Detecting Resource Leaks on Android with Alpakka

Gustavo Santos
Faculty of Engineering, University of Porto
Porto, Portugal
up201907397@up.pt

João Bispo
INESCTEC, Faculty of Engineering, University of Porto
Porto, Portugal
jbispo@fe.up.pt

Alexandra Mendes
INESC TEC, Faculty of Engineering, University of Porto
Porto, Portugal
alexandra@archimendes.com

Abstract

Mobile devices have become integral to our everyday lives, yet their utility hinges on their battery life. In Android apps, resource leaks caused by inefficient resource management are a significant contributor to battery drain and poor user experience. Our work introduces Alpakka, a source-to-source compiler for Android's Smali syntax. To showcase Alpakka's capabilities, we developed an Alpakka library capable of detecting and automatically correcting resource leaks in Android APK files. We demonstrate Alpakka's effectiveness through empirical testing on 124 APK files from 31 real-world Android apps in the DroidLeaks [12] dataset. In our analysis, Alpakka identified 93 unique resource leaks, of which we estimate 15% are false positives. From these, we successfully applied automatic corrections to 45 of the detected resource leaks.

CCS Concepts: • **Human-centered computing → Ubiquitous and mobile computing**.

Keywords: Mobile Applications, Energy Efficiency, Resource Leaks, Decompiler, Android, Smali, Green Software

ACM Reference Format:
Gustavo Santos, João Bispo, and Alexandra Mendes. 2025. Detecting Resource Leaks on Android with Alpakka. In *Proceedings of International Conference on Software Language Engineering (SLE '25).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

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 [19]. 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.

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. A *resource leak* occurs when developers unintentionally retain resources 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.

In light of this, our work introduces a solution that facilitates a shift towards making resource leak detection and resolution more accessible in Android app development.

Alpakka is a powerful source-to-source compiler for the Smali syntax, enabling developers to analyze and transform compiled Android APK files without the need for access to the original source code. By working directly with Smali, a human-readable representation of Android's DEX bytecode, instead of decompiled Java code, Alpakka ensures that all functionality and instructions are accurately preserved while offering the capability to apply modifications. Furthermore, by targeting APKs, the approach is agnostic to the programming language or platform used to develop the application.

To demonstrate how Alpakka could be used in Android research, we applied it to the detection and automatic correction of resource leaks. However, Alpakka's potential extends far beyond this. Its ability to analyze and transform APK files opens up possibilities for a wide range of tasks, including performance optimization and malware detection.

To assess Alpakka's performance for the specific use case of detecting and correcting resource leaks, we executed it on 124 Android APK files from a diverse set of 31 real-world open-source applications from the DroidLeaks [12] dataset.

In summary, our main contributions are:

SLE '25, Koblenz, Germany

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

- Alpakka, a source-to-source compiler for Android's Smali syntax, capable of performing analysis and modifications on most Android apps without requiring access to the original codebase.
- An easy-to-use Alpakka API for building Control Flow Graph representations of Android apps.
- Alpakka libraries for detecting and automatically correcting resource leaks in Android apps.
- An evaluation of Alpakka with 31 real-world applications from the DroidLeaks dataset (124 APKs).
- A comparative analysis of Alpakka's performance against other state-of-the-art resource leak detection and resource leak correction tools.

2 Background

2.1 The Android Application

Android applications are installed via Android Packages (APKs), a standard format used for this purpose. 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 [8] 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. 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 often results in broken or missing code due to the potential inaccuracies in the translation from DEX bytecode to Java source code. These issues usually make it unfeasible 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 [10] is an assembler specifically designed for Android's DEX format. Its syntax was inspired by the Jasmin assembler [14] 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 those who need to decompile, analyze, and recompile Android applications.

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 with the Android framework, providing insights into an app's entry points.

A typical Android app may contain multiple entry points. From multiple activities to other kinds of components such as broadcast receivers, services, and content providers — declared in the manifest and invoked independently by the system. Each component has its own lifecycle[1] 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.

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.

2.2 Resource Leaks in Android

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 when it ceases to be needed.

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, stability, and energy efficiency.

Consider the example presented in Figure 1. 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 not pointing towards the positions before the first row or after the last row of the query result (i.e. it must be positioned between the first and last rows of the query result).

[1] https://developer.android.com/guide/components/activities/activity-lifecycle

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.

