Extending EcoAndroid with Automated Detection of Resource Leaks

Ricardo Filipe Baía Pereira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. João Fernando Peixoto Ferreira

Examination Committee

October 2021

Acknowledgments

First, I would like to give my thanks to Professor João F. Ferreira, for his guidance and support throughout this project.

To all my friends and collegues from Instituto Superior Técnico, for their friendship and immensurable knowledge, which influenced me in choosing this challenging project.

To my *Consílio* friends, for keeping up with me and sharing precious moments over the last years of my life, which I will forever remember.

To Sarita, who, for some reason, has been able to put with my stubborness on a level never before seen. Thank you for being with me.

To all my family, who have always been with me and supported each and every one of my decisions, which allowed me to follow this path and achieve this milestone.

Last but not least, an honorable mentioned to the hard-working folks at Biblioteca Municipal de Alcochete, which has been my second home during this project.

To each and every one of you - you have my deepest thanks.

Abstract

Mobile devices are now more than ever present in our everyday lives. They have multiple hardware components that can enrich user experience, like WiFi, cameras, and GPS. When developing mobile applications that utilize these resources, the developer has to carefully manipulate when to acquire and when to release them. Not managing to do so has an energy impact, causing the application to consume more battery than necessary and, in some cases, causing the resource to not function properly. This problem is known as a resource leak and can affect any Android application that uses hardware components available on the device. To help developers fix this problem, we present an extension EcoAndroid, an Android Studio plugin that improves the energy efficiency of Android applications, with the ability detect resource leaks and present their location in the code to the developer. We implemented our detection on top of Soot, FlowDroid, and Heros, which provide a static-analysis environment capable of processing Android applications and performing inter-procedural analysis with the IFDS framework. It currently supports the detection of four Android resources - Cursor, SQLiteDatabase, Wakelock, and Camera. We evaluated our tool with the DroidLeaks benchmark and compared it with 8 other resource leak detectors. We obtained a precision of 72.5% and a recall of 83.1% on all the leaks detected. Our tool was able to uncover 194 previously unidentified leaks in this benchmark. These results show how our analysis can help developers on discovering resource leaks.

Keywords

Energy Consumption, Static Analysis, Resource Leaks, Android

Resumo

Os dispositivos móveis estão cada vez mais presentes no nosso dia-a-dia. Estes possuem diferentes componentes de *hardware* – como WiFi, cameras, e GPS – que enriquecem a experiência do utilizador. Quando estão a desenvolver aplicações que utilizem estes recursos, os programadores têm que manipular cuidadosamente quando os adquirem e os libertam. Fazer isto incoretamente têm um impacto energético, que faz com que a aplicação consuma mais bateria do que necessário, e podendo causar o incorreto funcionamento do recurso. Este problema é conhecido como fuga de recursos e pode afetar qualquer aplicação Android que utilize componentes de *hardware* e não só. Para ajudar os programadores a solucionar este problema, apresentamos uma extensão para o EcoAndroid, um plugin para o Android Studio, que melhora a eficiência energética de aplicações Android, com a capacidade de detetar autoamticamente fugas de recursos e apresentar a sua localização no código ao programdor. Implementámos esta deteção com o auxílio das *frameworks* Soot, FlowDroid, e Heros, capazes de processar aplicações Android e de realizar análises inter-procedimentais com a *framework* IFDS. A nossa ferramenta suporta atualmente a deteção de quarto recursos de Android – `Cursor`, `SQLiteDatabase`, `Wakelock`, e `Camera`. Avaliamos a nossa análise com *benchmark* DroidLeaks e comparámos com oito ferramentas que detetam fugas de recurso. Obtivemos uma precisão de 72.5% e uma exatidão de 83.1% nas fugas detetadas. Fomos foi ainda capaz de detetar 194 fugas nunca antes identificadas neste *benchmark*. Estes resultados mostram que a nossa análise pode ajudar os programadores a identificarem fugas de recursos.

Palavras Chave

Eficiência Energética; Análise Estática; Fuga de Recursos; Android.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Acronyms

API	Application Programming Interface
APK	Android Application Package
AST	Abstract Syntax Tree
CFG	Control Flow Graph
CSV	Comma Seperated Values
DEX	Dalvik Executable
GUI	Graphical User Interface
IDE	Integrated Development Environment
IFDS	Inter-procedural Finite Distributive Subset
PSI	Program Structure Interface
SDK	Software Development Kit
UI	User Interface

1 Introduction

Contents

Mobile devices are more than ever prevailing in our society. The number of smartphone users in 2020 is estimated to be around 3.8 billion worldwide [4]. Of the two most used operating systems in smartphones, Android is in the top one, with its market share hitting an estimated 85%, followed by iOS with 15% [5]. The market for Android applications has also grown throughout the years, totaling a number of 3 million applications on the Google Play Store [6, 7].

Recent research has been uncovering energy problems and inefficiencies, created by application developers, that decrease the battery life of Android devices [8–10]. Taking action to solve these energy problems and increasing the overall energy efficiency of Android applications can have an impact in user experience. A 2013 study has shown that approximately 18% of the complaints in the Google Play Store were related to energy problems in applications [11].

Another factor that gives rise to new energy problems in mobile applications is the evolution of smartphones. Smartphones have been evolving, and the diversity of sensors they provide have also been growing [12]. The sensors and resources that smartphones possess (e.g. camera, GPS, etc) allow the developers to create applications that interact with them. This interaction between applications and sensors can be handled manually by the developer through the Application Programming Interface (API) provided by Android; however, if not well implemented, this can have huge costs on the battery life of the device [13]. One problem that may arise from this incorrect implementation of resources is known as resource leak, and happens when the developer acquires a resource to be used by the application, but forgets to release it (i.e. turning off the resource). Recent research around resource leaks shows that this problem is prevalent regarding energy and performance in Android devices [13–15], but not always have researchers been able to find resource leaks in applications [16].

1.1 Objectives and Contributions

The main goal of this project is to extend EcoAndroid [17] – an Android Studio plugin – with the ability to automatically detect resource leaks in Android applications. In this work, we also (1) introduce basic notions of the Android framework that are relevant for our work (2) research about the topic of static analysis and some existing techniques that could be applied to our project, and (3) discuss the current research around energy problems in Android applications and existing tools to detect and fix these problems.

Our main contribution translates in the creation of a fully-precise context- and flow-sensitive interprocedural static analysis capable of detecting resource leaks in Android applications, integrated in an IntelliJ plugin. Currently, our analysis supports the detection of four resources: `Cursor`, `SQLiteDatabase`, `Wakelock` and `Camera`. These resources were chosen based on how frequently Android developers use them, and the impact they have on the mobile device if a leak occurs [13].

We evaluated our analysis on DroidLeaks, a publicly available resource leaks benchmark, and managed to detect 203 leaks, where 194 are new and undiscovered leaks. From the 50 experimented leaks of this benchmark, we obtained a bug detection rate of 18% and a false alarm rate of 2%. Regarding all the detected leaks, our tool achieved a precision of 72.5%, a recall of 83.2%, and an F-Score of 77.5%.

Contributions summary. The main contributions achieved from our work can be summarized as follows:

- a fully-precise context- and flow-sensitive inter-procedural static analysis capable of detecting resource leaks in Android applications
- integration of the aforementioned resource leak analysis on two Integrated Development Environment (IDE): IntelliJ and Android Studio
- the extension of the DroidLeaks benchmark, with the addition of 194 new resource leaks identified and described

Illustrative example. Listing 1.1[1] provides an example of a resource leak detected by our tool, in an older version of Wordpress which is part of the DroidLeaks dataset. This resource leak was found by our analysis and was not identified in the benchmark. As we can see in this example, the leaked resource is a cursor, acquired in lines 11-12, which is never released and, therefore, is leaked.

1.2 Organization of the Document

This thesis is organized as follows:

- Chapter 2 (Background and Related Work) explores the work being done in the area of energy problems and patterns of Android applications - from categorization of problems to their detection. This Chapter also provides insight on the basics of the Android framework, and also on the fundamental of static analysis, its techniques, and frameworks used
- Chapter 3 (Design and Architecture) provides an overview of our tool, and shows how its requirements were achieved

[1]Source code at `https://github.com/wordpress-mobile/WordPress-Android/blob/3f6227e2d4c80d9b758928ce4b3d7488ac982e62/src/org/wordpress/android/util/ImageHelper.java`

```java
public static int[] getImageSize(Uri uri, Context context){
    String path = null;
    if (uri.toString().contains("content:")) {
        String[] projection;
        Uri imgPath;

        projection = new String[] { Images.Media._ID, Images.Media.DATA };

        imgPath = uri;

        Cursor cur = context.getContentResolver()
                     .query(imgPath, projection, null, null, null);
        String thumbData = "";

        if (cur.moveToFirst()) {
            int dataColumn;
            dataColumn = cur.getColumnIndex(Images.Media.DATA);
            thumbData = cur.getString(dataColumn);
            path = thumbData;
        }
    } else { // file is not in media library
        path = uri.toString().replace("file://", "");
    }

    BitmapFactory.Options options = new BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    BitmapFactory.decodeFile( path, options);
    int imageHeight = options.outHeight;
    int imageWidth = options.outWidth;
    int[] dimensions = { imageWidth, imageHeight};
    return dimensions;
}
```

Listing 1.1: Cursor leak in an older version of Wordpress

- Chapter 4 (Implementation) describes in detail the implementation of the different components of our tool
- Chapter 5 (Evaluation) shows the results of the evaluation of our analysis on the DroidLeaks dataset
- Chapter 6 (Conclusions) addresses some shortcomings and future work, and summarizes our contributions.

2

Background and Related Work

Contents

In this chapter we start by presenting, in Section 2.1, the basics of the Android architecture to give a simple overview of how the Android operating system works, and to introduce some of the terminology used in the course of this work. We then describe, in Section 2.2 the fundamentals of program analysis and static analysis techniques commonly used for analyzing Android applications. In Section 2.3 present some frameworks that leverage this techniques to analyze Java programs, as well as the frameworks used in our work. Finally, for the remainder of the chapter, we focus on Android energy problems and patterns being researched and how they are being detected, with a focus on resource leaks.

2.1 Android Architecture

Android applications are built upon four essential components [1,18,19]. Figure 2.1 illustrates how these components interact with each other.

1. **Activity**. It represents a screen with a user interface and handles all user interaction.
2. **Service**. Component that runs in the background to perform time intensive operations and work related to remote processes. It does not provide a user interface.
3. **Broadcast Receivers**. Allows an application to receive events from the user or the system.
4. **Content Provider**. Is used to manage shared data between multiple applications.

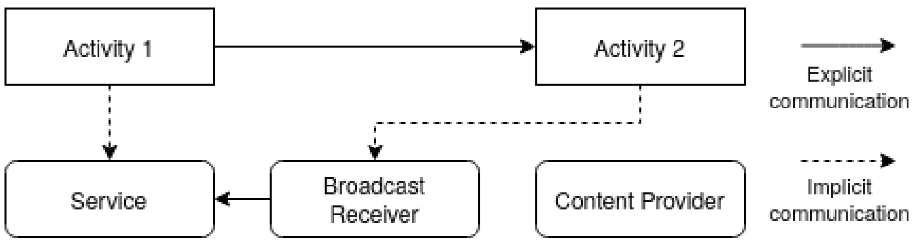


Figure 2.1: Android component communication (adapted from Li et al. [1])

An activity can transition through multiple states as the user interacts with the application and with the system itself. There are four states an activity can go through: running, paused, stopped, and destroyed. The developer has to explicitly program how an activity transitions between these states. This is done using callbacks provided by the Android API: `onCreate()`, `onStart()`, `onResume()`, `onPause()`, `onStop()`, and `onDestroy()` [2,19]. The complete lifecycle and state transitions of an activity are illustrated in Figure 2.2.

The Android system starts a new Linux process when an application component starts and no other component from that application is running. After that, all components from an application run in the

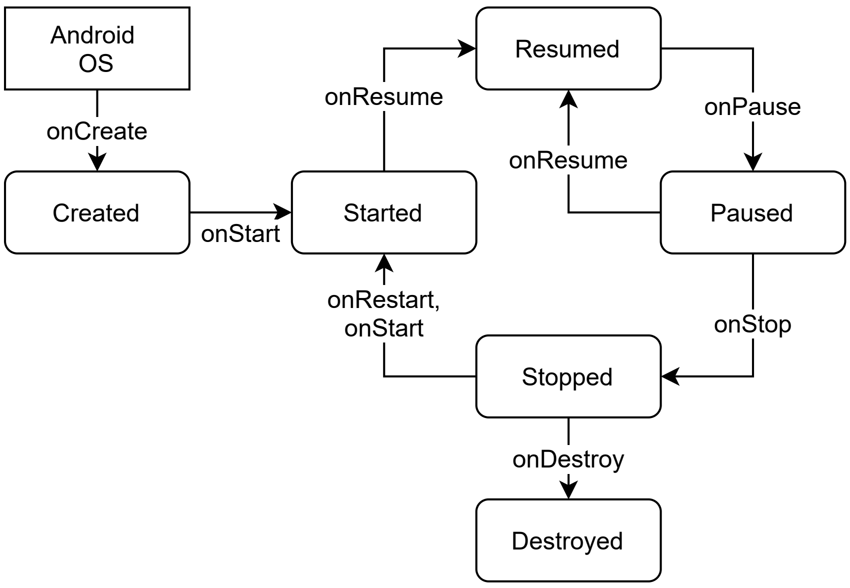


Figure 2.2: Android activity lifecycle (adapted from Android Guide [2] and Android Fundamentals [3])

same process and in the same thread, unless otherwise specified. The thread created when the application is launched is called the main thread. It is responsible for dispatching events to the user interface widgets, and is almost always the thread that interacts with the components from the Android User Interface (UI) toolkit, and so it is often called the UI thread. To avoid blocking the UI thread, as to keep the application responsive, tasks that are not instantaneous should be done using a separate thread [2].

The Android framework is mainly event-driven [20]. Event-based programs make use of callbacks, which are functions that are called after certain events are completed. An example of callbacks are the functions used in the activity lifecycle to transition between states. These functions are called after certain events occur, and are responsible for managing the activity's state. A more specific callbacks are event handlers, which are functions that are executed after a certain event related to the user interaction happens (e.g. the function that executes when a user clicks on a button) [13, 21, 22].

2.2 Static Analysis Fundamentals and Techniques

Li et al. [1] studied and compiled several works of static analysis related to private data leaks, vulnerabilities, and energy consumption in Android applications. This section summarizes some of the findings in their work - focused on energy - and adds insights regarding program analysis [23, 24].

Program analysis – static or dynamic – is typically done using an abstraction of the program. This abstraction differs depending on the type of analysis. One of the main abstractions models used in static analysis of Android applications is control-flow graphs [1]. Variations of this abstraction can be found in the literature – a component call graph is such an example, as it is used to abstract the relationship between components of an Android application [25]. More examples can be found in the works presented in Section 2.6 and Section 2.7.

