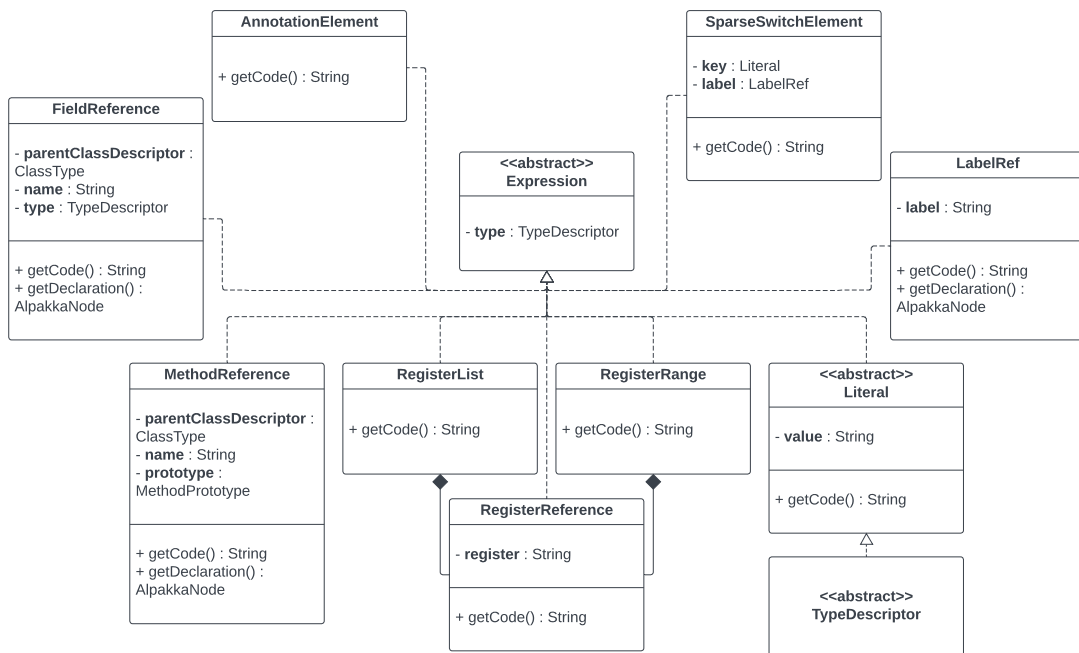


Figure 4.4: Comprehensive representation of expressions in Alpakka.

This tree structure is what allows us to perform comprehensive code analysis and transformation, as it allows Alpakka to navigate and manipulate the code with precision.

When building the AST of an application's smali code, we need to process and organize all

components, ensuring they can be easily manipulated while remaining compilable back into an APK format. Handling large applications can be memory-intensive, so we incorporated a filtering mechanism to manage this complexity efficiently. This filter allows us to specify which smali files are of interest, for example, enabling us to ignore libraries and focus on the core application's code. This selective processing makes the analysis of large applications more feasible while still guaranteeing the filtered smali files are included when recompiling the application, ensuring that the final APK remains complete and functional.

As a final step, we link references within the smali code to their corresponding declarations by mapping method calls, class references, label references, and field references to their respective definitions within the codebase, enriching this intermediate representation with semantic information.

This AST is then used in our LARA *Weaver* so we can work with the representation of these objects in our scripts and, in the end, rebuild the app back into APK format from our AST with Apktool.

4.1.3 The Application's Flow

In order to effectively detect resource leaks in Android apps, it is crucial to comprehend the program's flow. This understanding allows us to identify where resources are acquired and ensure they are properly released. Ideally, this flow analysis should be integrated within Alpakka to streamline the detection process. To achieve this, we developed a Control Flow Graph API specifically designed for use with smali code within our LARA source-to-source compiler, Alpakka. This API provides a powerful tool for analyzing the control flow of smali code, enabling our scripts to pinpoint potential resource leaks with precision.

Other LARA Framework projects have already developed their own Control Flow Graph APIs. Notably, Clava's implementation, in particular, recently underwent a refactor to enhance type safety with Typescript in an effort dubbed Clava Flow [6]. For our project, we leveraged the foundational structures from Clava Flow and focused on adapting the graph's creation to suit the unique aspects of smali code.

Smali is a lower-level representation of Android application code, distinct from higher-level languages like Java. In smali, constructs such as *while* or *for* loops are absent. That is not to say loops do not exist, as they are represented in other forms through combinations of *if* conditions and *goto* statements.

Typically, when executing a smali program, an instruction within a smali function is followed sequentially by the instruction immediately after this one, with only a few albeit important exceptions. These exceptions include:

- **Goto statements:** These cause an unconditional jump to a specified label, altering the natural sequential flow.
- **If conditions:** If the specified condition is met, execution jumps to a referenced label; otherwise, it proceeds with the instruction immediately following the if condition.