We were able to detect this resource leak using Alpakka and correct it automatically with the solution presented in Figure 1. In this solution, we check if the cursor is not null and is still open, then close it after it is no longer necessary.

```
    Cursor c = db.fetchTransactions (...);
    if (!(c == null || c.isClosed() ||
    (c.isBeforeFirst() && c.isAfterLast ()))) {
        ...
        c.close();
    }
+   if (c != null && !c.isClosed()) {
+       c.close();
+   }
```

Figure 1. A database cursor leak in Bankdroid[2], detected through Alpakka's analysis of Smali code, via GitHub.

3 Related Work

While the subject of detecting and correcting resource leaks in Android apps is not new, it persists as a critical concern impacting battery life and user experience. 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.

In the following sections, we discuss tools in each of these two categories.

3.1 Resource Leak Detection Tools

EcoAndroid [16, 18] is a detection tool for resource leaks in Android apps. It is distributed as an open-source Android Studio plugin, originally designed to identify and automatically apply energy patterns in Java source code and later expanded to incorporate resource leak detection. Although it only supports Java, it poses as a convenient tool for developers looking to improve their application's energy efficiency. EcoAndroid detects leaks related to four critical Android resources: Camera, Cursor, SQLiteDatabase, and WakeLock.

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

In order to understand the application's flow, EcoAndroid builds a Control Flow Graph (CFG) using 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 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 the generation of a CFG that accurately represents Android's lifecycle.

Relda2 (REsource Leak Detection for Android) [21] is another example of a resource leaks detection tool for Android apps, in particular for Dalvik bytecode. 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. It 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 [7], a tool that allows for disassembling, decompiling, and analyzing Android APKs to extract valuable information about their structure and behavior.

E-APK [9] is a tool that 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 [17], a Java-based framework for developing source-to-source compilers for code analysis and transformations. 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.

Vala [13] is a tool designed to detect resource leaks and other bugs caused by variant lifecycles in Android apps — problems that arise when an activity's lifecycle deviates from the standard sequence. Like other advanced analysis tools, Vala [13] employs FlowDroid to build a CFG, 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.

Statedroid [22] 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.

3.2 Resource Leak Correction Tools

The following tools, in addition to detecting resource leaks, also support the automatic correction of such leaks. *RelFix* [11] and *PlumbDroid* [3] are examples of tools that offer automatic correction capabilities. *RelFix* [11] is a refactoring tool from the same developers behind Relda2 [21]. It addresses resource leaks in Android applications by decompiling APK files to Smali code using Apktool [20], 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. It is then possible to apply to the Smali code the corresponding release operation in the appropriate Android lifecycle step.

PlumbDroid [3] takes a similar approach to RelFix in which it decompiles an APK file to Smali code using Apktool and builds a CFG using Androguard with the app's resource usage. This enables PlumbDroid to detect resource leaks and insert resource release operations at appropriate points in the code. It 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. These are: AudioRecorder, BluetoothAdapter, Camera, LocationListener, MediaPlayer, Vibrator, WakeLock, WifiLock, and WifiManager.

3.3 Overview

The landscape of resource leak detection tools for Android apps 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 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.

By focusing on APK files, 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. This approach eliminates the need for access to the application's original codebase and makes the tools independent of the language in which the app was developed.

E-APK also accepts APK files but takes a different approach by decompiling them into Java code. This allows E-APK to have the benefits of both being able to target APKs, and take advantage of mature Java analyses, but prevents modifications to the application.

RelFix and PlumbDroid introduce an extra layer of versatility by decompiling the DEX files in an APK into Smali code, a lower-level representation of Android application code, to detect and fix the resource leaks on the resulting decompiled code. By working with Smali code, RelFix and PlumbDroid can directly manipulate the bytecode to implement resource release operations.

The decompilation to Smali is typically done using Apktool [20], a widely-used reverse engineering tool for Android applications. This is the case in both RelFix and PlumbDroid. 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.

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 is worth noting that only a subset of these tools, such as EcoAndroid, has 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 in Android APKs. The distinguishing feature of our approach lies in proposing a script-based source-to-source compiler for Smali. Analysis and transformations are implemented as libraries which can be easily modified and distributed, avoiding modifications to the compiler itself, which is responsible for interpreting the scripts. This provides a versatile framework with readily composable components, allowing for broader applicability beyond resource leaks. In contrast with the previously mentioned tools, resource leaks serve only as a demonstration of what is possible to achieve with Alpakka in regards to Android research, given its script-based nature.