There are properties and attributes an analysis can have that change depending on its type:

- an analysis is **intra-procedural** when the focus is at the procedure level (i.e. only inside a procedure).
- an analysis is **inter-procedural** when the it takes into account information about the relationship between multiple procedures.
- an analysis is **flow sensitive** if the information discovered at some point in the program depends on the (control-flow) path involving that point.
- an inter-procedural analysis is **context sensitive** if the information discovered regarding a function is different when the calling context of that function is also different.

There are three main static analysis techniques used in static analysis of Android applications that leverage control-flow graphs:

- **Control-flow analysis**: used to find information about what basic block may lead to what basic blocks. More specifically, the order in which instructions are executed in the program.
- **Data-flow analysis**: used to gather information about the runtime behaviour of a program (e.g. if a variable definition can reach a point in the program)
- **Points-to analysis**: used to get information about the data a pointer can point during the program execution.

Regarding energy problems, control- and data-flow analysis are the most used in the literature to solve them [1]. Specific examples of tools and techniques that use these types of analysis are shown later in Sections 2.6 and 2.7.

Besides the aforementioned techniques, there are also six more than can be used to enhance analysis, and that are also used in conjunction in the techniques presented above:

- **Abstract Interpretation**: the program semantics are approximated while ensuring soundness. It makes use of abstract values (i.e. a set of concrete values), flow functions that define the abstract semantic of statements, and an initial state. The flow function receives as input the abstract values as well as an input statement, and outputs the abstract state of the program after executing that statement.

- **Taint analysis**: objects are tainted (i.e. marked) and through data-flow analysis, checked to see if they reach a point that they should not.
- **Symbolic execution**: the program is executed but its input are replaced by symbols. In the end, the result are expressions and constraints composed by those symbols. This can be a way to determine what inputs cause each part of a program to execute, and help detect, for example, infeasible paths.
- **Code instrumentation**: used to help static analysis by addressing some of its problems (e.g. switching reflection calls with normal Java calls and, specific to Android, artificially connecting components for inter-component communication detection).
- **Type checking**: used to ensure the program is type-safe (e.g. float operation applied to a string).
- **Model checking**: used to verify if a system meets a given specification, using an abstraction of the program as a model and expressing its properties as boolean formulas.

2.3 Static Analysis Frameworks

There are several logical and programming frameworks that aid researchers in building static analysis tools for Java programas that can also be used for Android applications. Next we describe, with more detail, the frameworks used in our work.

Soot[1] started as a Java optimization framework [26]. Now it is used by researchers to instrument and analyze Java and Android applications. It works by translating programs into one of four intermediate representations that can later be analyzed. It supports call graph construction, point-to analysis, intra- and inter-procedural data-flow analysis, and taint analysis. For its purposes, Soot has a few data structures to represent objects used in analysis. The `Scene` class is used to represent the environment the analysis will take place in. Through it, the developer can see the application classes, the main class (which contains the application's main method), and access information the analysis (e.g. control-flow graphs, call graphs). The `SootMethod` represents a single method of a class. Classes loaded or created in Soot are represented as a `SootClass`. A `Body` represents a method body, which contain `Units`. `Units` represent statements (e.g. assignment statements, return statements, conditional statements, etc) in the code. A single datum is expressed as a `Value`, which can be locals (`Local`), expressions (`Expr`), and more. A `Local` represents a variable in Soot's intermediate representations. Soot provides four intermediate representations to be used: Baf, Jimple, Grimp, and Shimple. The most used intermediate represenation is Jimple, which is typed and statament based. It has a total of 15 statements, compared

[1] https://soot-oss.github.io/soot

to more than 200 instructions in Java bytecode. This, and other properties not described here, make Jimple more convenient for performing analysis and optimizations. The execution of Soot is divided into phases called packs. The developer can create transformations, which can be registered to a pack that will run it. Transformations are what allow developers to create optimizations, analysis, or even annotate code in an intermediate representation.

FlowDroid is a data-flow analysis tool capable of computing data-flows in Android applications and Java programs [27]. It specializes in tracking the flow of sensitive information through sources and sinks defined by the developer. FlowDroid can be used as a library together with Soot, from which it also depends. When used as a library, FlowDroid also allows Soot to take as input Android applications (as an Android Application Package (APK)), and allows the creation of call graphs with knowledge of the callbacks of the Android framework.

Inter-procedural Finite Distributive Subset (IFDS) [28] is a framework for solving inter-procedural data-flow subset problems. These problems must have distributive flow-functions over finite domains, and the merge operator for two data-flow facts must be the set union. The IFDS framework works by reducing these problems into a graph reachability problem by specializing the inter-procedural Control Flow Graph (CFG) of the program to the analysis being conducted, creating an exploded super graph. Instead of containing one node to represent program statements, the exploded super graph contains multiple nodes to represent the data-flow facts. In the exploded super graph, a node *n* containing a data-flow fact *f* is reachable from a start node if and only if the data-flow fact *f* holds at the node *n*. The flow-functions must be represented as nodes and edges. To express IFDS problems, the user needs to define four different kinds of edges:

- **Call edges**: responsible for passing information from call sites to callees
- **Return edges**: pass information from callees to call sites
- **Call-to-return**: pass information from before a call site to all possible call site's successor statements. Information passing from this edges typically do not concern the callee
- **Normal edges**: for all other statements

Heros is a generic IFDS/IDE solver that can be plugged into existing Java-based analysis frameworks [29]. Connecting Heros to a program analysis framework only requires the user to implement a version of the interprocedural CFG. The authors already provide an implementation for the Soot framework. As per the definitions of the IFDS framework, to specify an IFDS problem in Heros, the user needs to choose a representation for the data-flow facts, and also needs to implement the four flow-functions required by IFDS. In the Heros framework, the flow-functions are the following:

- **getNormalFlowFunction**: to handle normal edges
- **getCallFlowFunction**: to handle call edges
- **getReturnFlowFunction**: to handle return edges
- **getCallToReturnFlowFunction**: to handle call-to-return edges

For the remainder of the section, we present other frameworks also used for static analysis of Java programs.

ASM [2] is a Java bytecode engineering library used to generate, transform, and analyze compiled Java classes. It enables both data- and control-flow analysis.

The T. J. Watson Libraries for Analysis (WALA) [3] is a framework that provides static analysis capabilities for Java bytecode. Some of its features include Java type system and class hierarchy analysis, general analysis utilities and data structures, a bytecode instrumentation library and, like Soot, it features a framework for iterative data-flow, pointer analysis, call graph construction and inter-procedural data-flow analysis.

Androguard [4] is a Python 3 tool and library that can be used to parse and decompile APK, Dalvik Executable (DEX), and other files from Android projects. Ĩt also provides API to perform some static analysis.

APKtool [5] is a Java tool that allows reverse engineering of Android applications. Apktool is also capable of rebuilding decoded applications back to APK.

Android Lint [6] is a static analysis tool that detects potential problems and optimizations in Android projects. Developers can also extend Android Lint with custom analysis of their own. The tool is included in Android Studio.

2.4 Android Energy Problems and Patterns

There are several works in the literature that aim to define energy-related patterns to help developers and researchers in their work.

Cruz and Abreu [30] define 22 energy patterns for Android applications. The detection of 5 of these patterns (i.e. Dynamic Retry Delay, Push Over Poll, Reduce Size, Cache, and Avoid Extraneous Graphics and Animations) is already implemented in the current version of EcoAndroid.

[2] https://asm.ow2.io/index.html
[3] http://wala.sourceforge.net/wiki/index.php/Main_Page
[4] https://github.com/androguard/androguard
[5] https://ibotpeaches.github.io/Apktool/
[6] http://android-doc.github.io/tools/help/lint.html

Jiang et al. [25] list typical energy bugs, divided into resource leaks and layout defects. Resource leaks bugs (also called no-sleep bugs) refer to when some sensors or wakelocks are acquired, but never released. Layout defects are related to how the layout of the activities is constructed.

Pathak and Jindal [31] specify no-sleep bugs into three categories: no-sleep code path (i.e. when there is a code path that acquires a component wakelock, but never releases), no-sleep race condition (i.e. when the power management of a particular component was carried out by different threads in the application), and finally no-sleep dilation (i.e. when a component is put to sleep, but after a long period of time than necessary).

Guo et al. [21] categorize resource leak bugs, but taking into account the type of the resource and the Android API reference. It separates them into 3 categories: exclusive resources (e.g. camera), memory-consuming resources (e.g. media player), and SensorManager releasing.

Liu et al. [22] categorize common bug patterns regarding performance of Android applications while stating that some also cause energy problems. Such example of this is wasted computations for Graphical User Interface (GUI) (i.e. when an Android application switched to the background, but continues to update its GUI).

Le Goaër [32] describes 11 Android energy bugs alongside with their severity. Some of them are already presented here (e.g. sensor leak) but new ones are described (e.g. everlasting service).

Riganelli et al. [33] expand on the works of Liu et al. [13], Banerjee et al. [34], and Wu et al. [35] to create a resource leak benchmark to help developers and researchers. The benchmark contains 40 reproducible resource leaks, each one containing information about the source code location of the resource leak, information about the faulty and fixed application, and a brief execution summary of the test cases the authors generated.

Similarly to Riganelli et al. [33], Liu et al. [13] create a database of 292 leaks from 33 different resource classes, contained in some Android applications. They categorize the resource leaks into 2 categories: Android platform resources, and Java platform resources, providing examples on how to acquire and release them.

2.5 Resource Leaks

As introduced in Chapter 1, the number of sensors and hardware components in mobile devices has been growing over the years. These components – also called resources – are known for being one of the biggest energy consumers in Android devices [35]. When a developer wants to use a resource, they must do it manually. This is done via Android-specific API calls, which vary from resource to resource [13]. Here, we show an example from an older version of AnkiDroid[7]. In Listing 2.1, we see a

[7] https://github.com/ankidroid/Anki-Android

```java
private static SQLiteDatabase upgradeDB(...) {
    (...)
    Cursor c = mMetaDb.rawQuery(...);
    int columnNumber = c.getCount();
    if (columnNumber > 0) {
        if (columnNumber < 7) {
            (...)
        }
    } else {
        mMetaDb.execSQL(...);
    }
    mMetaDb.setVersion(databaseVersion);
    Timber.i(...);
    //leak! missing call to c.close()
    return mMetaDb;
}
```

Listing 2.1: Resource leak of a database cursor on an old version of AnkiDroid

resource – in this case, a database cursor – being acquired at the beggining of a function. The developer performs some operations but, at the end of the function, forgets to close the database cursor, creating a resource leak [8]. A resource leak occurs when a programmer forgets to release a resource they previously acquired, after it is done being used. In Listing 2.1 the leak no longer exists, as the developer now uses a try-catch to perform the operations and releases the resource only if it was used [9]. A resource leak causes components to stay active and consume battery, even if they are not being used. Apart from the unnecessary battery usage, the leak of some resources may cause them to not function properly for other applications or even cause the Android system to crash [13, 35].

2.6 Detection of Android Energy Problems

The works present below focus on the the detection of several Android energy problems, although some of them also touch on the detection of problems related to resource leaks.

Liu et al. [22] develop **PerfChecker**, a tool capable of detecting two performance bugs (i.e. lengthy operations in main thread, and violation of View Holder pattern). This detection is done using a class hierarchy analysis to create checkpoints and then building a call graph for each of them and check if the checkpoint transitively invokes heavy API or if a rule is violated, respectively for each bug.

Wang et al. [10] focus on searching for energy optimizations regarding I/O operations. This is done by searching for specific I/O patterns that are poorly optimized (i.e. periodic I/O operations that could

[8]Commit at https://github.com/ankidroid/Anki-Android/commit/3725ce75828aaf4fa0b7bc36416a973f2ea6a157
[9]Commit at https://github.com/ankidroid/Anki-Android/commit/c993a3398ec325672fd43307d8e2b457b3be9db7

```
private static SQLiteDatabase upgradeDB(...) {
    (...)
    Cursor c = null;
    try {
        c = mMetaDb.rawQuery(...);
        int columnNumber = c.getCount();
        if (columnNumber > 0) {
            if (columnNumber < 7) {
                (...)
            }
        } else {
            mMetaDb.execSQL(...);
        }
        mMetaDb.setVersion(...);
        Timber.i(...);
        return mMetaDb;
    } finally {
        if (c != null) {
            c.close();
        }
    }
}
```

Listing 2.2: Resource leak fix of Listing 2.1

be closer together) and then building a flow graph between the two I/O operations. Lastly, data-flow analysis is performed on this graph to check if it is possible to optimize the I/O operations.

Li et al. [36] explore how to optimize HTTP requests in Android applications. They employ intra- and inter- procedural analysis on dominator trees – computed from control-flow graphs from Soot – to search for sequential HTTP request sessions that could be bundled together to save energy.

Couto et al. [16] focus on automatic refactoring 11 Android energy-inefficient patterns. They use Android Lint to detect the patterns, and AutoRefactor to automatic refactor some of them. They successfully detect and refactor 3 resource leaks regarding the sensor, camera, and media, but in the set of analyzed applications, there were no resource leaks found.

Cruz et al. [37, 38] develop a tool called **Leafactor** to automatic refactor 5 Android energy patterns (i.e. Draw Allocation, Wake Lock, Recycle, Obsolete Layout Parameter, and View Holder) using AutoRefactor. This tool is integrated in the Eclipse IDE.

2.7 Detection of Resource Leaks

In this section we present some works that focus exclusively on the detection of resource leaks.

Liu et al. [39] create a technique called **Elite** capable of detecting common wakelock misuses. This is done in two steps: first decompiling the application's APK files to Java bytecode using Dex2jar, and then performing an analysis with the help of Soot and Apache Byte Code Engineering Library (BCEL). This analysis is done by locating the application entry points, and then traversing the call graph of each entry point to see if wakelock acquiring and releasing calls can be transitively reached.

Guo et al. [21] create **Relda**, a tool that detects resource leaks. Relda uses Androguard to translate the application APK into Dalvik bytecode. The bytecode is then traversed in sequential order to build the control-flow graph of the application. To find resource leaks, an algorithm that uses depth-first search is run, producing a resource summary.

Wu et al. [15] develop a tool called **Relda2** (the successor of the aforementioned Relda [21]) capable of detecting resource leaks. Unlike most tools that are built on top of frameworks like Soot and WALA, Relda2 analyzes Dalvik bytecode directly, leveraging only Androguard to disassemble the app into the Dalvik bytecode. It first preprocesses the application and builds a function call graph to perform inter-procedural analysis which can be flow-sensitive or flow-insensitive.

Vekris et al. [19] create a tool to verify if an Android application complies with a set of energy policies, focused on the acquiring and releasing of wakelocks. The analysis is done by using inter-procedural data-flow analysis from WALA on a control-flow graph that has the notion of the Android lifecycle in it.

Jiang et al. [25] build a tool called **SAAD** capable of detecting energy bugs. They use Apktool to transform the APK file into Dalvik bytecode, using then SAAF, an analysis framework, to search for resource leaks, and Android Lint, to search for layout defects. For research leak detection, they apply intra- and inter-procedural analysis on a component call graph of the application.