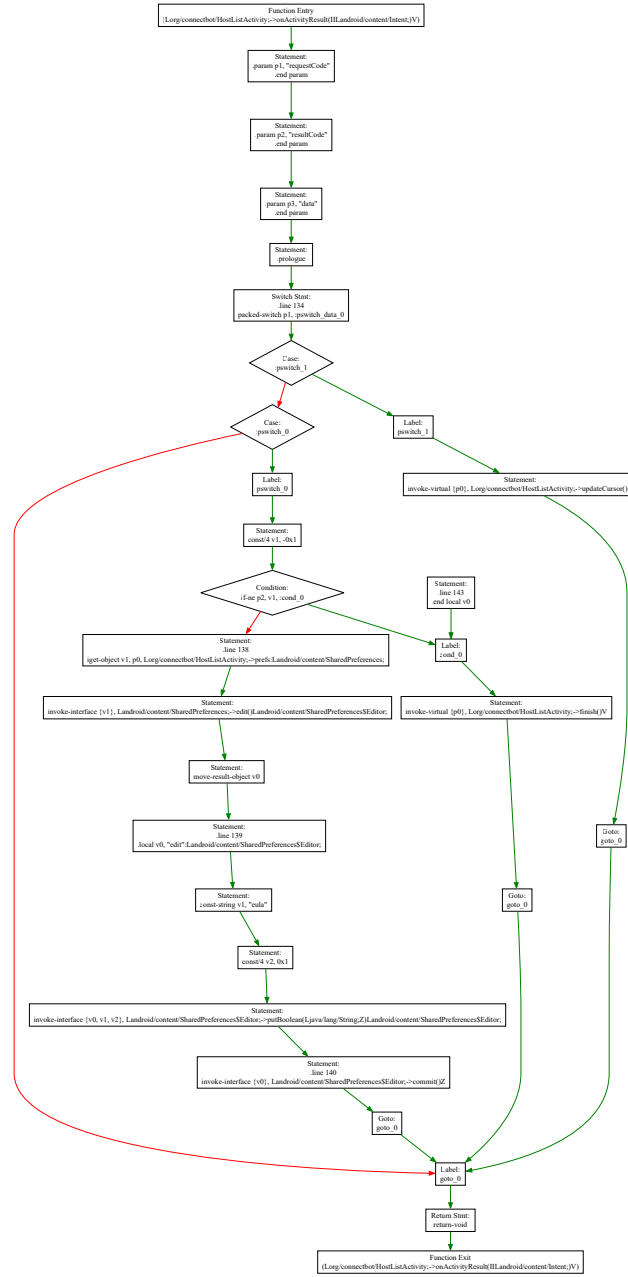


Figure 4.5: The control flow graph of a function with a simple switch case, created with Alpakka.

- **Switch statements:** If a register's value matches a switch case, execution jumps to the corresponding label; if no match is found, it continues with the instruction following the switch statement.
- **Return and throw statements:** These can terminate the execution of the current function, returning control to the caller or throwing an exception.
- **Try-catch blocks:** These define a range of instructions (from start to end labels) within which exceptions are monitored. If a matching exception occurs, the execution jumps to the

catch block specified by a third label.

Notably, try-catch blocks present a unique challenge in smali code: determining whether an instruction within a try block throws an exception. Other tools for building Control Flow Graphs of smali code, such as Androguard, adopt the approach of assuming that at the end of a set of instructions within a try block, there could be a jump to a catch label. However, this assumption oversimplifies the problem and does not capture the full complexity of exception handling in smali.

From smali’s *dexlib2*, which provides specific information about which opcodes can potentially throw exceptions, combined with Google’s documentation on Dalvik bytecode [11], we can get insights that allow us to more accurately model control flow by identifying precise points where a jump to a catch label might occur and the specific exceptions that might be thrown. However, there are still some situations where we cannot be certain an exception will be thrown. In these cases, we conservatively assume the instruction will throw a generic *java/lang/Exception*. This conservative approach, while safe, can lead to an overestimation of potential exception paths.

A particular challenge arises with certain *invoke* instructions, which utilize dynamic dispatch to determine the method being invoked, meaning the actual method is determined at runtime based on the type of the object, making it difficult to determine during static analysis. For instance, consider the following smali code snippet:

```
invoke-virtual v0, Ljava/io/Closeable;->close()V
```

In this example, the *invoke-virtual* instruction calls the *close* method on an object referenced by register *v0*. The precise implementation of the *close* method is determined at runtime based on the actual type of the object assigned to *v0*. If *v0* is an instance of *android.database.Cursor*, *java.io.InputStream*, or any other class implementing the *Closeable* interface, the corresponding *close* method specific to that class will be invoked, complicating our static analysis.

Additionally, compiled APK files do not include system libraries. Consequently, if a system API is invoked, our current information may be insufficient to ascertain whether an exception will be thrown. To overcome this limitation, we would need to supplement our analysis with additional information, such as data from the Android JAR file containing the compiled classes of Android APIs, which is a part of the Android SDK. While this is outside the current scope of our project, it stands as a promising avenue for future enhancement. By incorporating insights from the Android JAR file, we could extend Alpakka’s capabilities, making it more robust and versatile.

By thoroughly understanding these constructs and their interactions, our Control Flow Graph API within Alpakka can effectively model the flow of smali code functions, as seen in Figure 4.5.

As mentioned previously in Chapter 2, Android relies heavily on the use of callback functions to manage application behavior. When a user interacts with an application, the system orchestrates the invocation of these callback methods. Because of this, simply examining an application’s smali code does not always reveal the sequence in which functions are called. Understanding this sequence requires knowledge of the lifecycle of the specific component being analyzed, which follows a standard pattern for each component type.

To address this, Alpakka reads an application's manifest to accurately identify the declared activities and service components within the application. This information is vital for understanding the overall structure and flow of the application.

With this, we should be able to detect resource leaks and other issues, as we can precisely track resource allocation and release points throughout the application.

4.1.4 Detecting Resource Leaks

To effectively detect resource leaks, we designed a JavaScript library that we can use in our LARA source-to-source compiler, Alpakka. These libraries facilitate interaction with smali code by manipulating the elements of our AST rather than working directly with the smali code. This approach simplifies complex tasks by including a variety of helper methods that streamline the analysis process. For instance, it can easily check if a method is static, a critical factor that influences the number of registers available in a given function. By abstracting these lower-level details, Alpakka allows for more intuitive and higher-level code transformations in its scripts, making it easier to track resource usage and identify potential leaks.

The detection process begins with building the application's Control Flow Graph using Alpakka's API, followed by examining the application's declared components, identified through the manifest.

For each component, we systematically traverse the control flow graph of the standard callback methods (e.g., onCreate, onStart, onResume, onPause, onStop, onDestroy for activities) and perform static analysis. This traversal is essential for pinpointing where resources are acquired and ensuring they are subsequently released.

For this detection, we rely on predefined knowledge of which method calls generate specific resources and the corresponding close operations necessary to release them. A more generalized approach could be possible as most resources in Android applications are implementations of the Closeable interface, which provides a standard method for releasing resources. This could potentially simplify our detection process by allowing us to identify resources through their implementation of the Closeable interface. However, this generalized strategy presents some incompatibilities with our current approach. As previously mentioned, compiled APK files do not include system libraries, meaning that from the decompiled application alone, we cannot determine if a particular resource implements the Closeable interface. This is another situation where additional information, possibly from Android's JAR file, would be needed to supplement our analysis.