4 The Alpakka Compiler

4.1 LARA Framework

Alpakka is a source-to-source compiler for Smali code that is built on top of the LARA Framework [17], a library meant to simplify the development of source-to-source compilers. The LARA Framework provides a comprehensive set of tools and APIs that facilitate the creation of fully working compilers,

Table 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	Abstract Syntax Tree	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

allowing developers to focus on the core aspects of their analysis and transformations.

When working with LARA, developers build a *Weaver* for a specific language using the framework's 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 created to perform detailed analysis and apply precise transformations to a source program. These scripts can be developed in either JavaScript or TypeScript. The modular nature of this 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.

Alpakka is also the first LARA compiler for a low-level assembly-like language like Smali, since other LARA compilers have targeted high-level languages, such as Java [5] and C/C++ [4].

4.2 Alpakka's Implementation

An overview of Alpakka's workflow is illustrated in Figure 2. 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, the Alpakka compiler 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 [15] parser to parse the application's Smali files. 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, and capable of generating back Smali code that reflects changes in the representation.

Generally, in our representation, 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*.

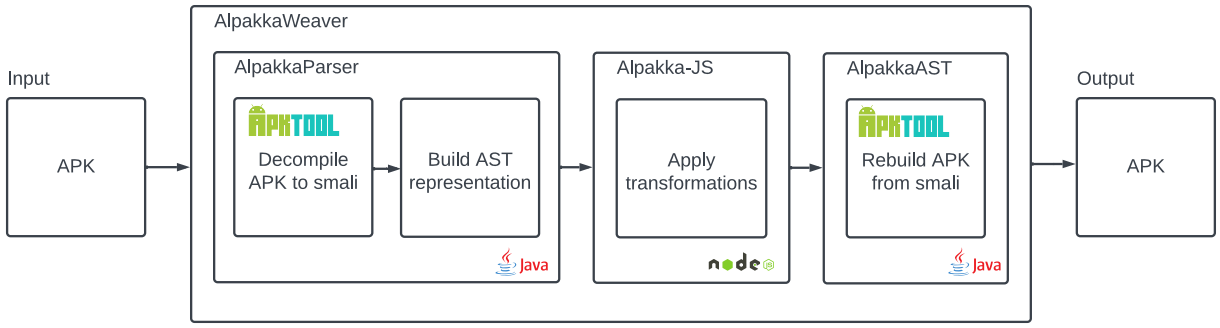

Figure 2. Alpakka's workflow.

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 can be:

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

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. 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 compiler so we can work with the representation of these objects in our scripts. Any transformations applied to the AST will be reflected in the output code it generates and consequently in the output APK, since we rebuild the app back into APK format with Apktool.

5 Detecting and Correcting Resource Leaks

5.1 The Application's Flow

In order to effectively detect resource leaks in Android apps, we need to understand the program's flow. This allows us to identify where resources are acquired and ensure they are properly released. To achieve this, we developed a Control Flow Graph API specifically designed for use with Smali code within Alpakka. This API provides a powerful tool for analyzing the control flow of Smali code, enabling us to perform dataflow analysis and pinpoint potential resource leaks with precision.

Other LARA-based projects have already developed their own Control Flow Graph APIs. Notably, Clava's implementation 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, and instead are represented through combinations of *if* conditions and *goto* statements.

Typically, when executing a Smali program, instructions are executed sequentially, with the following exceptions:

- **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.
- **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: determining whether an instruction within a try block throws an exception. Other tools for building CFGs of Smali code, such as Androguard, adopt a conservative approach and assume 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.

Using Smali's *dexlib2*, which provides specific information about which opcodes can potentially throw exceptions, combined with Google's documentation on Dalvik bytecode [8], 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* as all Android exceptions inherit from this type. This more conservative approach can lead to an overestimation of potential exception paths which may not precisely reflect the actual program behavior, leaving room for future refinement.

A particular challenge arises with certain *invoke* instructions, that utilize dynamic dispatch to determine the method being invoked, meaning the actual method is determined at runtime based on the type of the object, preventing static analysis from determining, in the general case, whether an exception can be thrown. 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 a *Closeable* object present in register *v0*. The precise implementation of the *close* method is determined at runtime based on the actual type of the object assigned to *v0*, which could be an instance of *android.database.Cursor*, *java.io.InputStream*, or any other class implementing the *Closeable* interface.