The **Automated Android Energy-Efficiency InspectiON (AEON)** [40] is an IntelliJ IDEA plugin capable of inspecting energy problems related to the Android API. The plugin is able to detect resource leaks, mainly focusing on wakelocks. It is also capable of estimating the energy consumption of methods and has integration with Trepn profiler. AEON was used in the work of Deng et al. [41] to design the WakeLock Release Deletion mutation operator, used to mimic an energy bug.

Bhatt and Furia [14] implement **PlumbDroid**, a tool capable of detecting and repairing resource leaks. It works by building several resource-flow graphs – an abstraction based on control-flow graphs – that captures information about the acquiring and releasing of resources. The underlying control-flow graphs are built using Androguard. The tool performs intra-procedural analysis using pushdown automatons, and inter-procedural analysis by combining the results of the intra-procedural analysis. In a final stage, it fixes the resource leak by injecting the corresponding release operation in a suitable location.

Wu et al. [20] aim to detect two patterns related to resource leaks – activity adding a listener but not removing, and activity adding a listener but putting it in a long-wait state – using a stack-based approach. They first build a inter-procedural control-flow graph, in order to create candidate paths that can represent the aforementioned patterns. In a last phase, they take the candidate paths and search for sequences where a leak is present.

Banerjee et al. [34] expand on their previous work [9] and create a framework for detecting resource leaks and implement it into an Eclipse plugin called **EnergyPatch**. The detection is done by building an event flow graph and applying abstract interpretations techniques. After detection, a reduction of the search space (i.e. removing the event flow graph nodes that are not in acquire-release paths) is performed. In the last phase of the analysis, the detected resource leaks are repaired by generating the resource release expression and inserting it into an existing activity.

3

Design and Architecture

Contents

In this chapter, we give an overview of our tool and discuss the design choices made and its architecture. Section 3.2 goes through the functional and non-functional requirements of the tool. In Section 3.3, we present information regarding the several components, and explain how our extension operates.

3.1 Overview

EcoAndroid is an Android Studio plugin capable of automatically detecting and refactoring energy patterns in Android applications [17]. It is built upon IntelliJ Platform Software Development Kit (SDK), which provides developers with the tools needed to build plugins for the InteliiJ IDEA. The IntelliJ Platform SDK also allows plugins to be used by IDE based on IntelliJ IDEA, where Android Studio is one such example. One of the tools IntelliJ SDK provides is the Program Structure Interface (PSI), which is used extensively by EcoAndroid to detect and refactor energy patterns. The PSI is responsible for parsing files and creating semantic and syntactic code model of projects. It also allows the manipulation of PSI files, which represent the hierarchy of PSI elements (e.g. a PSI file contains multiple PSI elements, which can represent variables, identifiers, methods, etc.). This allows EcoAndroid to detect energy patterns and create their respective fixes directly in the code. Figure 3.1 shows the current process, from beginning to end, of detection and refactoring of energy patterns.

The proposed work extends EcoAndroid in order to automatically detect resource leaks in Android applications, and is built upon some of the existing features of the plugin, while integrating static analysis frameworks required for the detection. The extension is fully compatible with energy pattern detection, which remains fully functional. The automated detection of resource leaks is divided into two main components – the Analysis Component and the Results Component – each one responsible for a specific step in the detection of resource leaks.

3.2 Requirements

In this section, we define the functional and non-function requirements for our work. These requirements follow the previous guidelines set for the EcoAndroid plugin.

3.2.1 Functional Requirements

Static analysis. The extension must use static analysis techniques to detect resource leaks. This is one of the main requirements of the extension. As seen from current research, detecting resource leaks requires the use of static analysis techniques.

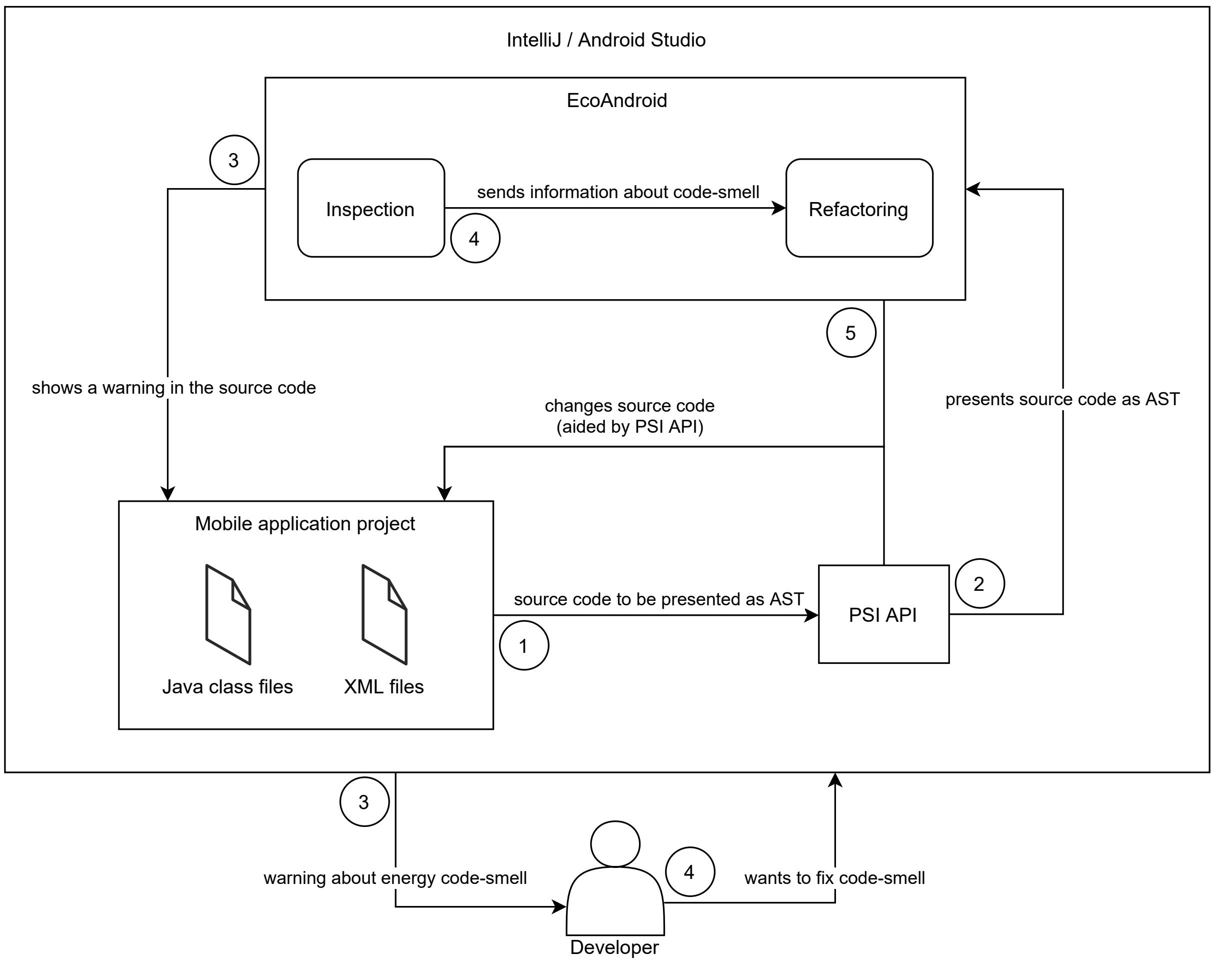

IntelliJ / Android Studio
EcoAndroid
3
Inspection
sends information about code-smell
4
Refactoring
5
shows a warning in the source code
changes source code (aided by PSI API)
presents source code as AST
Mobile application project
Java class files
XML files
source code to be presented as AST
1
PSI API
2
3
warning about energy code-smell
4
wants to fix code-smell
Developer

Figure 3.1: EcoAndroid detection and refactoring of energy patterns

Language elements. The extension must process elements from the Java language and the Android framework. This is essential to perform static analysis and is offered by open-source static analysis frameworks and the API provided by IntelliJ IDEA.

Report analysis results. The extension must report the analysis results to the user. This is a requirements of EcoAndroid, which our extension must also follow. Going even further, this must be done in the same way done in the current version of EcoAndroid: displaying warnings in the source code, specifically where the resource was leaked.

3.2.2 Non-Functional Requirements

Integration. The extension shall be consistently integrated with the current version of EcoAndroid and IntelliJ IDEA. As previously said, the extension shall uphold the design choices made in the current version of EcoAndroid, providing a consistent user experience in every aspect of the plugin.

Extensible. The extension shall ease the implementation of the detection of other resources. This is an important requirement, as the Android framework is continuously updated, as new resources are added and old ones are changed.

User control. The extension shall give the user total control over their actions. The user shall have the ability to start, check the analysis progress, and cancel the analysis at any point in execution. This is important specially because this type of analysis may take some time.

Standalone version. The extension shall be able to run as an independent application. This provides an easy way to run the analysis on multiple projects and applications, which allows automated evaluation and testing of the analysis.

3.3 Architecture

In this section, we will expand on how the different components of our work were structured and designed, while discussing the different decisions made during this process.

3.3.1 Analysis Component

The Analysis component is one of the the two main components of our tool. It is responsible for creating and setting up the environment for the analysis, and is also responsible for running the analysis itself.

Our work focus is the design and implementation of an inter-procedural resource leak analysis. There are multiple ways of approaching this task, as seen in Chapter 2. With the time constraints for the development of our work, we decided to use existing static-analysis frameworks. During our research, we verified that the Soot ecosystem provided all the necessary tools for us to implement our analysis. As mentioned in Section 2.3, the Soot framework enables the transformation and analysis of Java applications. Since we intend to analyze Android applications, we require a framework that can parse these applications and also create the necessary abstractions with information of how the Android framework works (see Section 2.1). FlowDroid satisfies these two needs, although it can only parse Android APK. Lastly, we want to be create an inter-procedural analysis. Soot alone does not have the necessary means for this, and although FlowDroid implements a data-flow analysis, this analysis does not satisfy our requirements. For these reasons we chose Heros, another framework from the Soot ecosystem, that enables the creation of inter-procedural analysis through the IFDS framework. We also explored the VASCO framework and created an inter-procedural resource leak analysis with it; however, we were unable to model our problem to be compatible with the restrictions imposed by the VASCO framework – specifically, that the flow functions had to be monotonic.

The Analysis component was designed to accommodate different types of resource leak analysis. As previously said, the focus of this work is on the creation of an inter-procedural resource leak analysis, but we have also implemented a simpler, intra-procedural analysis. The intra-procedural analysis was implemented to understand how the Soot framework works before deciding on how to implement the inter-procedural analysis. In the current version of our extension, due to its simplicity, the intra-procedural analysis is disabled.

Summary. The Analysis component handles everything related to the analysis, from setting up the environment to running the analysis itself. This is enabled by Soot, FlowDroid, and Heros, three static-analysis framework that meet our requirements and demands.

3.3.2 Results Component

The Results component is the other main component of our tool. It is responsible for acquiring the results at the end of the analysis and then, from these results, collect the location of possible leaks, process them, and present the final results to the user.

As we support multiple analysis, the Results component is designed to be able to acquire results from the implemented analysis, and process them accordingly. In the case of the inter-procedural analysis, to acquire its results, we use Soot and Heros. Heros' IFDS solver provides the necessary method for this task. As for the intra-procedural analysis, we simply implemented it to allow retrieving its results. The Analysis and the Results component are then designed in a way that allows the Results component to visit each analysis after they terminated, to acquire the results.

Another aspect to take into consideration when designing the Results component is that our tool can be used in EcoAndroid or in the command line as a standalone tool. The Results component must take into account the different requirements for each version, since the results are presented differently in each one. This means that each version of the tool has their own leak storage, which allows the reported leaks to be presented accordingly. The Results component is responsible for, after acquiring the leaks and processing them, storing them correctly depending on the version of the tool being run. For the EcoAndroid version, the Results component requires the use of the PSI API to correctly assign each leak location to the corresponding method in the source code of the application. The standalone version does not require the PSI API, as leaks are simply stored to be stored in `.csv` files.

Summary. The Results component handles everything related to acquiring, processing, and storing the results from the analysis – which depends on the analysis and version of the tool being run.

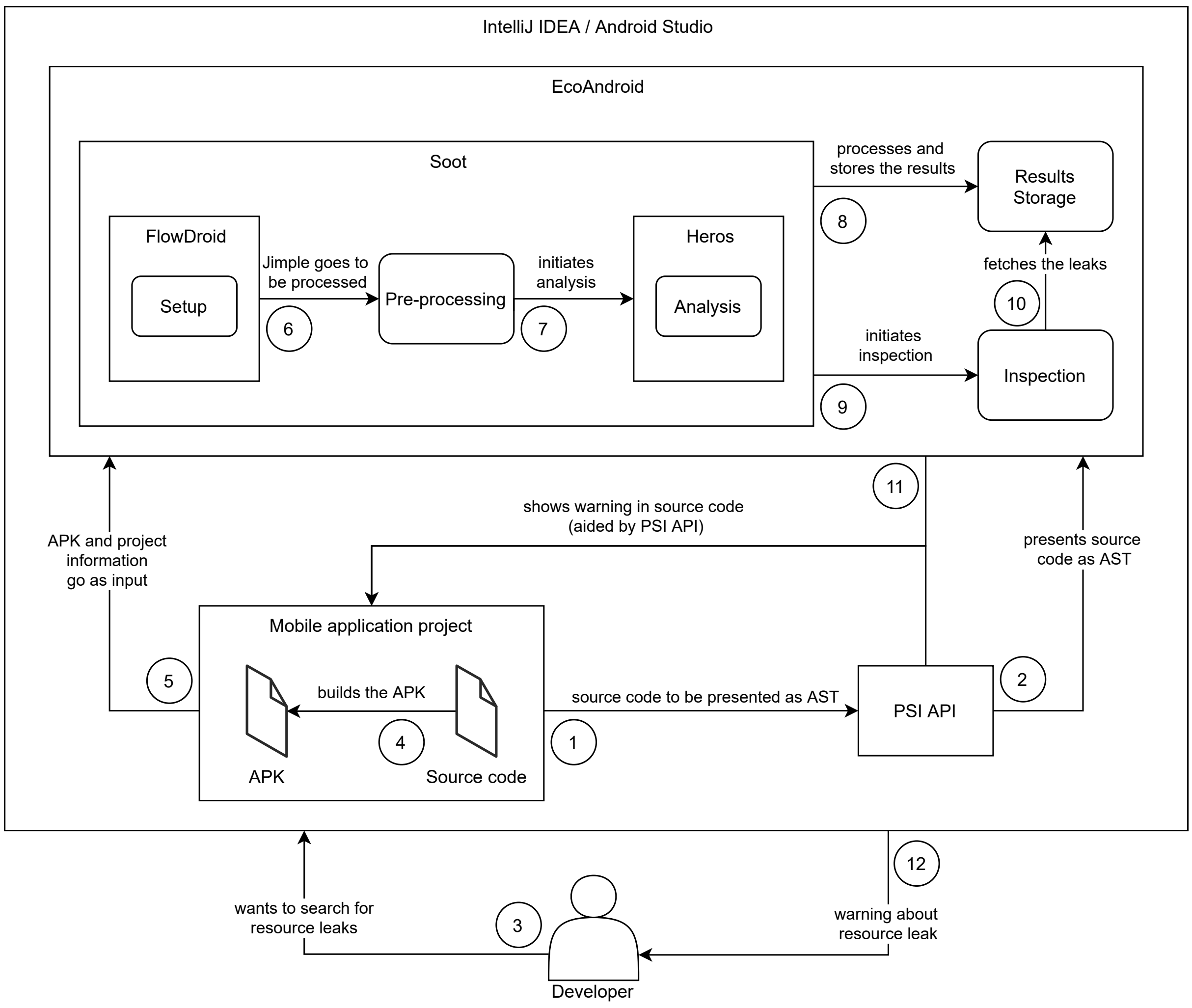
3.3.3 Other Components

Besides these two main components described in this chapter, there are two auxiliary and essential components that ensure the integration with the IntelliJ IDEA and Android Studio: the Platform component and the UI component.