To add the capability to detect a new Android resource within our library, we simply need to provide specific details that facilitate accurate detection and management. These details include:

- Acquisition Methods: The classes and corresponding methods used to acquire the resource.
- Release Methods: The classes and methods used to release the resource.
- Verification Methods: Methods that check if a resource has already been released.

- **Resource Instance Behavior:** Information on whether the acquisition method always returns the same instance of the resource or creates a new instance each time.
- **Dependent Resources:** Whether the resource has any dependent resources that might affect its management.

For our current work, we supplied it with information on three Android resources:

- `android.database.sqlite.SQLiteDatabase`
- `android.database.Cursor`
- `java.io.InputStream`

Table 4.1: The Android resources analyzed with Alpakka and relevant information on each of them.

Resource	Acquisition		Release		Verification Method	Single Instance	Has Dependents
	Class	Methods	Class	Methods			
SQLiteDatabase	SQLiteOpenHelper	getReadableDatabase getWritableDatabase	SQLiteClosable SQLiteDatabase SQLiteOpenHelper	close	isOpen	True	True
SQLiteDatabase	SQLiteDatabase Context	openOrCreateDatabase	SQLiteClosable SQLiteDatabase SQLiteOpenHelper	close	isOpen	False	True
Cursor	SQLiteDatabase ContentResolver	query rawQuery queryWithFactory rawQueryWithFactory	Cursor Activity	close startManagingCursor	isClosed	False	False
InputStream	ContentResolver Context Resources AssetManager URL	openInputStream openFileInput openRawResource open openStream	InputStream	close	-	False	False

Part of this information, seen in Table 4.1, particularly for SQLiteDatabase and Cursor, was initially retrieved from Extending EcoAndroid’s [20] project. We supplemented this with additional details on both these resources and InputStream. The rationale behind selecting these specific resources is their widespread use in the Android landscape, making them a representative sample of Closeable resources that are susceptible to leak issues.

SQLiteDatabase and Cursor are pivotal in handling database operations, enabling efficient data manipulation and retrieval. InputStream, on the other hand, is crucial for various input operations, such as reading data from files or network streams. These resources are prone to leaks if not properly managed, making them excellent candidates for our analysis and detection methods.

Although EcoAndroid’s project included two other resources, Camera and Wakelock, we opted not to include them in our study. Despite not implementing the Closeable interface, these resources generally follow a similar acquisition and release pattern, which theoretically makes their inclusion straightforward. This decision was influenced by our lack of familiarity with them and their specific challenges, such as in the case of Wakelock, which is a mechanism used to keep the device awake during critical operations, demanding careful consideration of whether its acquisition is meaningful and appropriately timed.

By concentrating on SQLiteDatabase, Cursor, and InputStream, we aim to provide a clear and effective methodology for detecting and managing resource leaks, which can be broadly applicable to most other Android resources.

During this analysis in our LARA Framework JavaScript library, we focus on detecting known resource patterns, identifying situations where resources have been acquired but not released. We meticulously track acquired resources, noting the registers in which they are stored, operations where they were last involved, and any associated global fields. This tracking allows us to keep a precise record of resource usage and identify potential leaks accurately.

For each function we encounter, we process each statement, maintaining a list of visited nodes along with a state object for the resource leaks at the time of each visit, as seen in Algorithm 1. In our strategy, if the control flow loops back to a previously visited statement, we verify whether the state of leaks has changed. If discrepancies are found, we merge the resource leak maps from all incoming nodes leading to this statement and reprocess the function from that point onward.

When merging resource leak maps, we need to address scenarios where different incoming nodes have tracked the same resource leak across different statements. In such cases, we attempt to consolidate these instances by identifying a valid common node further along in the execution path, typically represented by a label such as in Figure 4.6. If this is not feasible, possibly due to the register where the resource was present being overwritten in at least one of the incoming branches, we maintain references to both statements where the resource leak was detected.

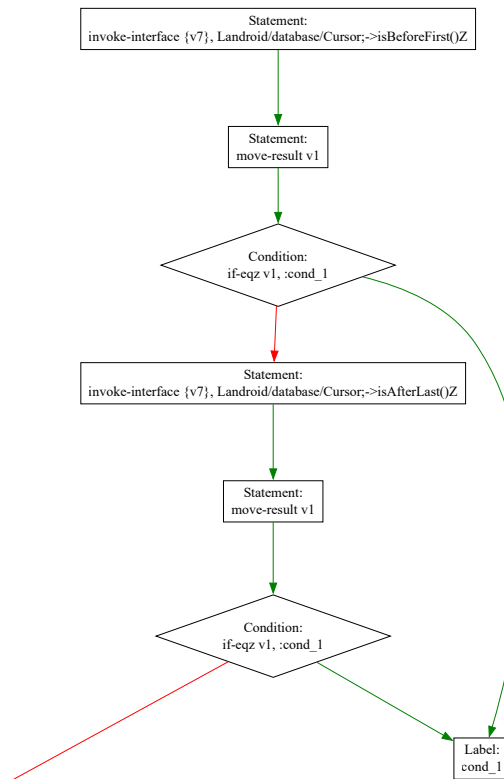


Figure 4.6: The same *Landroid/database/Cursor;* resource being tracked across two branches.