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 mentioned previously, 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 and tracking resource allocation and release points throughout the application.

5.2 Detecting Resource Leaks

For detecting resource leaks, we designed a JavaScript library for 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, generating a CFG for each method. Next, the application's declared components are 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 our dataflow 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, that can be provided by a user, 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, one simply needs to provide the following comprehensive details:

- Resource Class: The class of the resource we want to analyze.
- Acquisition Information: The classes and corresponding methods used to acquire the resource.
- Release Information: 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`

Part of this information, seen in Table 2, particularly for SQLiteDatabase and Cursor, was initially retrieved from Extending EcoAndroid's [16] project. We supplemented this with additional details on both these resources and InputStream. The rationale behind selecting these specific resources, besides their widespread use in the Android landscape, is that they are also relatively common in the DroidLeaks dataset, 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.

To identify situations where resources have been acquired but not released, the analysis tracks acquired resources, noting the registers in which they are stored, operations where they were last involved, and any associated class fields. For each function we encounter, we perform an interprocedural analysis, visiting each statement, while maintaining a list of visited nodes along with a state object for the resource leaks at the time of each visit. In our strategy, if the control flow loops back to a previously visited statement, we verify whether the state of leaks has changed. If the incoming resource states differ, we merge the incoming resource maps using a conservative join operation and reprocess affected parts of the CFG from that node onward.

When merging resource states, 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 — a post-dominator. 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.

Furthermore, in situations where two distinct resources of the same type are created along divergent paths but utilize the same register, we aim to synchronize their release operations at a common point ahead in the execution path. This approach ensures that all potential leak points are accounted for and that our analysis remains comprehensive.

For each Smali instruction we visit, we apply the following:

- **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

Table 2. The Android resources analyzed with Alpakka and relevant information on each of them.

Resource	Acquisition		Release		Verification	Single	Has
	Class	Methods	Class	Methods	Method	Instance	Dependents
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

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.

- **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 approach, we ensure that we can 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.

5.3 Correcting Resource Leaks

From our resource leak analysis, we derive a list of acquired resources that are yet to be released, organized on a per-class basis. This information provides developers with the necessary insight 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 same information to implement automatic corrections, significantly reducing the time and effort required for manual fixes.

From the predefined knowledge of the resources we are analyzing, we know 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 correcting 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 class 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 a 4-bit address, restricting us to registers v0 through v15. To overcome this challenge, we switched to using the *invoke-static/range* operation, which allows argument registers addresses to be up to 16 bits. However, the *iput-object* operation we were using for saving resources in class fields suffers from the same 4-bit register address 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. Since our priority is to maintain the application's stability and functionality, for the time being, we do not address situations where resources fall into this category.

All modifications are made possible through Alpakka's insertion methods, which enable us to alter the Smali code based on the contents of the AST. This ensures that changes in the AST are reflected in the generated Smali code. The integration and implementation of these insertion methods were significantly streamlined by utilizing the LARA Framework.

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

6 Alpakka's Results

6.1 Evaluation

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, running Windows 11 and Node.js version 20.10.0.

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 work is comprised of 292 manually verified resource leaks in 32 open-source Android apps. Out of these, 189 are of types we currently support in Alpakka, 13 pertaining to *android.database.sqlite.SQLiteDatabase*, 143 associated with *android.database.Cursor* and 33 involving *java.io.InputStream*. However, DroidLeaks does not publicly provide all the APK files used in their work. Among the 189 resource leaks we would be able to support with Alpakka, only 50 are identified in the publicly available dataset, 38 of *Cursor*, 9 of *InputStream* and 3 of type *SQLiteDatabase*.

The resource leaks in the DroidLeaks dataset were originally identified using an automated script that scanned commit messages for keywords related to resource leaks. 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.

With this in mind, we executed our Alpakka script on all the 138 APK files publicly available in the DroidLeaks dataset, in some cases encompassing multiple versions of the same applications, filtering our search to exclude third-party libraries and focus on application-specific code. We were able to detect 93 unique leaks, 32 of which are identified in DroidLeaks, as seen in Table 3.

Table 3. 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 publicly 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 Authenticator[3]. As we rely on analyzing the dataflow of the application, following 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 undetected leaks may be a result of less common undocumented callbacks.