The Platform component handles the required verification to ensure the analysis is run either on IntelliJ IDEA or Android Studio. It is also responsible for locating the APK for the analysis input, and locating the Android SDK, also required for the analysis. For this purpose, the PSI API is needed and used.

The UI component generates error and information messages for the analysis, and also handles the creation of line markers in source code location where a leak is present. It works by using the available functionalities of the PSI API regarding UI and line markers.

These two components are heavily inspired in the IntelliJ version of CogniCrypt [42].



IntelliJ IDEA / Android Studio
EcoAndroid
Soot
FlowDroid
Setup
Jimple goes to be processed
6
Pre-processing
initiates analysis
7
Heros
Analysis
processes and stores the results
8
Results Storage
fetches the leaks
10
initiates inspection
9
Inspection
11
shows warning in source code (aided by PSI API)
APK and project information go as input
presents source code as AST
Mobile application project
5
builds the APK
4
APK
Source code
source code to be presented as AST
1
PSI API
2
12
wants to search for resource leaks
3
warning about resource leak
Developer

Figure 3.2: EcoAndroid detection of resource leaks

3.3.4 User Interaction and Operation

The current EcoAndroid interaction process and operation of the plugin is unaffected by our extension, remaining unaltered and fully functional, and is described in Figure 3.1. The new functionality brought by our extension adds more complexity to EcoAndroid. This translates in a new, longer, and more detailed user interaction process and plugin operation. Figure 3.2 shows this new user interaction process and operation. The plugin starts the process by transforming the source code into an Abstract Syntax Tree (AST), aided by the PSI API (① and ②). The developer, who wants to detect resource leaks, must compile the application into an APK, which will be used as input for the analysis, together with information related to the project and the location of the Android SDK (③, ④, and ⑤). Now, everything is set for the analysis to start. First, FlowDroid is in charge of setting up the environment and Soot's `Scene`, by loading the application's classes, building the call graph and choosing Soot's options. Then, Jimple is ready to be pre-processed, and the analysis can start, with Heros in charge (⑥ and ⑦). After the analysis is completed, the plugin processes and stores the results (⑧). When the results are processed and stored, the plugin can initiate the inspection. The inspection will go through the AST and match any methods that contain resource leaks (⑨ and ⑩). Finally, aided by the PSI API, the plugin highlights the leaks' location in the source code, and the user is warned about any possible resource leaks (⑪ and ⑫).

The operation of the standalone version of our tool is much simpler. Using a Gradle command, the user inputs the APK and the location of the Android SDK, which is sent to Soot (④ and ⑤). Then, the process follows the same flow until the results storage (⑥, ⑦, and ⑧), where it stops and outputs the reported leaks to `.csv` files.

4 Implementation

Contents

In this chapter we show, in detail, how the multiple components presented in Section 3.3 were developed and how they interact with each other. Section 4.1 describes the data structure used to represent Android resources. Section 4.2 and Section 4.3 explains how the Analysis and Results component were implemented in our tool. Finally, Section 4.4 shows how our analysis is integrated in IntelliJ IDEA and Android Studio.

4.1 Resource Representation

The resource leak detection extension for EcoAndroid supports the detection of four different Android resources. The resources chosen for our detection are presented in Table 4.1. Figure 4.1 shows the class diagram of the resource representation. To represent an Android resource and its proprieties, we use an `Enum` containing the following fields:

- `type` - A `String` containing the fully-qualified class name of the resource
- `acquireOp` - An `Array` of `String` containing the name of the methods used to acquire the resource
- `acquireClass` - An `Array` of `String` containing the fully-qualified name of the class where the `acquireOp` can be used
- `releaseOp` - An `Array` of `String` containing the name of the methods used to release the resource
- `releaseClass` - An `Array` of `String` containing the fully-qualified name of the class where the `releaseOp` can be used
- `heldCheckOp` - A `String` containing the name of the method used to check if the resource is acquired (if this method does not exist, the placeholder `#NONE` [1] is used)
- `placeToRelease` - A `String` containing the name of the callback method where the resource is supposed to be released (if there is no such callback, the placeholder `#NONE` [1] is used)

Resource name	Resource class
Cursor	android.database.Cursor
SQLite Database	android.database.sqlite.SQLiteDatabase
Wakelock	android.os.PowerManager.WakeLock
Camera	android.hardware.Camera

Table 4.1: Resources detection by our extension

[1] Using # as the first chracter of the placholder prevents collisions with any Java method.

Representing a resource with this structure allows any interested developer to easily add another resource to the analysis. The main restriction is that the resource respects the contract established by our representation, which means it must abide by the acquire-release pattern, i.e. must possess methods for opening and closing, and this mechanism must be done manually by the developer using the resource.

While it is easy to create a new representation for a resource, further steps must be done to ensure its correct implementation in the analysis:

Step 1 The developer must check the Android documentation and learn the resource's respective acquire and release methods. They also need to check if there is a method for checking if the resource is being acquired, and, for class-scope resources, if there is a suggested place to release the resource.

Step 2 Examine how Soot translates the resource usage in Jimple and extract needed information about the resource. This means, for example, debugging and inspecting the `Body` and its `Units` of the `SootMethod` where the resource is being used

Step 3 Create a new instance in the `Enum` with the newly acquired information about the resource

Step 4 (optional) Modify the analysis and/or the processing of the results to handle new mechanisms specific to the resource being added

We will use the Cursor as a follow-along example of this process. Examining the Jimple translation in Listing 4.2, we have an example of how a cursor can be acquired: In this example, we see an `IdentityStmt` assigning the first method parameter, a `ContentResolver`, to the `$r1` local. Then, we see a `AssignStmt` where the `query` method is being invoked on the previously assigned `ContentResolver`. The result of this method invocation is an instance of a `Cursor`, which is being assigned to `$r6`. With this information, we are able to discern that a `Cursor` can be acquired with the `query` method on a `ContentResolver` instance. Using this methodology, a developer can learn how a resource is acquired and released in the Jimple representation, providing the necessary information to implement the resource in the analysis.

4.1.1 Cursor

A Cursor provides read-write access to the result returned by a database query [43]. It can be acquired via a database our via a content provider. A cursor works, typically, by acquiring and releasing an instance of it. However, as the Android framework evolves, new mechanisms to facilitate their usage have appeared. The `ContentQueryMap` is one such mechanism, and is used to cache the contents of a cursor into a map. It works by passing the cursor to the `ContentQueryMap` constructor, performing all

```java
public static String getContactName(final Context context, final String address) {
  //(...)
  Cursor cursor = getContact(context, address);
  //(...)
  return cursor.getString(ContactsWrapper.FILTER_INDEX_NAME);
}

public Cursor getContact(final ContentResolver cr, final String number) {
  //(...)
  final Uri uri = Uri.withAppendedPath(Contacts.Phones.CONTENT_FILTER_URL, n);
  final Cursor c = cr.query(uri, PROJECTION_FILTER, null, null, null);
  //(...)
  return c;
}
```

Listing 4.1: Resource leak (simplified) of an older version of SMSDroid

```
public Cursor getContact(final ContentResolver cr, final String number) {
    $r1 := @parameter0: android.content.ContentResolver
    (...)
    $r6 = virtualinvoke $r1.<android.content.ContentResolver:
            android.database.Cursor query(...)>($r3, $r5, null, null, null)
    (...)
}
```

Listing 4.2: Snippet of Jimple code from the `getContact` method seen in Listing 4.1

the operations needed, and then closing the `ContentQueryMap` [44]. This and other mechanisms add new logic to how a cursor works, which becomes harder to support in our analysis. We only support traditional (i.e. acquiring and releasing the `Cursor` resource) use of cursors. Table 4.2 shows how we represent it in our tool.

Field name	Field content
`type`	android.database.Cursor
`acquireOp`	rawQuery, query
`acquireClass`	android.database.sqlite.SQLiteDatabase, android.content.ContentResolver
`releaseOp`	close
`releaseClass`	android.database.Cursor
`heldCheckOp`	isClosed
`placeToRelease`	#NONE

Table 4.2: Cursor information

4.1.2 SQLite Database

The SQLite Database resource is used to manage SQLite databases. Like the Cursor, the SQLite Database also has additional mechanisms designed to ease its use by the developers [45]. One such mechanism is the SQLiteOpenHelper, which helps manage database creation and version control. The `SQLiteOpenHelper` encapsulates the methods used to acquire and release an SQLite Database, which interfere with how Soot creates the calllgraph and interfere with how our analysis works. We only support traditional use (i.e. acquiring and releasing the `SQLiteDatabase` resource) of SQLite databases. Table 4.3 shows how we represent it in our tool.

Field name	Field content
`type`	android.database.sqlite.SQLiteDatabase
`acquireOp`	getWritableDatabase, getReadableDatabase
`acquireClass`	android.database.sqlite.SQLiteOpenHelper
`releaseOp`	close
`releaseClass`	android.database.sqlite.SQLiteClosable, android.database.sqlite.SQLiteOpenHelper, android.database.sqlite.SQLiteDatabase
`heldCheckOp`	#NONE
`placeToRelease`	#NONE

Table 4.3: SQLite Database information

4.1.3 Wakelock

The Wakelock is a mechanism used to indicate to the application that the device needs to stay on. It is usually used when performing critical operations to keep the device from going to sleep. It is acquired through the `PowerManager`. The wakelock can also be acquired with a timeout, in which the wakelock is automatically released after the timeout has passed [46]. Our detection does not support this mechanism. Table 4.4 shows how we represent the Wakelock in our tool.

Field name	Field content
`type`	android.os.PowerManager$WakeLock
`acquireOp`	acquire
`acquireClass`	android.os.PowerManager$WakeLock
`releaseOp`	release
`releaseClass`	android.os.PowerManager$WakeLock
`heldCheckOp`	isHeld
`placeToRelease`	onPause

Table 4.4: Wakelock information

4.1.4 Camera

The Camera resource manage operations related to the camera, which include setting image capture settings, start/stop preview, snap pictures, and retrieve frames for encoding for video [47]. Table 4.5 shows how we represent the Camera in our tool. We support the Camera resource up until the API level 21 of the Android framework.

Field name	Field content
`type`	android.hardware.Camera
`acquireOp`	lock, open, startFaceDectection, startPreview
`acquireClass`	android.hardware.Camera
`releaseOp`	unlock, close, stopFaceDetection, stopPreview, release
`releaseClass`	android.hardware.Camera
`heldCheckOp`	#NONE
`placeToRelease`	surfaceDestroyed

Table 4.5: Camera information

4.2 Analysis Component

As mentioned in Section 3.3.1, the Analysis component is built upon Soot, FlowDroid, and Heros. These three frameworks are built to be easily integrated with each other, as they are maintained by the same group of developers. To connect Heros to a program analysis framework only requires the user to implement a version of the interprocedural CFG. The framework's authors already provide an implementation for the Soot framework. As mentioned in Section 2.3, Heros implements a solver for the IFDS framework, and provides the following functions, which we need to implement to create our analysis:

- `getNormalFlowFunction`, which handles normal edges
- `getCallFlowFunction`, which handles call edges
- `getReturnFlowFunction`, which handles return edges
- `getCallToReturnFlowFunction`, which handles call-to-return edges

Each flow-function serves a different purpose for the IFDS framework. Their implementation reflects this fact. Next, we describe the implementation of these functions in our work.

getNormalFlowFunction. The main goals of the `getNormalFunction` is to handle acquiring and releasing class-scope resources and to handle the flow of data-flow facts when dealing with `if` statements.

We use a trick to known when we are before a class-scope resource. Typically, and from what we have seen, class-scope resources are declared in Jimple as `SootField`. When Soot translates the source code to Jimple, these types of resources must be always assigned to a local before calling methods on them. When this happens, under the correct conditions, we know we are dealing with an operation on a class-scope resource. After knowing this, we only need to check if are dealing with an acquire (or release) operation, and if so, we create (or eliminate) the data-flow fact.

For a correct flow of facts about resources, we must take into consideration `if` statements. A correct usage of resources uses null checks and held checks (i.e. checking, via a method, if the resource is acquired) in `if` statements – if a resource is null or not acquired, it does not make sense to branch and release it. For this same reason, if we are before a `if` statements that branches if a resource is null or not acquired, we must kill the data-flow fact regarding that resource, if the fact exists. For the best of our knowledge, our analysis takes into account all the possible cases regarding null checks and held checks.

getCallFlowFunction. This function is responsible for handling flow of facts when a method is called. When resources are passed as arguments to methods, we must update which local refers to the resource, because in Jimple different methods assign different locals to the same resource.

This function only looks at method-scope resources. Class-scope resources do not follow the same logic, and can be used freely throughout the methods in the class where the resource is declared.

getReturnFlowFunction. The `getReturnFlowFunction` is responsible for the flow of facts when returning from a method. There are two important cases to deal with: (1) when a resource is acquired in the called method, and returned to the callee, and (2) when a resource is passed by reference from the callee to the called method.

For the first case, we simply update the local of the data-flow fact, for the same reasons described in `getCallFlowFunction`. The second case is needed to avoid false positives. If a resource is passed by reference, we let any facts related to it flow through this flow function. To ensure the correct flow of these facts, we must eliminate them in `getCallToReturnFlowFunction`.

getCallToReturnFlowFunction. In our analysis, this function is responsible for acquiring and releasing method-scope resources and also for their correct flow, in conjunction with `getReturnFlowFunction`.

When a resource is acquired or released, it is necessary to call a method. In the case of our analysis, this method belongs to the Android framework. Due to how Soot works, we are unable to handle it in `getCallFlowFunction`, and need to handle in this flow function. For the best of our knowledge, in Jimple, acquiring a method-scope resource can be done through an `AssignStmt` or an `InvokeStmt`. Releasing a method-scope resource, on the other hand, is done only through `InvokeStmt`. Our implementation reflects this rules.

Regarding the intra-procedural analysis, Soot provides everything need to implement it. Our intra-procedural analysis is implemented as `ForwardBranchedFlowAnalysis`. The `ForwardBranchedFlowAnalysis` allows the developer to propagate different data-flow facts into different branches of the code, enabling the construction of a more precise analysis.