Furthermore, in situations where two different resources of the same type are created along

Algorithm 1: Traverses the CFG of a method to find resource leaks.

```

1 findResourceLeaksInFunction (node, leaksObj, fieldsObj);
   Input: Starting point node, previously tracked leaks leaksObj and previously tracked
           fields fieldsObj
2 nodesToVisit  $\leftarrow \{node, leaksObj, fieldsObj\}$ ;
3 visitedNodes  $\leftarrow \emptyset$ ;
4 while nodesToVisit.length > 0 do
5   {currNode, currLeaks, currFields}  $\leftarrow nodesToVisit.pop()$ ;
6   existingVisitedObj  $\leftarrow undefined$ ;
7   for i  $\leftarrow 0$  to visitedNodes.length - 1 do
8     if visitedNodes[i].node = currNode then
9       | existingVisitedObj  $\leftarrow visitedNodes[i]$ ;
10    end
11  end
12  if existingVisitedObj  $\neq undefined$  then
13    if existingVisitedObj.leaks = currLeaks then
14      | continue;
15    end
16    currLeaks  $\leftarrow mergeIncomingLeaks(currNode, visitedNodes)$ ;
17  end
18  if currNode.is(FunctionExitNode) then
19    | leaksObj  $\leftarrow currLeaks$ ;
20  end
21  if currNode.jp instanceof Instruction then
22    | processInstruction(currNode, currLeaks, currFields);
23  end
24  if existingVisitedObj  $\neq undefined$  then
25    if existingVisitedObj.leaks = currLeaks then
26      | continue;
27    end
28    visitedNodes.delete(existingVisitedObj);
29  end
30  visitedNodes.add({currNode, currLeaks, currFields})
31  for k  $\leftarrow 0$  to currNode.nextNodes.length - 1 do
32    | processComparison(currNode, currNode.nextNodes[k], currLeaks);
33    | nodesToVisit.push({currNode.nextNodes[k], leaksObj, fieldsObj});
34  end
35 end
36 return leaksObj;

```

different branches utilizing the same register, we strive to release both resources in a single operation at a common node ahead in the execution path, such as in Figure 4.7. This approach ensures that all potential leak points are accounted for and that our analysis remains comprehensive.

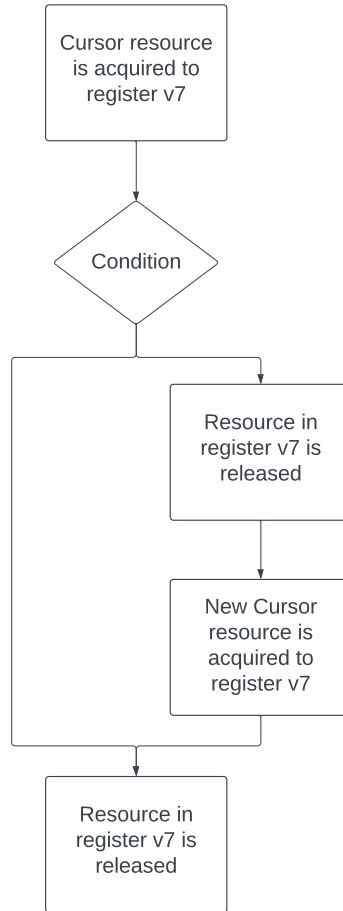


Figure 4.7: Two different resources of the same *Landroid/database/Cursor*; type being released with only one operation.

For each smali instruction we visit, we have a comprehensive processing step:

- **Register Setting:** We first determine if it can set a result in a register, possibly altering our current information. For this, we once again take advantage of the information smali provides on instruction opcodes to identify instructions that can set a register.
- **Track Register Usage:** We then identify the registers involved in the instruction to maintain an accurate record of where we last saw a resource. This helps in tracking the lifecycle of resources throughout the function's execution.
- **Method Calls:** We establish if the instruction is a method call we can process by checking if a function exists in a given class or in any of its superclasses. If it is, we combine the results from this processing with our existing resource leaks map.

- **Resource Acquisition and Release:** We verify if the instruction pertains to resource acquisition or release. If so, we update our resource leaks map accordingly.
- **Field Operations:** We evaluate whether the instruction involves setting or getting a field reference. When it does, we update our fields information to reflect this.
- **Resource State Validation:** We identify operations that check the state of a resource, such as determining if it has been closed. The result is later used to accurately reflect the resource states in conditional nodes, as these checks often influence subsequent code execution.
- **Register Moves:** We examine if the instruction is a move operation involving registers that hold relevant information. If so, we update our data to track these changes.
- **Return Statements:** We determine if the instruction is a return statement that returns a value currently associated with an acquired resource.

By following this detailed approach, we ensure that our analysis is thorough and that we accurately track and manage resource leaks throughout the application's control flow, allowing us to identify complex leak scenarios that may arise due to intricate control flow patterns.

4.1.5 Correcting Resource Leaks

From our resource leaks analysis, we derive a comprehensive list of acquired resources that are yet to be released, organized on a per-class basis. This detailed information provides developers with the necessary insights to manually address these issues by locating the exact lines in the code where resources are mishandled. However, since resource management typically follows consistent patterns, we can leverage this to implement automatic corrections, significantly reducing the time and effort required for manual fixes.

From our predefined knowledge of the resources we are analyzing, we understand the necessary methods to invoke a resource's release and the methods to verify if a resource has not been released previously. To ensure we do not disrupt an application's functionality, we deliberately avoid addressing resources with sub-dependencies, such as `SQLiteDatabase`, which can create `Cursors`. Closing the `SQLiteDatabase` can restrict further `Cursor` operations, potentially causing unintended issues.

In our approach, we first verify if the leak occurred in a component with a known standard lifecycle, such as an activity or a service. If this is not the case, we release the resource at the last point where it was found. For components with standard lifecycles, we save the resource in a global field, allowing us to retrieve and release it as soon as possible based on the callback during which it was acquired. As a final safeguard, we ensure the resource is released in the `onDestroy` callback if no earlier opportunity arises.

To release the resource, we create a new method that attempts to release it only if it is available. This involves first checking if the resource is not null and verifying if it has not been previously released, provided the resource has a method that allows such verification.

Initially, we encountered issues where invoking this new method in certain situations led to compilation errors, making it impossible to recompile the files into an APK. This problem arose because we used an *invoke-static* operation, which limits argument registers to 4 bits each, restricting us to registers v0 through v15. To overcome this challenge, we switched to using the *invoke-static/range* operation, which allows argument registers to be up to 16 bits. However, the *iput-object* operation we were using for saving resources in global fields suffers from the same 4-bit register limitation, and in this case, the solution is not as straightforward, necessitating further operations, thereby increasing the risk of inadvertently breaking the application's functionality. Our priority is to maintain the application's stability and functionality, ensuring that our interventions do not inadvertently disrupt the app's normal operation. Therefore, for the time being, we do not address situations where resources fall into this category.