Interestingly, of the 32 detected resource leaks that were also identified in DroidLeaks, only 16 of them were associated with the publicly available APK files. The remaining 16 were linked to different versions of the applications tested, which are not included in the public dataset. However, according to our tool, these leaks were already present in the versions we tested.

Effectively, our automated approach successfully detected 32 of the resource leaks identified in the DroidLeaks dataset. 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 our automated process might miss.

The manual use of our library's *findResourceLeaksInFunction* method provides a viable fallback, with the feasible future possibility of creating an automated intraprocedural

[3] https://github.com/google/google-authenticator/blob/f7dee7574d30fb7f948acdc1ccc9fe2e0fcdc432/mobile/android/src/com/google/android/apps/authenticator/AccountDb.java#L76

analysis 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 estimate is based on a manual review of a sample of these results which revealed that this rate of false positives stemmed primarily 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 some resource leaks remained present in the APK files where they were supposedly manually corrected. This occurred because the applied corrections were not effective in all execution branches of the program, as is the case with the code presented in Figure 1. This emphasizes the need for more comprehensive analysis tools that can ensure resource management is properly addressed.

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

By running our script again on the corrected APK files generated by Alpakka, we verified that our library successfully and automatically corrected 45 resource leaks, as seen in Table 4. 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.

Table 4. Number of resource leaks automatically corrected with Alpakka.

Resources	# Leaks Found	# Leaks Fixed
android.database.Cursor	67	43
java.io.InputStream	7	2
Total	74	45

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 are not addressing for now.

To test Alpakka on more modern Android apps, we executed it on a small set of randomly selected APKs. Since the selection did not include any known resource leaks, our findings were limited. However, an interesting observation emerged: Alpakka identified a reference to a SQLiteDatabase object that was never released in a launcher app[4]. Upon further investigation, it appears this was intentional on the part of the developer, as the database seems to be essential to the core functionality of the app, but it highlights Alpakka's ability to flag potential issues.

6.2 Comparing with others

Comparing the performance of our library with existing resource leak tools that were executed on the DroidLeaks dataset, we can understand its strengths and limitations.

The DroidLeaks work already includes a comparison that we can build upon, on the detection performance of several popular tools, including some that are not specifically aware of Android components' lifecycles.

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.

Of the resource leaks used for this comparison, 50 of them are identified as involving one of the three resource types our Alpakka library supports. 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.

Alpakka's library achieved a bug detection rate of 33.3%, successfully identifying 16 out of 48 experimented leaks. More importantly, Alpakka demonstrated a 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.

Based on PlumbDroid's publicly available information, we estimate it could be applied on 15 bugs present in the DroidLeaks dataset. Of these 15 bugs, PlumbDroid would likely detect approximately 7, yielding a bug detection rate of 47%. Additionally, given the methodology and accuracy described, its false alarm rate would likely be very low.

[4]https://github.com/Neamar/KISS/blob/master/app/src/main/java/fr/neamar/kiss/db/DBHelper.java#L23

Table 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

With these results, we consider Alpakka to be an interesting alternative to existing tools for detecting and correcting resource leaks, striking 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 the presented 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. 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.

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 public dataset of 138 files, 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 application-specific 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 leaks detected.

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. This demonstrates how Alpakka could be used in Android given its versatile foundation, that can be adapted and extended to meet a wide range of research needs.

7 Conclusion

Using Alpakka, our source-to-source compiler for Smali, and leveraging the CFG Alpakka API we developed in this project, we created JavaScript libraries that enable the identification and automatic correction of resource leaks in Android APKs. Alpakka allowed for precise analysis and transformations at the bytecode level without requiring access to the original source code of an Android application.

Our extensive testing on 124 APK files from the DroidLeaks dataset, alongside comparisons with existing solutions, confirm that Alpakka is effective in detecting and rectifying resource leaks. We detected 93 resource leaks, of which we estimate 15% are false positives due to a limitation in our CFG modeling. From these, we successfully applied automatic fixes to 45 of the detected resource leaks. Given its flexibility, we believe Alpakka could be a good fit for other research applications beyond resource leaks, such as employing static analysis for the detection of malware patterns in Android apps or applying performance optimizations in an automated way.

As future work, 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. Additionally, this enriched data would allow us to accurately determine whether resources implement the *Closeable* interface — crucial for generalized detection and correction strategies in 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.

Data Availability

This work is fully reproducible. We provide the complete source code for Alpakka at: https://github.com/specs-feup/alpakka. The Alpakka scripts used to obtain the data and the obtained results are available at: https://doi.org/10.5281/zenodo.14037002. The DroidLeaks dataset is publicly available.