For the representation of the data-flow facts for the IFDS analysis, we created a class called `ResourceInfo`, which contains, as the name suggests, information about the specific resources seen during the analysis. This is a different representation from the one used for Android resources. It is composed by:

- `name` - A `String` that is the name of the local that represents the resource. For class-scoped variables, this is their source-code name, for method-scope variables, this is typically `$rx`, where x is a number generated by Soot
- `resource` - The `Resource` representation of the seen resource
- `declaringClass` - The `SootClass` where the seen resource was declared
- `declaringMethod` - The `SootMethod`, belonging to the `declaringClass`, where the seen resource was declared

- `isClassMember` - An `boolean` used to indicate if the seen resource was declared in a class field or not

In addition to the `ResouceInfo`, we use a `Local` to keep track of the resource and (1) check if it leaves the scope where it was defined, and (2) check if is passed as argument to a method, or returned from a method. The final data-flow fact representation is a `Pair` of a `ResourceInfo` and a `Local`. Figure 4.1 shows a class diagram of the `Resource` and `ResourceInfo` structures created for our analysis.

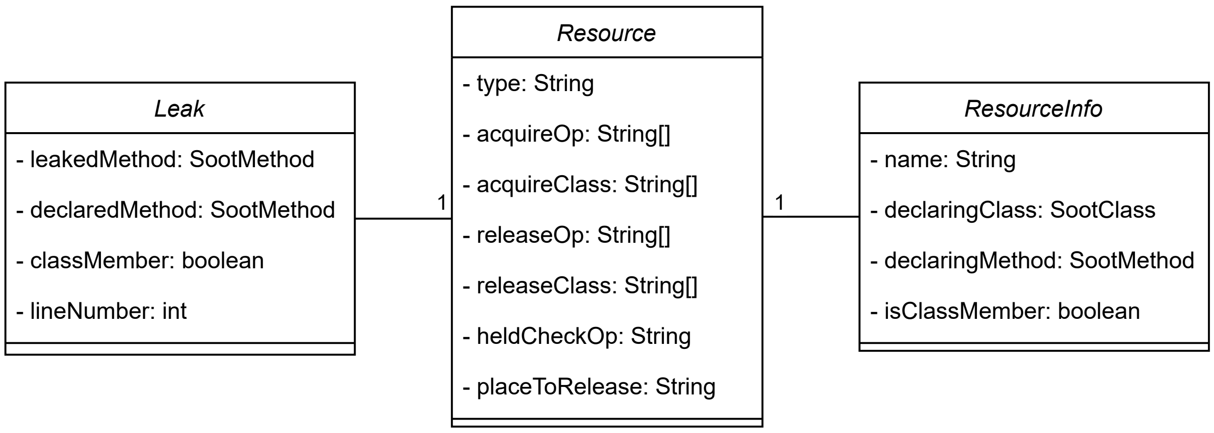


Figure 4.1: Class diagram of the implemented data structures: `Resource`, `ResourceInfo` and `Leak` (getters and setters omitted)

It is also important to note the differences between handling resources that are class-scoped, and resources that are method-scoped. The expected behaviour with class-scoped resources is that they can be used freely within the class, without the need to pass them as arguments or to return them from methods. This does not apply to method-scope resources. With this basis, we assume that class-scope resources are available only on their respective class. This decision was made based on their usage on some examples in the datasets [13, 33], and based on how the Android lifecycle works [3].

Resource leak example. To illustrate and better understand how the IFDS framework and our analysis work, we provide a real-world example of a leak detected by our tool and taken from the DroidLeaks dataset, shown in Listing 4.1. This is a cursor leak that spans two different methods, `getContact` and `getContactName` in a version of SMSDroid [2]. In `getContact`, the cursor `c` is acquired (line 11) and returned (line 13). The `getContactName` then calls `getContact` (line 3), and uses the `cursor` to return a string. From here, reference to `c` and `cursor` are lost, and the resource is never released, therefore, `c` is leaked. In Figure 4.2, we see the exploded super-graph of this example. The graph provides an overview of all the different type of edges defined in the IFDS framework, and how data flows through

[2]Source code at https://github.com/felixb/smsdroid/blob/5020594a25c7dd1d77b5e4571bce2135f4a17138/src/de/ub0r/android/smsdroid/AsyncHelper.java and https://github.com/felixb/ub0rlib/blob/master/lib/src/main/java/de/ub0r/android/lib/apis/ContactsWrapper3.java

them. In this specific example, there are only two facts present: the zero value – that represents a fact that is always valid, and used to generate another data-flow facts – and the *C* fact – that is our data-flow fact representing the cursor that is leaked. *C* is generated from the zero value when *c* is acquired, and flows through `getContact` until the end of `getContactName` since no release operation for `cursor` was performed.

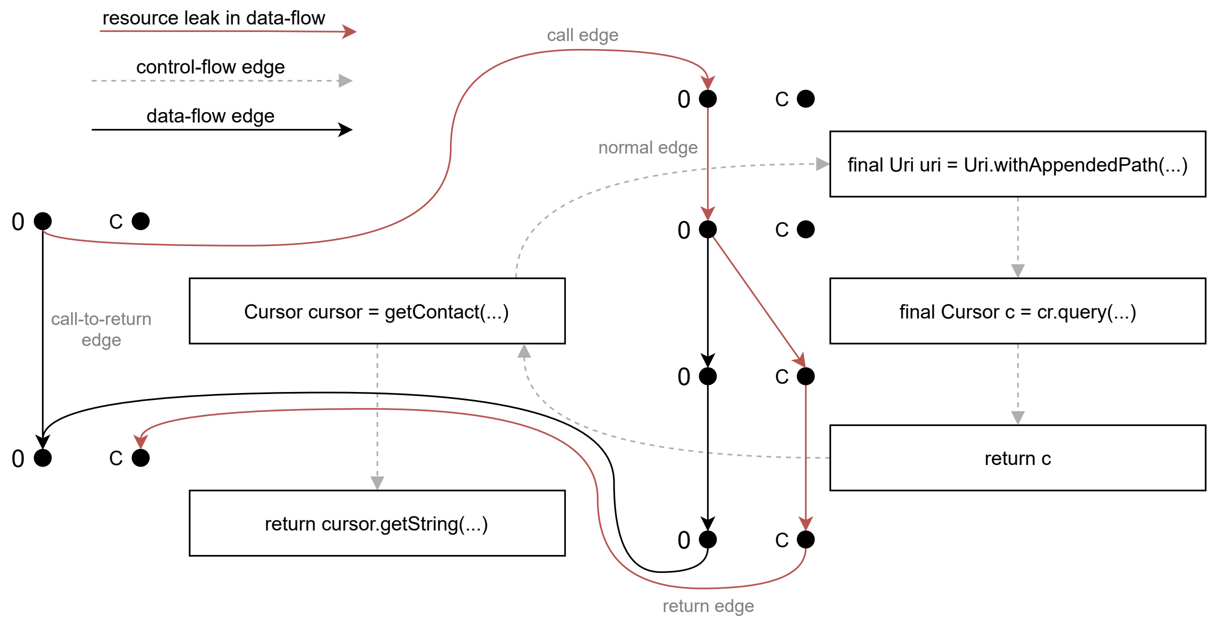

resource leak in data-flow
control-flow edge
data-flow edge
call edge
normal edge
call-to-return edge
return edge
0
C
Cursor cursor = getContact(...)
return cursor.getString(...)
final Uri uri = Uri.withAppendedPath(...)
final Cursor c = cr.query(...)
return c

Figure 4.2: Exploded super-graph for the example in Listing 4.1

The process to run the resource leak analysis is similar to when it is run in IntelliJ IDEA or as a standalone tool. Next, we describe the steps for this process in detail.

1. **Setup Soot.** Required to configure Soot together with FlowDroid. We define the Soot options and then create both FlowDroid's and Soot's configurations, which are used to setup the application to be analyzed.

2. **Pre-process Jimple.** On more long and complex methods, Jimple sometimes aliases locals. This creates false positives in our analysis, because the IFDS framework requires problems to be distributive, which do not handle aliasing well. To prevent aliasing altogether, we perform a simple transformation of Jimple code: whenever an instance of a resource is assigned to multiple locals in a method, we change the name of these multiple locals to a single one of the form `RESOURCE_ID`, where ID is a unique number.

3. **Register the transformers.** In order to run the analyis, Soot requires the implementation of a scene transformer, which is then associated to a specific pack. In our analysis, we have implemented two transformers: one for the inter-procedural analysis and another for the intra-procedure

analysis. The transformers are associated, respectively, with the whole-Jimple transformation pack (wjtp) – that enables transformations on the entire `Scene` – and the Jimple transformation pack (jtp) – that enables transformations on each individual `SootMethod` contained in the `Scene`.

4. **Run the analysis.** Having registred the scene transformers, the analysis can be started by applying the aforementioned packs

5. **Gather performance metrics.** In the last step of the analysis, we gather performance metrics. If the user is running the analysis on IntelliJ IDEA, the total duration is presented in the Event Log; if the user is running the standalone version, this information is presented in the output file containing all the results.

The current version of our extension allows multiple analysis to be implemented using the `IAnalysis` interface. This allows the creation of a visitor to connect the analysis' results to the Results component.

4.3 Results Component

As mentioned in Section 3.3.2, the Results component is built upon the Analysis component and the PSI API, while using some structures defined in Soot.

The `AnalysisVisitor` is responsible for visiting all the analysis that have run. This class is the core of the Results component, and is responsible for: acquiring the results, collecting the possible leaks, processing them, and sending them to their respective storage. Both existing analysis – intra-procedural and IFDS analysis – have methods to extract the raw results. The raw results obtained from the intra-procedural analysis are a `Set` of `Locals`, while the raw results from the IFDS analysis are a `Set` of `Pairs`, which contain a `ResourceInfo` and a `Local` (i.e. our data-flow facts). The `AnalysisVisitor` has access to these raw results and processes them independently by visiting both analysis.

Currently, the results obtained from the intra-procedural analysis do not suffer any kind of processing, and are simply added to the final results. On the other hand, the results from the IFDS analysis go through a 2-step process of collecting the location of possible leaks, and processing these locations to filter false positives. Figure 4.3 provides an overview of the process done by the Results component. From now on, we will focus on the process regarding the IFDS analysis.

4.3.1 Collection of Raw Results

To collect the results, we first need to know how to gather them after the analysis is finished. Heros' IFDS solver provides a method to gather results from individual statements of analyzed methods. The results are a set containing the data-flow facts at any given statement of the analyzed methods. Considering

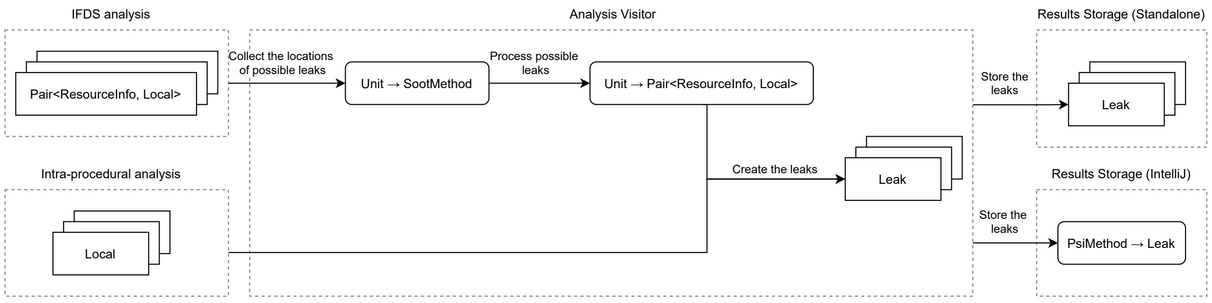

IFDS analysis
Pair<ResourceInfo, Local>
Intra-procedural analysis
Local
Analysis Visitor
Collect the locations of possible leaks
Unit → SootMethod
Process possible leaks
Unit → Pair<ResourceInfo, Local>
Create the leaks
Leak
Results Storage (Standalone)
Store the leaks
Leak
Results Storage (IntelliJ)
Store the leaks
PsiMethod → Leak

Figure 4.3: Data overview of the Results component processes

the properties of our problem, we designed a simple collection algorithm. As previously said, our data-flow facts are used to indicate if a given resource is acquired at some point in the code. If, in some statement, we have a data-flow fact, it means that, prior to that statement, a resource was acquired and has not yet been released. Having this in mind, our algorithm gathers, under certain conditions, the return statements where there are data-flow facts present. The conditions in which we gather the results depend mainly on the scope of the (possibly) leaked resource. For class-scoped resources, we are interested to see if they are leaked in a callback of interest - `onStop()` or `onPause()`. If a class-scoped resource was not released in these two callbacks, it is most likely leaked. For method-scoped resources, there can be a leak if the supposed leaked resource is not the being returned by the return statement and is leaked at this point – because this means that the program will lose reference to the supposed leaked resource. A pseudo-code representation of the algorithm created to collect the raw results is shown in Algorithm 4.1.

Algorithm 4.1: IFDS possible leaks location collection algorithm

```
begin
    for each analyzed method m do
        for each statement stmt in m do
            Collect the raw results in stmt into results
            if stmt is a ReturnStmt then
                for each fact in results do
                    if (the fact's resource is class-scoped and m is a callback of interest) or (stmt
                      does not return the fact's Local and the fact's resource is not class-scoped)
                      then
                        Collect the pair stmt and m
            else if stmt is a ReturnVoidStmt then
                if (the fact's resource is class-scoped and m is a callback of interest) or (fact's
                  resource is not class-scoped) then
                    Collect the pair stmt and m
```

4.3.2 Processing the Collected Results

This step focus on filtering false positives collected in the previous step. When using our algorithm, it is not enough to collect leaks at the end of a method's execution – we have to keep in mind the inter-procedural nature of the analysis, and that the collected leaks may not be real leaks (i.e. they can be false positives). This problem can be presented in a simple example.

False positive example. Let us imagine that `methodA` acquires a resource *r* and then calls `methodB` with *r* as a parameter. Then, `methodB` uses *r* but does not release it neither does return it. Then, after the call to `methodB`, `methodA` releases the resource *r*, meaning that the resource is not leaked. In this example, our analysis would propagate to `methodB` the fact that *r* was acquired in `methodA`. Then, our algorithm would collect a leak in `methodB` – seeing that this method does not return the resource and that there is a data-flow fact regarding *r* in the method's return statement. Figure 4.4 shows the exploded super-graph of this example, with `methodA` on the left and `methodB` on the right.

With this problem in mind, we developed the algorithm presented in Algorithm 4.2, to process the collected results. The algorithm goes through the previously collected possible leaks and, for method-scoped resources, checks if the callers of the method where the leak was found use the leaked resource and also have the leak; if so, this means we have a leak. For class-scoped resources, there is a leak if the resource was leaked in the method where it was supposed to be released.