The mentioned necessary additional operations involve a move operation to transfer the resource to a register with 4 bits or less. If a register matching this criteria is already not in use, we could simply add the move operation. However, if all registers are in use, we would need to introduce a new register. The complication arises because the *iput-object* operation requires a register within the 4-bit limit, and the register we just added at the end of the register list would exceed this limit if the list is already large enough. To address this, we would need to increase the values of existing registers by one, ensuring that a register like v15 can be used for the move operation while maintaining the current functionality of the application. Even though we do not currently address this, Alpakka's capabilities make this process much more manageable as it enables us to easily identify all register references within a function and systematically increase their values by one.

All these modifications are made possible through Alpakka's insertion methods, which enable us to alter the smali code based on the contents of our AST. This ensures that our changes are precise and seamlessly integrated into the application's existing code structure. The integration and implementation of these insertion methods were significantly streamlined by utilizing the LARA Framework. LARA's robust toolset and APIs simplify the process of interacting with the AST, enabling us to focus on the core aspects of our analysis and transformations.

Once the corrections are applied, Alpakka compiles all files back to an unsigned APK format using Apktool.

Chapter 5

Results

5.1 Putting It To The Test

All of our testing was performed on a 2022 Zephyrus G14 laptop equipped with an AMD Ryzen 7 6800HS processor and 16GB of DDR5 memory. The operating system used was Windows 11, and we utilized Node.js version 20.10.0 with our Weaver. Additionally, Asus' Turbo mode was enabled.

We tested the effectiveness of our resource leak detection library in real-world applications by applying our script to the applications in the DroidLeaks dataset. When resource leaks were identified, the script subsequently ran our correction library, utilizing the gathered information to apply the necessary fixes. After the corrections are made, the script generates a new APK file with the applied modifications.

The DroidLeaks dataset is comprised of 292 manually verified resource leaks in 32 open-source Android apps, which were first identified using an automated script that looked for certain keywords related to resource leaks in the commit messages of these apps. However, this approach means the dataset is non-exhaustive, as there can be additional resource leaks within these applications that were not captured by the script. Additionally, the dataset provides APK files for only a subset of the leaks mentioned, not being expected that all documented leaks are identified, as they may not be present in the available sample of APK files.

5.1.1 DroidLeaks Results

From the 292 identified resource leaks in DroidLeaks, 13 pertain to *android.database.sqlite.SQLiteDatabase*, 143 involve *android.database.Cursor* and 33 are associated with *java.io.InputStream*. Out of these, only 50 are identified as present in the available APK files, 3 of type *SQLiteDatabase*, 38 of *Cursor*, and 9 of *InputStream*. However, our analysis revealed that, from the leaks identified in the dataset, there were actually more than just these 50 reported present in the available APK files. This discrepancy arises from situations where a resource leak was identified in one version of an app within the dataset that was not offered as an

APK file, but the same leak was already present in previous versions of the app that were available as APKs.

Running our Alpakka script on all 138 APK files included in the DroidLeaks dataset - which encompasses multiple versions of some applications - and filtering our search to exclude third-party libraries and focus on first-party code, we were able to detect 93 unique leaks, 32 of which are identified in DroidLeaks, as seen in Table 5.1.

Table 5.1: Resource leaks detected with Alpakka in the DroidLeaks dataset.

Resources	Leaks	
	# Found	Of Which Are In DroidLeaks
android.database.sqlite.SQLiteDatabase	19	1
android.database.Cursor	67	31
java.io.InputStream	7	0
Total	93	32

Notably, we encountered issues when attempting to run our library on several versions of the K-9 Mail application included in the DroidLeaks dataset. This was due to a yet unresolved bug in our tools, which caused an unexpectedly high consumption of heap memory. Consequently, we were unable to process all versions of K-9 Mail effectively. The K-9 Mail application is known to include 12 leaks of type `InputStream` and 2 of type `SQLiteDatabase`, with at least 3 of the type `InputStream` being included in the available APK files. Despite the challenges, we were able to successfully detect the two `SQLiteDatabase` leaks in the versions we managed to execute. However, due to the incomplete analysis across all versions of K-9 Mail, we have decided to exclude these versions from our overall totals.

The reasons some of the leaks described in DroidLeaks were not detected are varied:

- One of the leaks was present in a test file that is not included in the compiled APK file. Since our analysis focuses on the compiled APK file, any leaks in test files would not be detected.
- Multiple leaks occurred in methods that were never actually reached during the program's execution, as appears to be the case with `nameExists()` in Google Authenticator¹. As we rely on constructing and analyzing control flow graphs, which represent possible execution paths, if a method is not invoked during any execution path, it will not be analyzed.
- Some others were not accounted for due to gaps in our documentation of Android's lifecycle callbacks. Android applications can have complex lifecycle methods, and we did not exhaustively document all possible callbacks for every component. This means that some leaks that may occur as a result of less common or undocumented callbacks were missed.

¹<https://github.com/google/google-authenticator/blob/f7dee7574d30fb7f948acd1ccc9fe2e0fcdc432/mobile/android/src/com/google/android/apps/authenticator/AccountDb.java#L76>

Effectively, out of the 112 identified resource leaks in the DroidLeaks apps we tested, our automated approach successfully detected 32. The majority of the undetected leaks were attributed to execution paths that were never reached, either because the methods were never actually invoked or because our analysis did not fully account for the specific lifecycle of the Android components involved. However, in these situations, we found it is still possible to achieve successful detection results manually. By leveraging the *findResourceLeaksInFunction* method in our library, developers can manually analyze specific functions to identify resource leaks that the flow-sensitive automated process might miss.

The manual use of our library's *findResourceLeaksInFunction* method provides a viable fallback, with the feasible future possibility of creating a flow-insensitive automated process that checks each function in an application individually for resource leaks.

From our analysis, in which we detected 93 unique resource leaks, we estimate that approximately 15% of these detected leaks are false positives. This rate of false positives stems from the previously mentioned limitation in our Control Flow Graph modeling. Specifically, the lack of detailed information on whether certain method calls will throw an exception, forcing us to conservatively assume that an instruction can throw an exception when it will not actually do so. Consequently, this limitation results in an overestimation of potential execution paths, leading to some inaccuracies in leak detection.