References

[1] AppBrain. 2023. 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. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. *SIGPLAN Not.* 49, 6 (jun 2014), 259–269. doi:10.1145/2666356.2594299

[3] Bhargav Nagaraja Bhatt and Carlo A. Furia. 2022. Automated repair of resource leaks in Android applications. *Journal of Systems and Software* 192 (2022), 111417. doi:10.1016/j.jss.2022.111417

[4] João Bispo and João M.P. Cardoso. 2020. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX* 12 (2020), 100565. doi:10.1016/j.softx.2020.100565

[5] Tiago Carvalho, João Bispo, Pedro Pinto, and João M.P. Cardoso. 2023. A DSL-based runtime adaptivity framework for Java. *SoftwareX* 23 (2023), 101496. doi:10.1016/j.softx.2023.101496

[6] Pedro Correia. 2024. clava-flow. Available at https://github.com/Goncalerta/clava-flow, last accessed in June 2024.

[7] Anthony Desnos. 2024. Androguard. Available at https://github.com/androguard/androguard, last accessed in May 2024.

[8] Google. 2024. Dalvik bytecode format. Available at https://source.android.com/docs/core/runtime/dalvik-bytecode, last accessed in June 2024.

[9] Nelson Gregório, João Bispo, João Paulo Fernandes, and Sérgio Queiroz de Medeiros. 2023. E-APK: Energy pattern detection in decompiled android applications. *Journal of Computer Languages* 76 (2023), 101220. doi:10.1016/j.cola.2023.101220

[10] Ben Gruver and Google. 2024. Smali. Available at https://github.com/google/smali, last accessed in June 2024.

[11] Jierui Liu, Tianyong Wu, Jun Yan, and Jian Zhang. 2016. 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)* (2016), 342–352. https://api.semanticscholar.org/CorpusID:16555098

[12] Yepang Liu, Jue Wang, Lili Wei, Chang Xu, Shing-Chi Cheung, Tianyong Wu, Jun Yan, and Jian Zhang. 2019. DroidLeaks: a comprehensive database of resource leaks in Android apps. *Empirical Software Engineering* 24, 6 (01 Dec 2019), 3435–3483. doi:10.1007/s10664-019-09715-8

[13] Yifei Lu, Minxue Pan, Yu Pei, and Xuandong Li. 2022. Detecting resource utilization bugs induced by variant lifecycles in Android. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 642–653. doi:10.1145/3533767.3534413

[14] Jonathan Meyer and Daniel Reynaud. 2004. Jasmin. Available at https://jasmin.sourceforge.net/, last accessed in October 2024.

[15] Terence Parr and Sam Harwell. 2024. ANTLR. Available at https://www.antlr.org/, last accessed in June 2024.

[16] Ricardo B. Pereira, João F. Ferreira, Alexandra Mendes, and Rui Abreu. 2022. Extending Ecoandroid with Automated Detection of Resource Leaks. In *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems* (Pittsburgh, Pennsylvania) *(MOBILESoft '22)*. Association for Computing Machinery, New York, NY, USA, 17–27. doi:10.1145/3524613.3527815

[17] Pedro Pinto, Tiago Carvalho, João Bispo, Miguel António Ramalho, and João M.P. Cardoso. 2018. Aspect composition for multiple target languages using LARA. *Computer Languages, Systems & Structures* 53 (2018), 1–26. doi:10.1016/j.cl.2017.12.003

[18] Ana Ribeiro, João F. Ferreira, and Alexandra Mendes. 2021. 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)*. 62–69. doi:10.1109/QRS54544.2021.00017

[19] Statista. 2023. 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.

[20] Connor Tumbleson. 2024. Apktool. Available at https://apktool.org/, last accessed in June 2024.

[21] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. 2016. Light-Weight, Inter-Procedural and Callback-Aware Resource Leak Detection for Android Apps. *IEEE Transactions on Software Engineering* 42, 11 (2016), 1054–1076. doi:10.1109/TSE.2016.2547385

[22] Zhiwu Xu, Cheng Wen, and Shengchao Qin. 2018. State-taint analysis for detecting resource bugs. *Science of Computer Programming* 162 (2018), 93–109. doi:10.1016/j.scico.2017.06.010 Special Issue on TASE 2016.