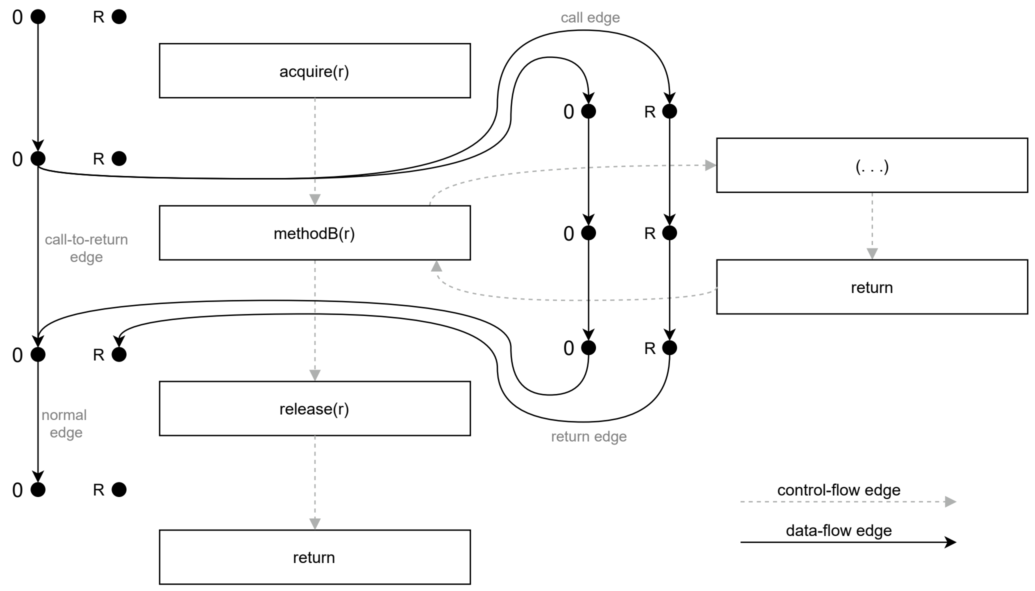


Figure 4.4: Example of a false positive of a resource leak

Algorithm 4.2: IFDS leaks processing algorithm

```
begin
    for each statement stmt and method m pair previously collected do
        for each fact at stmt do
            if fact's resource is class-scoped then
                if the declaring class of the resource in fact is the declaring class of m and the
                  place to be released of the resource in fact is m then
                    Collect the pair stmt and fact
            else
                for each caller of m do
                    if caller uses fact's resource and fact's resource is leaked in caller then
                        Collect the pair stmt and it fact
```

4.3.3 Result Storage and Presentation

As mentioned in Section 4.3, we implemented our tool to allow obtaining the results from both the intra- and inter-procedural. To make this process uniform, we created the `Leak` data structure to represent, as the name indicates, a leak. This is a very simple data structure, containing:

- `resource` - the `Resource` that was leaked
- `leakedMethod` - the `SootMethod` where the resource was leaked
- `declaredMethod - the SootMethod` where the resource was declared
- `classMember` - a boolean which indicates if the resource is class-scoped
- `lineNumber` - the line number in the source code where the resource was declared

This is the data structure used to store the final results for both analysis. Its class diagram is shown in Figure 4.1. We also need to take into account how to store the results depending on how the tool is being run – on IntelliJ IDEA/Android Studio or standalone. To know how to store the results, we first need to evaluate how we want to present them to the user.

For the standalone version, the results are to be presented in Comma Seperated Values (CSV) files. For this purpose, we simply store the leaks in three sets: one for the intra-procedural, one for the inter-procedural analysis, and one containing the leaks from both analysis. The CSV files are generated at the very end of the detection process, having the information contained in all the leaks, plus the class where the resource was declared and the class where the resource was leaked, and performance metrics.

For the IntelliJ IDEA version, we wish to follow the current methodology in EcoAndroid, which is to give warnings in the code, as well as to make them available as results of a code inspection. To allow this, we first identify the `PsiMethods` corresponding to the `leakedMethod` in the reported leaks, and we map the leaks to the corresponding `PsiMethod` where they were leaked. To present them to the user, we implement a code inspection responsible for visiting each `PsiMethod` in the PSI tree and checking, in the reported results, if there are any leaks in the visited `PsiMethods`. At the end of the detection process, we force IntelliJ Code Analyzer Daemon to restart, which causes the code to be inspected and code warnings to appear without the user needing to run a full code analysis.

4.4 Integration with IntelliJ IDEA

Throughout this chapter we have explained how our analysis was integrated with IntelliJ IDEA and Android Studio. In this section, we summarise this aspects and provide more insights on the integration.

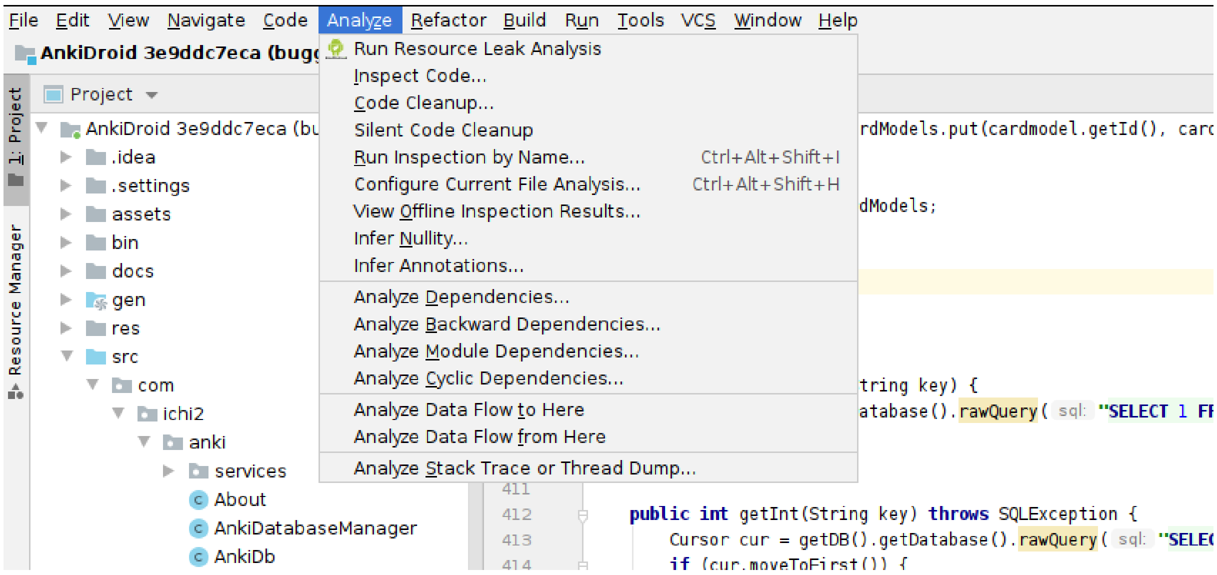
When it comes to integration, we aim to maintain consistency with the current version of EcoAndroid – even if we need to add new logic to the user interface of the plugin – and keep the interaction process simple. The current implementation of EcoAndroid, which detects and refactors energy patterns, only uses the IntelliJ SDK and the PSI API, meaning that the detection is already implemented within the plugin, and therefore, integrated with the IDE. On the other hand, our analysis is implemented with other static analysis frameworks, which forces us to think on how to integrate with the plugin and with the IDE.

There are three important aspects regarding the integration of our analysis in the EcoAndroid plugin: (1) how the user will initiate the analysis, (2) how can the user know the current state of the analysis, and (3) how will the analysis present the results to the user.

Initiating the analysis. When a user wants to run the resource leak analysis, they must go to menu `Analyze | Run Resource Leak Analysis`. We achieve by implementing an Action in `plugin.xml`. Then, the user selects the APK built from the current source code, and the analysis starts. How the user starts the analysis is shown in Figure 4.5 and Figure 4.6.

Checking the state of the analysis. The lower toolbar of the IntelliJ IDEA shows the current running tasks in the IDE. Our analysis is implemented as a task, which allows us to program information about each step, so that its progress is shown in the lower toolbar. This can be seen in Figure 4.7. There are current three steps shown to user: ”Setting up Soot”, ”Running intra-procedural analysis”, ”Running inter-procedural analysis”. When the analysis is over, a popup appears, notifying the user.

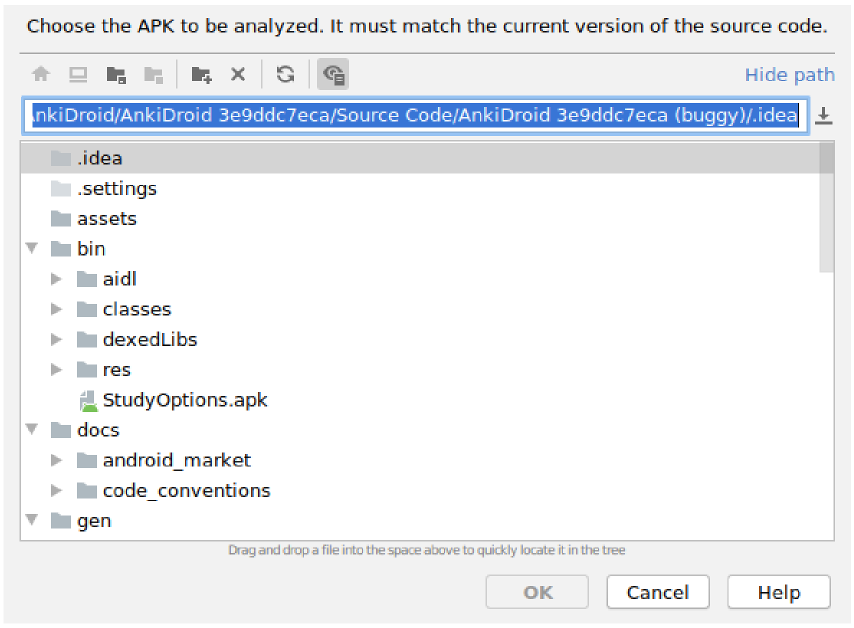
Showing the results. As mentioned in Section 4.3.3, the results are shown in two different ways: as warning in the source code, and as a report obtained when the user performs a Code Inspection. The warning in the source code follow the current implementation for the warnings of the energy patterns; however, they are accompanied by line markers, so the user can easily identify in which methods there


File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
AnkiDroid 3e9ddc7eca (bug
Project
1: Project
Resource Manager
AnkiDroid 3e9ddc7eca (bu
.idea
.settings
assets
bin
docs
gen
res
src
com
ichi2
anki
services
About
AnkiDatabaseManager
AnkiDb
Run Resource Leak Analysis
Inspect Code...
Code Cleanup...
Silent Code Cleanup
Run Inspection by Name... Ctrl+Alt+Shift+I
Configure Current File Analysis... Ctrl+Alt+Shift+H
View Offline Inspection Results...
Infer Nullity...
Infer Annotations...
Analyze Dependencies...
Analyze Backward Dependencies...
Analyze Module Dependencies...
Analyze Cyclic Dependencies...
Analyze Data Flow to Here
Analyze Data Flow from Here
Analyze Stack Trace or Thread Dump...

```
rdModels.put(cardmodel.getId(), carc

dModels;

tring key) {
atabase().rawQuery( sql: "SELECT 1 FF

    public int getInt(String key) throws SQLException {
        Cursor cur = getDB().getDatabase().rawQuery( sql: "SELEC
        if (cur.moveToFirst()) {
```

Figure 4.5: Starting the resource leak analysis

were leaks reported. To implement this functionality, we used a code inspection, just like EcoAndroid currently does for detecting energy patterns. The results of the resource leak analysis are given to the inspections, which prompts it to highlight the leaked methods. The line markers in IntelliJ work similarly to code inspections – they are implemented in the code and then described in `plugin.xml`. Both the warnings in the source code and the line markers can be seen in Figure 4.8. The Code Inspection report presents the full list of resource leaks, together with the source code of the method where the leak is reported, as it can be seen in Figure 4.9.

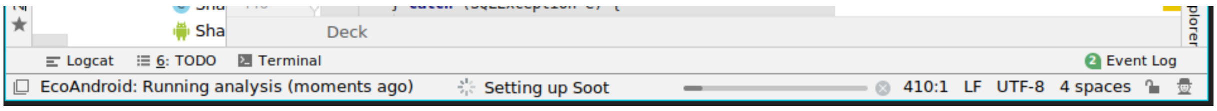

Choose the APK to be analyzed. It must match the current version of the source code.

```
nkiDroid/AnkiDroid 3e9ddc7eca/Source Code/AnkiDroid 3e9ddc7eca (buggy)/.idea
```

```
.idea
.settings
assets
▼ bin
    ▶ aidl
    ▶ classes
    ▶ dexedLibs
    ▶ res
      StudyOptions.apk
▼ docs
    ▶ android_market
    ▶ code_conventions
▼ gen
```

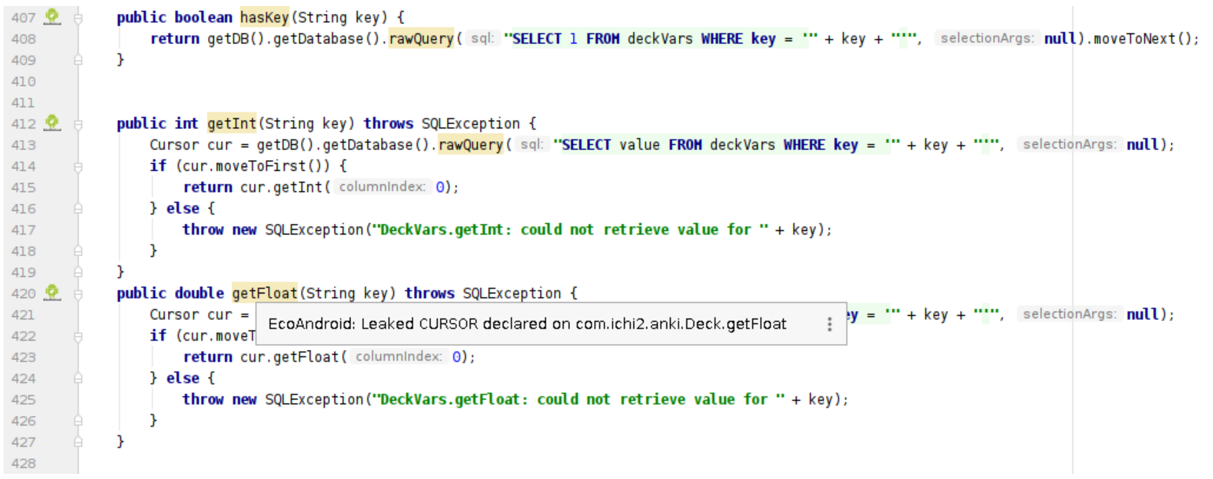
Drag and drop a file into the space above to quickly locate it in the tree

Figure 4.6: Choosing the APK to analyze



```
EcoAndroid: Running analysis (moments ago)    Setting up Soot    410:1  LF  UTF-8  4 spaces
```