For some leaks, DroidLeaks includes both the version of the app containing the defect and the fixed version corrected by the original developer. Interestingly, during our analysis, we discovered that some resource leaks remained present in the APK files where they were supposedly manually corrected. This occurred because the applied corrections were not effective across all execution branches of the program, such as in the case of Figure 2.3. This emphasizes the need for more comprehensive and thorough analysis tools that can ensure resource management is properly addressed.

Processing all the applications in the DroidLeaks dataset was time-intensive, as detailed in Table 5.2, which provides the average metrics for each application, encompassing the various versions of each app available in the dataset. These metrics include the average number of nodes, edges, and methods, as well as the average generation, detection, correction, and rebuild times. It is important to note that the processing times observed in our study could be significantly reduced with targeted performance optimizations. Currently, our tools have not been specifically fine-tuned for efficiency, which leaves considerable room for improvement. Potential optimizations include removing unnecessary nodes in the CFGs and implementing more streamlined algorithms for analysis tasks. These enhancements would likely lead to substantial reductions in processing times, making our tools more practical and efficient for large-scale applications.

Additionally, it is worth noting that CallMeter, CSipSimple, and Open-GPSTracker encountered issues during the rebuilding process with Apktool. These issues arose due to problems decompiling their assets, rendering it impossible to recompile them back into a functional APK format. This could perhaps be solved using a different configuration for Apktool.

Table 5.2: Detailed metrics for processed apps from the DroidLeaks dataset, including average number of nodes from different versions of the same app, average number of edges, average number of methods, average generation time, average detection time, average correction time, and average rebuild time.

Application	Control Flow Graph			Resource Leaks			Avg. Rebuild Time (s)
	Avg. # Nodes	Avg. # Edges	Avg. # Methods	Avg. Generation Time (s)	Avg. Detection Time (s)	Avg. Correction Time (s)	
AnkiDroid	123409	135794	3673	646.772	1024.165	0.020	6.980
AnySoftKeyboard	6667	7125	244	2.861	16.018	-	2.857
APG	105730	110879	3656	119.986	602.253	0.012	16.807
Bankdroid	14358	16403	420	6.660	39.340	0.003	1.682
Bitcoin-wallet	16890	17727	742	8.447	34.535	0.019	7.126
CallMeter	36261	37718	907	13.529	113.474	0.019	F
Cgeo	161884	165802	7589	411.414	5697.510	-	26.430
ChatSecure	249302	237624	8253	833.730	2561.173	0.084	18.323
ConnectBot	17963	19147	646	10.078	37.437	0.008	2.032
CSipSimple	67227	70667	2638	57.222	415.959	0.031	F
CycleStreets	27078	26779	1535	11.302	136.309	-	3.983
FBReaderJ	76469	80470	3435	102.667	3016.393	-	3.913
Google Authenticator	3021	3065	133	0.683	9.773	0.017	0.684
Hacker News	7522	7484	432	1.631	2.473	-	2.922
IRCCloud	76714	86645	1818	215.008	936.176	-	8.956
K-9 Mail	148601	160617	4851	594.343	F	-	-
Open-GPSTracker	27722	30580	755	21.766	79.200	-	F
Osmand	258031	264193	11507	1109.821	4345.121	-	20.122
OsmDroid	18855	19142	892	4.369	88.581	-	1.495
OSMTracker	11447	11641	348	2.282	28.614	0.016	2.008
Owncloud	80580	85506	2561	109.932	1090.461	0.006	8.002
Quran for Android	17592	18080	737	5.745	64.007	-	3.558
SipDroid	28727	30219	988	10.116	94.562	0.003	4.923
SMSDroid	13536	14074	365	223.449	111.725	0.014	1.296
Surespot	38563	41419	1549	49.052	136.640	-	11.870
Terminal	14663	15029	677	3.035	22.445	-	1.232
Transdroid	80510	88033	2773	308.945	492.419	0.002	8.531
Ushahidi	39284	40063	1877	38.692	101.137	0.003	3.507
VLC	39168	40527	1651	27.928	150.485	0.005	6.247
WordPress	111582	114600	4908	211.928	4010.632	0.014	12.845
Xabber	84421	84984	4285	91.826	1343.030	-	15.407
Zxing	29361	30271	912	6.828	15.910	-	1.272
Average	63536	66010	2430	164.439	865.095	0.016	7.322

By running our script again on the corrected APK files generated by Alpakka using the detection information, we verified that our library successfully and automatically corrected 45 resource leaks, as seen in Table 5.3. Specifically, we achieved the automatic correction of 45 resource leaks across all execution branches in which they occurred. This was out of a total of 74 detected leaks involving *InputStream* and *Cursor* resources. Our current solution does not attempt to release *SQLiteDatabase* resources due to the possible dependencies associated with them, which could potentially lead to unintended issues in the application’s functionality.

Additionally, for resources held in registers exceeding 4 bits, we did not apply corrections for the time being. Addressing these cases would necessitate further move operations, a situation we

Table 5.3: Number of resource leaks automatically corrected with Alpakka.

Resources	# Leaks Fixed
android.database.Cursor	43
java.io.InputStream	2
Total	45

are not addressing for now.

5.1.2 Comparing with others

Comparing the performance of our library against other tools that were executed on the DroidLeaks dataset we can understand the strengths and limitations of our library.

Building upon DroidLeaks’ existing comparison, we found that our Alpakka library was compatible with 50 of the experimented bugs. However, 2 of these were within versions of K-9 Mail we were unable to analyze. Therefore, our analysis focuses on the remaining 48 leaks, as detailed in Table 5.4.

Alpakka’s library achieved a bug detection rate of 33.3%, successfully identifying 16 out of 48 experimented leaks. More importantly, Alpakka demonstrated a very low false alarm rate of 2.1%, showcasing its reliability.

The one false alarm was encountered in the *getConversation* method of SMSDroid. Here, the use of the *synchronized* keyword in Java encapsulates the entire method within a try-catch block when translated to smali code. Due to our previously mentioned limitation in the modeling of the CFG, our analysis flagged a resource leak as still present in a possible execution path, based on the information available in the Control Flow Graph, even though the leak had already been properly addressed.