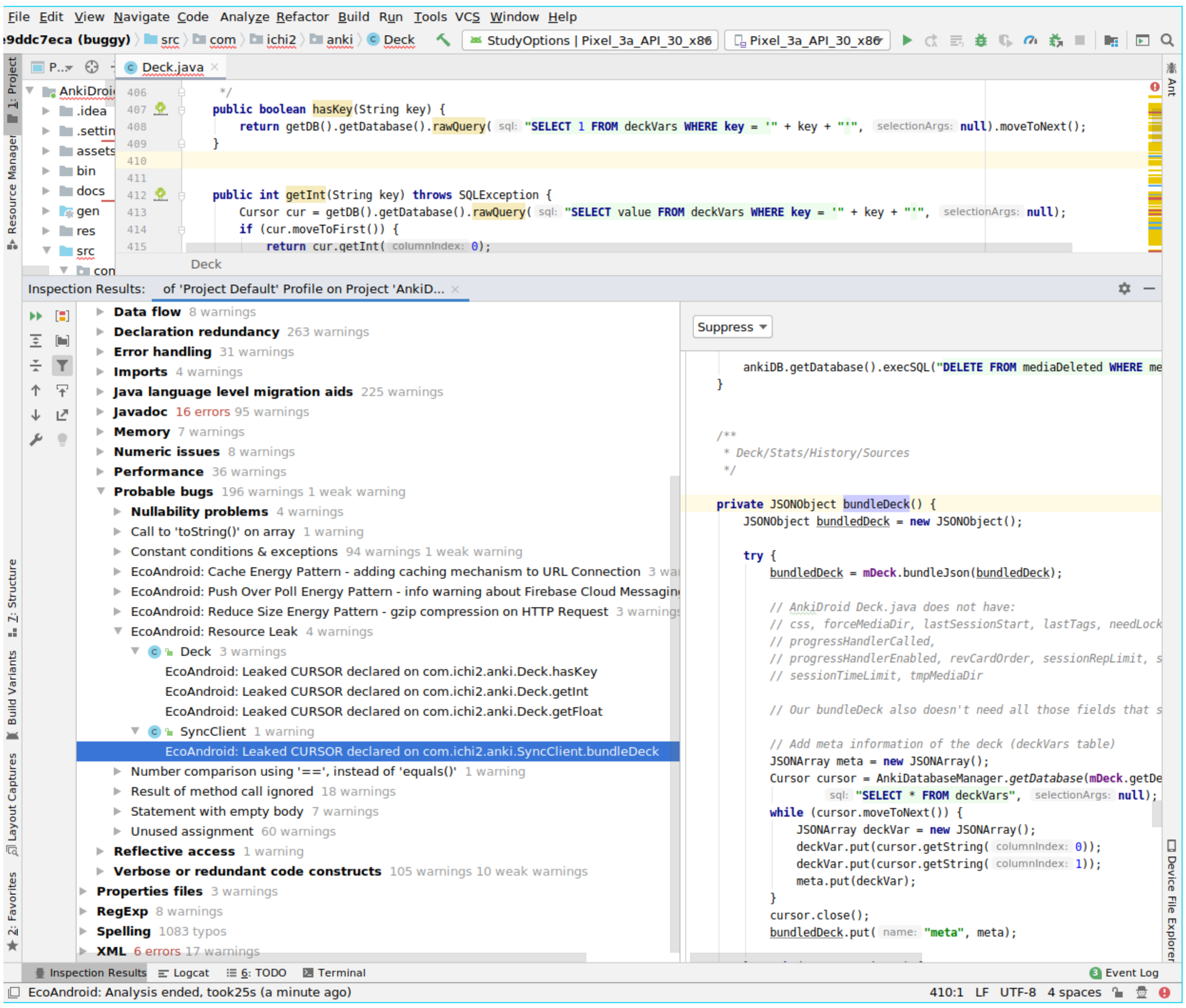
Figure 4.7: State of the analysis



```
public boolean hasKey(String key) {
    return getDB().getDatabase().rawQuery( sql: "SELECT 1 FROM deckVars WHERE key = '" + key + "'",  selectionArgs: null).moveToNext();
}


public int getInt(String key) throws SQLException {
    Cursor cur = getDB().getDatabase().rawQuery( sql: "SELECT value FROM deckVars WHERE key = '" + key + "'",  selectionArgs: null);
    if (cur.moveToFirst()) {
        return cur.getInt( columnIndex: 0);
    } else {
        throw new SQLException("DeckVars.getInt: could not retrieve value for " + key);
    }
}
public double getFloat(String key) throws SQLException {
    Cursor cur =                                                    y = '" + key + "'",  selectionArgs: null);
    if (cur.moveT
        return cur.getFloat( columnIndex: 0);
    } else {
        throw new SQLException("DeckVars.getFloat: could not retrieve value for " + key);
    }
}
```

```
EcoAndroid: Leaked CURSOR declared on com.ichi2.anki.Deck.getFloat
```

Figure 4.8: Warning of a leak in the source code



```
    */
public boolean hasKey(String key) {
    return getDB().getDatabase().rawQuery( sql: "SELECT 1 FROM deckVars WHERE key = '" + key + "'",  selectionArgs: null).moveToNext();
}


public int getInt(String key) throws SQLException {
    Cursor cur = getDB().getDatabase().rawQuery( sql: "SELECT value FROM deckVars WHERE key = '" + key + "'",  selectionArgs: null);
    if (cur.moveToFirst()) {
        return cur.getInt( columnIndex: 0);
```

Deck

Inspection Results: of 'Project Default' Profile on Project 'AnkiD...

```
Data flow  8 warnings
Declaration redundancy  263 warnings
Error handling  31 warnings
Imports  4 warnings
Java language level migration aids  225 warnings
Javadoc  16 errors 95 warnings
Memory  7 warnings
Numeric issues  8 warnings
Performance  36 warnings
Probable bugs  196 warnings 1 weak warning
  Nullability problems  4 warnings
  Call to 'toString()' on array  1 warning
  Constant conditions & exceptions  94 warnings 1 weak warning
  EcoAndroid: Cache Energy Pattern - adding caching mechanism to URL Connection  3 wa
  EcoAndroid: Push Over Poll Energy Pattern - info warning about Firebase Cloud Messagin
  EcoAndroid: Reduce Size Energy Pattern - gzip compression on HTTP Request  3 warning
  EcoAndroid: Resource Leak  4 warnings
    Deck  3 warnings
      EcoAndroid: Leaked CURSOR declared on com.ichi2.anki.Deck.hasKey
      EcoAndroid: Leaked CURSOR declared on com.ichi2.anki.Deck.getInt
      EcoAndroid: Leaked CURSOR declared on com.ichi2.anki.Deck.getFloat
    SyncClient  1 warning
      EcoAndroid: Leaked CURSOR declared on com.ichi2.anki.SyncClient.bundleDeck
  Number comparison using '==', instead of 'equals()'  1 warning
  Result of method call ignored  18 warnings
  Statement with empty body  7 warnings
  Unused assignment  60 warnings
Reflective access  1 warning
Verbose or redundant code constructs  105 warnings 10 weak warnings
Properties files  3 warnings
RegExp  8 warnings
Spelling  1083 typos
XML  6 errors 17 warnings
```

```
        ankiDB.getDatabase().execSQL("DELETE FROM mediaDeleted WHERE me
    }


    /**
     * Deck/Stats/History/Sources
     */

    private JSONObject bundleDeck() {
        JSONObject bundledDeck = new JSONObject();

        try {
            bundledDeck = mDeck.bundleJson(bundledDeck);

            // AnkiDroid Deck.java does not have:
            // css, forceMediaDir, lastSessionStart, lastTags, needLock
            // progressHandlerCalled,
            // progressHandlerEnabled, revCardOrder, sessionRepLimit, s
            // sessionTimeLimit, tmpMediaDir

            // Our bundleDeck also doesn't need all those fields that s

            // Add meta information of the deck (deckVars table)
            JSONArray meta = new JSONArray();
            Cursor cursor = AnkiDatabaseManager.getDatabase(mDeck.getDe
                     sql: "SELECT * FROM deckVars",  selectionArgs: null);
            while (cursor.moveToNext()) {
                JSONArray deckVar = new JSONArray();
                deckVar.put(cursor.getString( columnIndex: 0));
                deckVar.put(cursor.getString( columnIndex: 1));
                meta.put(deckVar);
            }
            cursor.close();
            bundledDeck.put( name: "meta", meta);
```

```
EcoAndroid: Analysis ended, took25s (a minute ago)          410:1  LF  UTF-8  4 spaces
```

Figure 4.9: Full report of resource leaks

5

Evaluation

Contents

In this chapter we describe the dataset used for evaluating our tool, and the process for collecting and analyzing the results. In Section 5.1 we present the methodology used in our evaluation process. Section 5.2 shows the results obtained in the evaluation of our tool.

5.1 Methodology

5.1.1 Resource Leak Dataset

As seen in Section 2.4, researchers have created public datasets and benchmarks containing resource leaks in multiple applications. We have chosen DroidLeaks [13] as our golden standard for evaluation. The DroidLeaks dataset provides information on resource leaks found on 32 popular and large-scale open-source Android applications, taken from F-Droid. The authors collected a total of 292 resource leaks from 33 resource classes, which include the 4 resources – Cursor, SQLite Database, Wakelock, and Camera – our tool is able to identify.

The authors of DroidLeaks also evaluated 8 resource leak detectors with the dataset, to help future researchers create and improve resource leak detection tools. For the evaluation of each tool *t*, the authors defined two metrics: the Bug Detection Rate, denoted *BDR(t)*, and the False Alarm Rate, denoted *FAR(t)*. A detected leak happens when a tool detects one of the specified leaks on the faulty version of the application. A false alarm happens when a tool detects one of the specified leaks on the patched version of the application (it should not since the leak is fixed).

The Bug Detection Rate and False Alarm Rate are calculated as follows:

$$BDR(t) = \frac{\text{\# bugs detected by t on buggy app versions}}{\text{\# bugs experimented on t}} \tag{5.1}$$

$$FAR(t) = \frac{\text{\# false alarms reported by t on patched app versions}}{\text{\# bugs experimented on t}} \tag{5.2}$$

The authors of DroidLeaks made the decision to evaluate only 116 of the 292 resource leaks found, due to the labor-intensive work of compiling all APK found. The 116 leaks they have chosen also include leaks from all the patterns described in their work which, according to them, is enough for their evaluation. For our evaluation, we are only interested in resource leaks regarding the resources our tool is able to detect. From those 116 resource leaks only 50 fit our criteria (herafter “reduced dataset”). We will use the reduced dataset to evaluate our tool with DroidLeaks. Table 5.1 shows, regarding the reduced dataset, the number of leaks from each resource class, as well as the applications where they were identified.

There is a publicly available website [1] that contains all the information about the dataset. From the

[1] `http://sccpu2.cse.ust.hk/droidleaks/`

Resource class	# leaks	Related applications
`Cursor`	38	AnkiDroid, AnySoftKeyboard, APG, BankDroid, ChatSecure, CSipSimple, Google Authenticator, IRCCloud, Osmand, OSMTracker, Owncloud, SMSDroid, TransDroid, WordPress
`SQLiteDatabase`	3	AnySoftKeyboard, ConnectBot, FBReader
`Wakelock`	8	CallMeter, ConnectBot, CSipSimple, K-9 Mail,
`Camera`	1	SipDroid

Table 5.1: Subset of resource leaks evaluated in DroidLeaks

available information, there is a spreadsheet [2] containing the 292 identified leaks together with their relevant information:

- name of the application where the leak was found
- the concerned class, i.e. the resource class
- the version of the application where the problem was discovered, and the version where the problem was resolved
- the problematic method, and the file where this method is implemented
- the bug report, if exists
- for the 8 evaluated resource leaks detectors, whether they detected the resource leak or not
- information regarding leak: if is related to component life cycle, if the resource escapes local context, and the extent of the leak (complete leak, only in certain paths, etc)

Additionally, the authors of DroidLeaks provide the APK used in the evaluation they performed. There is a total of 137 made publicly available[3] – which includes the versions were the leaks were found and the versions where the leaks were fixed. From what we have verified, only 129 APK were used in DroidLeak's evaluation of the 8 resource leak detectors. In our evaluation, we will consider the 137 available APK, as described next.

5.1.2 Data Collection and Analysis

To gather the results, we will run our analysis on the 137 provided APK by DroidLeaks (hereafter "full analysis"). We will first consider the evaluation with the reduced dataset to compare the efficiency of our tool with the others evaluated in DroidLeaks. We will use the Bug Detection Rate and the False

[2] `http://sccpu2.cse.ust.hk/droidleaks/project_data/droidleaks.xlsx`
[3] `http://sccpu2.cse.ust.hk/droidleaks/bugs/apks.php`

Alarm Rate, as to also compare with the other 8 tools evaluated in DroidLeaks. Additionally, we will measure three metrics: precision, recall, and F-Score [48]. These metrics will be calculated based on the full analysis. As for performance, we will calculate the average and median time that our tool took to analyze the provided applications.

We ran our evaluation on the standalone version of our analysis, on an Intel i5-8265U (8 cores) machine, with 8GB of RAM running Ubuntu 18.04.5 LTS. The process used to evaluate our tool is summarized below:

1. Run our analysis in standalone mode on the 137 APK from DroidLeaks
2. Collect and organize the obtained results into a spreadsheet
3. Compare the obtained results with the reduced dataset to identify correctly detected leaks and non-detected leaks (i.e. true positives and false negatives, respectively).
4. Manually categorize the remaining results (i.e. the results obtained and not described in the reduced dataset)
5. Calculate the analysis' detection rate and compare with the tools evaluated in DroidLeaks, from the reduced dataset
6. Calculate the remaining efficiency metrics – precision, recall, and F-Score – based on the full analysis and false negatives obtained from DroidLeaks
7. Calculate performance metrics – average and median duration of the analysis – based on the full analysis.

5.2 Results

5.2.1 Errors in the Analysis

From the 137 APK provided by DroidLeaks, our analysis failed to run on 30 due to call graph generation failure in Soot and FlowDroid. We define a call graph generation failure as the failure to generate a call graph in under 5 minutes. The applications suffering from this failure and their versions can be seen on Table 5.2. For these applications, our analysis is unable to run and detect resource leaks. Regarding evaluation on the reduced dataset, this means that the cursor leak on version 1747b81da8 of BankDroid can not be evaluated, but will be accounted in our evaluation as a call graph generation failure. Regarding the evaluation of the full analysis, this means that we will only consider 107 out of the 137 APK provided by DroidLeaks.