Extending EcoAndroid’s close performance to Alpakka in terms of the false alarm rate is noteworthy, as EcoAndroid significantly influenced our work, sharing some of its foundational principles. Alpakka’s superior bug detection rate can be attributed to a more in-depth and nuanced analysis.

As previously stated, we can estimate PlumbDroid would have likely detected approximately 7 out of the 15 leaks it would have been experimented on while maintaining a low false alarm rate. This gives an estimated detection rate of around 47%, which, while higher than Alpakka’s, comes with the caveat of being an estimate.

Alpakka strikes a balance between detection rate and false alarm rate, making it a practical tool for Android research and development.

Going beyond this test, the authors of some of these tools conducted a broader analysis by executing them on a wider set of applications from the DroidLeaks dataset, to identify resource leaks in apps that supposedly did not include any identified leak of a resource type their tool was designed to detect.

Table 5.4: DroidLeaks’ comparison of different resource leak detection tools, modified to include Extending EcoAndroid and Alpakka.

Detector	# Experimented Bugs	# Detected Bugs (Bug detection rate)	# False Alarms (False alarm rate)
Code Inspection	89	61 (68.5%)	47 (52.8%)
Infer	72	40 (55.6%)	16 (22.2%)
Alpakka	48	16 (33.3%)	1 (2.1%)
EcoAndroid Extended	48	9 (18.8%)	1 (2.1%)
Lint	40	12 (30.0%)	0 (0.0%)
FindBugs	38	6 (15.8%)	0 (0.0%)
Relda2 Flow-Sensitive	15	13 (86.7%)	10 (66.7%)
Relda2 Flow-Insensitive	15	9 (60.0%)	4 (26.7%)
Elite	8	7 (87.5%)	5 (62.5%)
Verifier	8	4 (50.0%)	3 (37.5%)

It is important to note that not all tools were tested on the same number of applications. Despite this, we can still draw meaningful comparisons based on the available data, as illustrated in Table 5.5.

EcoAndroid was executed on 107 APK files from the DroidLeaks dataset, detecting 127 unique leaks, 86 of which the authors confirmed as true positives.

PlumbDroid was tested on 17 applications from the DroidLeaks dataset, detecting 50 resource leaks. Of these, 6 were deemed inconclusive, and 5 of the automatic fixes were deemed invalid. However, based on the corrections data provided by the authors, it appears that several of the detected leaks were in third-party libraries integrated into the applications. As such, since many Android apps end up using the same third-party libraries, we are not sure about the uniqueness of the detected leaks.

Our Alpakka library was executed on 124 APKs from the DroidLeaks dataset, which corresponds to the entire dataset, excluding the APKs for K-9 Mail. In this analysis, we detected 93 unique resource leaks. However, based on our findings, we estimate that approximately 15% of these detected leaks are false positives due to a limitation in our Control Flow Graph modeling, leading to some inaccuracies in leak detection. It is worth noting our analysis was filtered to ignore code in third-party libraries and to focus on first-party code, ensuring that the detected leaks were specific to the application’s own codebase and not influenced by external libraries. Additionally, we were able to automatically apply fixes to 45 of the detected resource leaks.

In summary, while our Alpakka library still has room for improvement, it has already demonstrated its effectiveness as a tool for detecting and correcting resource leaks in Android applications. Alpakka’s performance, when compared to other available options, shows promising results, indicating its potential as a valuable asset for developers concerned with resource management issues. One of the key strengths of Alpakka is its competitive false positive rate, which aligns closely with that of the best-performing existing tools. This reliability instills confidence in

Table 5.5: Comparison of different resource leak detection tools executed on the DroidLeaks dataset.

Detector	# Experimented APKs	# Detected Unique Bugs	# True Positives	# Corrections Made
Alpakka	124	93	79 (estimated)	45
EcoAndroid Extended	107	127	86	NA
PlumbDroid	17	50	44	45

developers using Alpakka.

5.2 Overview

Given our overall positive results in automating the detection and correction of resource leaks, we can confidently address our *RQ2* of whether a source-to-source compiler can facilitate research and improvements in Android apps. A source-to-source compiler proves to be an exceptionally effective tool in this domain, exemplified by our work with Alpakka. By using a source-to-source compiler for the lower-level smali syntax, we were able to sidestep any challenges associated with the diverse languages and frameworks used in Android development, facilitating efficient analysis and modification of Android applications without necessitating access to the original codebase. Our experience demonstrates that a source-to-source compiler is not only beneficial for detecting and correcting resource leaks but also holds potential for various other applications. For instance, it could be used in a research setting to employ static analysis for the detection of malware patterns in Android apps. Moreover, this approach could be adapted for other types of static code analysis, such as performance optimization or code quality improvement, further underscoring its utility and versatility in the Android development ecosystem.

By working with the LARA framework in Alpakka, we were able to effectively simplify the process of analyzing and manipulating lower-level smali code, thereby answering our *RQ1* regarding the utility of the LARA framework in simplifying work with lower-level languages. Smali, an assembly-like language for Android’s Dalvik bytecode, often presents complexities that are challenging to navigate. Leveraging the LARA framework enabled us to develop reusable high-level scripts, significantly enhancing our ability to interact with smali code more intuitively. Moreover, since smali does not allow as many liberties as higher-level languages, identifying and matching specific patterns became more straightforward. Using LARA libraries, we could effectively detect these consistent patterns and reliably pinpoint issues such as improper resource handling within Android applications.

Chapter 6

Conclusion and Future Work

Resource leaks in Android applications remain as a persistent challenge, negatively impacting performance, battery life, and user experience. Our research and development efforts have culminated in several key contributions to advance this field:

- Alpakka, a source-to-source compiler designed for the smali syntax, which can perform analysis and modifications on Android apps using our intermediate representation of smali code without needing access to the original codebase. This approach simplifies the process of handling diverse languages and frameworks inherent in Android development.
- An easy-to-use Alpakka API for building Control Flow Graph representations of Android applications in smali, facilitating detailed static analysis.
- An Alpakka library capable of automatically detecting and correcting resource leaks in Android applications. By leveraging the LARA framework, Alpakka simplifies the process of interacting with lower-level smali code, making the identification and rectification of resource leaks more efficient and reliable.

Our extensive testing on the DroidLeaks dataset and comparisons with existing solutions confirm that Alpakka is effective in identifying and rectifying resource leaks. Nevertheless, we see this as a starting point, with significant potential for iterative improvements.

Alpakka could be significantly enhanced by incorporating information retrieved from the Android JAR file included with each version of the Android Software Development Kit (SDK). This enhancement would enable us to achieve a more precise representation of an application's Control Flow Graph. By leveraging the data available in the JAR file, we can bridge existing gaps in our representation, particularly in identifying whether an instruction may throw an exception. As we refine our CFG model and incorporate more precise data on method behaviors and exception handling, we anticipate a reduction in false positives when detecting resource leaks, thereby enhancing the accuracy and reliability of our process. Additionally, With this enriched data, we would be able to accurately determine whether resources implement the *Closeable* interface, which is crucial for

a more generalized detection and correction strategy. This approach would streamline our process by allowing us to universally identify resources that require proper handling and release, thus enhancing our tool's effectiveness in detecting and correcting resource leaks.

Furthermore, we also intend to add other future improvements, such as a way to easily identify an APK file modified with Alpakka, ensuring that they are not mistakenly passed off as originals, preventing potential confusion and misuse.

Moreover, Alpakka's functionality extends beyond merely addressing resource leaks. It is feasible that it could be used to automate the detection of malware patterns in Android apps through static analysis or employed to automate the generation of mutants for mutation testing in Android apps. Alpakka presents itself as a versatile tool for research and development on Android applications, offering capabilities that can be expanded and adapted to various needs in the domain.

In conclusion, our work with Alpakka not only addresses the pressing issue of resource leaks in Android applications but also lays the foundation for broader applications in Android research.

References

- [1] AppBrain. Number of android apps on google play. Available at <https://www.appbrain.com/stats/number-of-android-apps>, last accessed in December 2023.
- [2] 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. *SIGPLAN Not.*, 49(6):259–269, jun 2014.
- [3] Bhargav Nagaraja Bhatt and Carlo A. Furia. Automated repair of resource leaks in android applications. *Journal of Systems and Software*, 192:111417, 2022.
- [4] João Bispo and João M.P. Cardoso. Clava: C/c++ source-to-source compilation using lara. *SoftwareX*, 12:100565, 2020.
- [5] Tiago Carvalho, João Bispo, Pedro Pinto, and João M.P. Cardoso. A dsl-based runtime adaptivity framework for java. *SoftwareX*, 23:101496, 2023.
- [6] Pedro Correia. clava-flow. Available at <https://github.com/Goncalerta/clava-flow>, last accessed in June 2024.
- [7] CURIA. Reference for a preliminary ruling – copyright and related rights – legal protection of computer programs – directive 91/250/eec – article 5 – exceptions to the restricted acts – acts necessary to enable the lawful purchaser to correct errors – concept – article 6 – decompilation – conditions. Available at <https://curia.europa.eu/juris/document/document.jsf?text=&docid=247056&pageIndex=0&doclang=en&mode=req&dir=&occ=first&part=1&cid=6413406>, last accessed in July 2024.
- [8] Anthony Desnos. Androguard. Available at <https://github.com/androguard/androguard>, last accessed in May 2024.
- [9] Google. About android app bundles. Available at <https://developer.android.com/guide/app-bundle>, last accessed in June 2024.
- [10] Google. The activity lifecycle. Available at <https://developer.android.com/guide/components/activities/activity-lifecycle>, last accessed in December 2023.
- [11] Google. Dalvik bytecode format. Available at <https://source.android.com/docs/core/runtime/dalvik-bytecode>, last accessed in June 2024.
- [12] Google. Ui/application exerciser monkey. Available at <https://developer.android.com/studio/test/other-testing-tools/monkey>, last accessed in January 2024.

- [13] Nelson Gregório, João Bispo, João Paulo Fernandes, and Sérgio Queiroz de Medeiros. E-apk: Energy pattern detection in decompiled android applications. *Journal of Computer Languages*, 76:101220, 2023.
- [14] Ben Gruver and Google. Smali. Available at <https://github.com/google/smali>, last accessed in June 2024.
- [15] Law Information Institute and Cornell Law School. reverse engineering. Available at https://www.law.cornell.edu/wex/reverse_engineering, last accessed in July 2024.
- [16] Jierui Liu, Tianyong Wu, Jun Yan, and Jian Zhang. Fixing resource leaks in android apps with light-weight static analysis and low-overhead instrumentation. *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 342–352, 2016.
- [17] Yepang Liu, Jue Wang, Lili Wei, Chang Xu, Shing-Chi Cheung, Tianyong Wu, Jun Yan, and Jian Zhang. Droidleaks: a comprehensive database of resource leaks in android apps. *Empirical Software Engineering*, 24(6):3435–3483, Dec 2019.
- [18] Yifei Lu, Minxue Pan, Yu Pei, and Xuandong Li. Detecting resource utilization bugs induced by variant lifecycles in android. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2022*, page 642–653, New York, NY, USA, 2022. Association for Computing Machinery.
- [19] Terence Parr and Sam Harwell. Antlr. Available at <https://www.antlr.org/>, last accessed in June 2024.
- [20] Ricardo B. Pereira, João F. Ferreira, Alexandra Mendes, and Rui Abreu. Extending ecoandroid with automated detection of resource leaks. In *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems, MOBILESoft '22*, page 17–27, New York, NY, USA, 2022. Association for Computing Machinery.
- [21] 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.
- [22] 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.
- [23] Statista. Global market share held by mobile operating systems from 2009 to 2023, by quarter. Available at <https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/>, last accessed in December 2023.
- [24] Connor Tumbleson. Apktool. Available at <https://apktool.org/>, last accessed in June 2024.
- [25] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. Light-weight, inter-procedural and callback-aware resource leak detection for android apps. *IEEE Transactions on Software Engineering*, 42(11):1054–1076, 2016.
- [26] 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.