Application	Versions
K-9 Mail	0a07250417, 0e03f262b3, 1596ddfaab, 2df436e7bc, 3077e6a2d7, 3171ee969f, 378acbd313, 57e55734c4, 58efee8be2, 71a8ffc2b5, 7e1501499f, acd18291f2
Cgeo	23bf7d5801, 253c271b34, 8987674ab4 e2c320b5f9, ea04b619e0, fb2d9a3a57
BankDroid	1747b81da8, 265504aa4, 2b0345b5c2, bf136c7b0a, f4fbbfd966
Ushahidi	337b48f5f2, 52525168b5, 9d0aa75b84, d578c72309
ConnectBot	2dfa7ae033, ef8ab06c34
CallMeter	4e9106ccf2

Table 5.2: Applications that suffered from call graph generation failure

5.2.2 Reduced Dataset

With everything considered in this Chapter, we evaluated our tool on the reduced dataset obtained from DroidLeaks. For the 50 resource leaks in the reduced dataset, our tool was able to detect 9 (18%), while failing to detect the remaining 41 (82%), meaning we achieved a Bug Detection Rate of 18% and a False Alarm Rate of 2%. We have investigated the cause of this results and observed that, for the 41 that our tool failed to detect, the two main reasons were due to Soot and Heros not analyzing the method where the resource was leaked, which happened in 25 (61%) of the leaks, and also due to special mechanisms used by some resources and not supported by our tool, which happened in 7 (17%) of the leaks. Table 5.3 shows each cause for failure to detect the leaks in the reduced dataset, together with their corresponding number of cases (percentage is calculated based on only the 41 leaks our tool failed to detect, and does not account for 100% due to approximation errors).

Cause for failing to detect	# of cases	% of cases
Method not analyzed	25	61%
Logic not supported	8	20%
Unresolved bug in tool	5	12%
Call graph generation failure	1	2%
Call graph generation error	1	2%
Unknown cause	1	2%
Total	41	100%

Table 5.3: Causes for failing to detect leaks in reduced dataset

As mention in Section 5.1.1 and Section 5.1.2, the authors of DroidLeaks performed an evaluation of 8 resource leak detectors with their dataset, calculating their detection rate. Table 5.4 shows how the tools evaluated in DroidLeaks and EcoAndroid performed on the reduced dataset.

Tool	# experimented leaks	# detected leaks (Bug Detection Rate)	# false alarms (False Alarm Rate)
EcoAndroid	50	9 (18.0%)	1 (2.0%)
Code Inspections	41	32 (78.0%)	19 (46.3%)
Infer	38	23 (60.5%)	2 (5.3%)
Lint	38	12 (31.6%)	0 (0.0%)
Relda2-FS	9	7 (77.8%)	7 (77.8%)
Relda2-FI	9	3 (33.4%)	2 (22.2%)
Elite	8	7 (87.5%)	5 (62.5)
Verifier	8	4 (50.0%)	3 (37.5%)

Table 5.4: Performance of evaluated tools in DroidLeaks, from the reduced dataset

5.2.3 Full Analysis

	Full reported leaks	Unique reported leaks
Total apps analyzed	107	
Number of leaks reported	312	127
Unclassified leaks	27	9
Errors	5	4
True positives (TP)	203	86
False positives (FP)	77	28
False negatives (FN)	41 (from reduced dataset)	
Precision	0.725	0.754
Recall	0.832	0.677
F-Score	0.775	0.714

Table 5.5: Results obtained from full analysis

As previously said, we also evaluated our tool on all 137 avaliable APK provided by DroidLeaks. Due to call graph generation failures on 37 APK, we only consider 107 for the evaluation of the full analysis presented in this section.

Our tool reported a total of 312 leaks, from which 203 (65%) are true positives, 77 are false positives (25%), 27 (9%) were not classified due to missing code in the application's repository and due to the leak being reported in an Android class, and 5 (1%) suffered from errors in the Jimple translation. We obtained a precision of 72.5%, a recall (with false negatives based on the reduced dataset) of 83.2%, and an F-Score of 77.5%.

Resource	Full reported leaks			Unique reported leaks		
	Total (%)	TP (%)	FP (%)	Total (%)	TP (%)	FP (%)
Cursor	165 (53%)	108 (53%)	42 (55%)	63 (50%)	40 (47%)	14 (50%)
SQLite Database	114 (37%)	90 (44%)	20 (26%)	51 (40%)	43 (50%)	6 (21%)
Wakelock	31 (9%)	5 (3%)	13 (17%)	12 (9%)	3 (3%)	7 (25%)
Camera	2 (1%)	0 (0%)	2 (2%)	1 (1%)	0 (0%)	1 (4%)
Sum	312	203	77	127	86	28

Table 5.6: Results obtained from full analysis, organized per resource

We observed that some of the reported leaks were duplicated in different versions of the same application. This phenomenon can be seen, for example, in WordPress: in four versions of this application (57c0808aa4, 4b1d15cb26, 42de8a232c, and 3f6227e2d4) we have uncovered several identical reported leaks. Since this happens in several applications, we decided to also present the results of our tool taking into account only unique reported leaks. In this case, our tool reported 127 leaks, from which 86 (67.7%) are true positives, 28 (22%) are false positives, 9 (7.1%) were unclassified, and 4 (3.1%) suffered errors in the Jimple translation. For the unique reported leaks, we obtained a precision of 75.4%, a recall (with false negatives based on the reduced dataset) of 67.7%, and an F-Score of 71.4%. Table 5.5 summarizes the results obtained and the efficency metrics calculated regarding the full analysis.

Table 5.6 shows the results obtained from the full analysis, from both all reported leaks and unique reported leaks, but categorized by each resource. Percentages in each column are calculated based on the sum of their respective column.

For performance evaluation, we recorded the time our tool took to setup and run the analysis. To setup the analysis, our tool took, on average, 43941 milliseconds and, on median, 20577 milliseconds. To run the analysis it took, on average, 3520 milliseconds and, on median, 3869 milliseconds. Table 5.7 shows this recorded times, as well as total time, presented in milliseconds and in minutes.

	Setup	Analysis	Total
Average time (ms)	43941	3520	47461
Median time (ms)	20577	3869	24356
Average time (min)	0.73235	0.05866	0.79102
Median time (min)	0.34295	0.06448	0.40593

Table 5.7: Time performance of the analysis

6 Conclusions

Contents

In this chapter we discuss the contributions provided by our extension to EcoAndroid, as well as the shortcomings and possible improvements for our work. Section 6.1 describes the contributions of our extension, as well as if our objectives were met. In Section 6.2 we discuss some of the limitations of our work, ways to overcome them, and improvements that could be made in a future version of our extension.

6.1 Conclusions

The main objective for this thesis was to extend EcoAndroid with automated detection of resource leaks in Android applications. This result was achieved through the design and implementation of a fully-precise context- and flow-sensitive inter-procedural static analysis with the IFDS framework. Our analysis supports the detection of leaks regarding four frequently used and impactful Android resources, and can be run in IntelliJ IDEA or Android Studio through EcoAndroid, and also as a command-line tool, if needed. When using our tool to analyze 107 Android applications from the DroidLeaks dataset, we have been able to detect 194 previously undetected leaks. Our analysis achieved a low Bug Detection Rate due to problems in the frameworks used, but our False Alarm Rate was one of the best when comparing to the 8 resource leak detectors evaluated in DroidLeaks. We also obtained a precision of 72.5% and a recall of 83.2% when evaluating the leaks detected in the 107 applications provided by DroidLeaks.

6.2 Shortcomings and Future Work

Architecture. While we designed and implemented our extension with the creation of other analysis in mind, the resulting architecture can be further improved. Taking into account the need to run the analysis as a standalone tool, one can abstract the whole Analysis and Results components into a separated module. This module could be implemented in such a way that could be used as a library by any developer. This would allow, for example, a implementation of our analysis in another IDE like Eclipse.

Use of static analysis frameworks. While static analysis frameworks like Soot provide the necessary tools to build static analysis, these frameworks also have problems of their own. In our extension we observed that Soot's and FlowDroid's call graph generation can sometimes fail, which makes it impossible to run our analysis. Another problem that can also happen is the erroneous construction of call graphs. Although that, in this case, it is possible to run the analysis, this can cause false positives or false negatives. Unfortunately, we could not uncover the causes nor fix this type of failures.

Improving intra-procedural analysis. As previously mentioned, although we have implemented an intra-procedural analysis, our inter-procedural analysis outperforms it and so it is currently disabled. The intra-procedural analysis could be revisited and improved as much as possible, with the goal of implementing single-method resource leak analysis in EcoAndroid.

Special mechanisms used by resources. Throughout testing and evaluation of our analysis, we uncovered that, for the resources supported, many possess different kinds of mechanisms that affect how they are acquired and released, as discussed in Section 4.1. One massive improved to our tool would be taking into account as many special mechanisms as possible, to improve the true positives detected, and reduce the false positives.

Refactoring resource leaks. An obvious step in our extension would be to implement automated refactoring of the detected leaks. This would require a greater expertise of how each resource works and the leaks express themselves in the code, so that the refactoring would not impact the rest of the application. A similar mechanism to the refactor of energy patterns could be used.

Broader evaluation. Although the user interacts directly with our extension, we did not perform user tests due to time constraints. For future work, an evaluation regarding usability could be performed and the interaction process improved based on these results.

Bibliography

[1] L. Li, T. F. Bissyandé, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein, and L. Traon, “Static analysis of android apps: A systematic literature review,” *Information and Software Technology*, vol. 88, pp. 67–95, 2017.

[2] Google, “Understanding the Application Lifecycle - Android Developers,” https://developer.android.com/guide/components/activities/activity-lifecycle, accessed: 15-12-2020.

[3] ——, “Android fundamentals 02.2 - Activity lifecycle and state,” https://developer.android.com/codelabs/android-training-activity-lifecycle-and-state#0, accessed: 15-12-2020.

[4] S. O’Dea, “Number of smartphone users worldwide from 2016 to 2021,” https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide, accessed: 19-12-2020.

[5] M. Chau and R. Reith, “Smartphone market share,” https://www.idc.com/promo/smartphone-market-share/os, accessed: 19-12-2020.

[6] Appbrain, “Number of android apps on google play,” https://www.appbrain.com/stats/number-of-android-apps, accessed: 19-12-2020.

[7] J. Clement, “Number of available applications in the google play store from december 2009 to september 2020,” https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/, accessed: 19-12-2020.

[8] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy, “Investigating the energy impact of android smells,” in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 115–126.

[9] A. Banerjee, L. K. Chong, S. Chattopadhyay, and A. Roychoudhury, “Detecting energy bugs and hotspots in mobile apps,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014, pp. 588–598.

[10] J. Wang, G. Wu, X. Wu, and J. Wei, “Detect and optimize the energy consumption of mobile app through static analysis: an initial research,” in *Proceedings of the Fourth Asia-Pacific Symposium on Internetware*, 2012, pp. 1–5.

[11] C. Wilke, S. Richly, S. Götz, C. Piechnick, and U. Aßmann, “Energy consumption and efficiency in mobile applications: A user feedback study,” in *2013 IEEE International Conference on Green Computing and Communications and IEEE Internet of Things and IEEE Cyber, Physical and Social Computing*. IEEE, 2013, pp. 134–141.

[12] S. Ali, S. Khusro, A. Rauf, and S. Mahfooz, “Sensors and mobile phones: evolution and state-of-the-art,” *Pakistan journal of science*, vol. 66, no. 4, p. 385, 2014.

[13] Y. Liu, J. Wang, L. Wei, C. Xu, S.-C. Cheung, T. Wu, J. Yan, and J. Zhang, “Droidleaks: a comprehensive database of resource leaks in android apps,” *Empirical Software Engineering*, vol. 24, no. 6, pp. 3435–3483, 2019.

[14] B. N. Bhatt and C. A. Furia, “Automated repair of resource leaks in android applications,” *arXiv preprint arXiv:2003.03201*, 2020.

[15] T. Wu, J. Liu, X. Deng, J. Yan, and J. Zhang, “Relda2: an effective static analysis tool for resource leak detection in android apps,” in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 762–767.

[16] M. Couto, J. Saraiva, and J. P. Fernandes, “Energy refactorings for android in the large and in the wild,” in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 217–228.

[17] A. Ribeiro, J. F. Ferreira, and A. Mendes, “Ecoandroid: An Android Studio plugin for developing energy-efficient Java mobile applications,” in *2021 IEEE 21th International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2021.

[18] Google, “Application Fundamentals - Android Developers,” https://developer.android.com/guide/components/fundamentals, accessed: 15-12-2020.

[19] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal, “Towards verifying android apps for the absence of no-sleep energy bugs,” in *Presented as part of the 2012 Workshop on Power-Aware Computing and Systems*, 2012.

[20] H. Wu, S. Yang, and A. Rountev, “Static detection of energy defect patterns in android applications,” in *Proceedings of the 25th International Conference on Compiler Construction*, 2016, pp. 185–195.

[21] C. Guo, J. Zhang, J. Yan, Z. Zhang, and Y. Zhang, "Characterizing and detecting resource leaks in android applications," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 389–398.

[22] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 1013–1024.

[23] U. Khedker, A. Sanyal, and B. Sathe, *Data flow analysis: theory and practice*. CRC Press, 2017.

[24] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of program analysis*. Springer Science & Business Media, 2004.

[25] H. Jiang, H. Yang, S. Qin, Z. Su, J. Zhang, and J. Yan, "Detecting energy bugs in android apps using static analysis," in *International Conference on Formal Engineering Methods*. Springer, 2017, pp. 192–208.

[26] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[27] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *Acm Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[28] T. Reps, S. Horwitz, and M. Sagiv, "Precise interprocedural dataflow analysis via graph reachability," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1995, pp. 49–61.

[29] E. Bodden, "Inter-procedural data-flow analysis with ifds/ide and soot," in *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, 2012, pp. 3–8.

[30] L. Cruz and R. Abreu, "Catalog of energy patterns for mobile applications," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2209–2235, 2019.

[31] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake? characterizing and detecting no-sleep energy bugs in smartphone apps," in *Proceedings of the 10th international conference on Mobile systems, applications, and services*, 2012, pp. 267–280.

[32] O. Le Goaër, "Enforcing green code with android lint," 2019.

[33] O. Riganelli, D. Micucci, and L. Mariani, "From source code to test cases: A comprehensive benchmark for resource leak detection in android apps," *Software: Practice and Experience*, vol. 49, no. 3, pp. 540–548, 2019.

[34] A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury, “Energypatch: Repairing resource leaks to improve energy-efficiency of android apps,” *IEEE Transactions on Software Engineering*, vol. 44, no. 5, pp. 470–490, 2017.

[35] T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang, “Light-weight, inter-procedural and callback-aware resource leak detection for android apps,” *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1054–1076, 2016.

[36] D. Li, Y. Lyu, J. Gui, and W. G. Halfond, “Automated energy optimization of http requests for mobile applications,” in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 2016, pp. 249–260.

[37] L. Cruz and R. Abreu, “Using automatic refactoring to improve energy efficiency of android apps,” *arXiv preprint arXiv:1803.05889*, 2018.

[38] L. Cruz, R. Abreu, and J.-N. Rouvignac, “Leafactor: Improving energy efficiency of android apps via automatic refactoring,” in *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 2017, pp. 205–206.

[39] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni, “Understanding and detecting wake lock misuses for android applications,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 396–409.

[40] D. Samudio, “Automated Android Energy-Efficiency InspectiON,” https://plugins.jetbrains.com/plugin/7444-aeon-automated-android-energy-efficiency-inspection/.

[41] L. Deng, J. Offutt, and D. Samudio, “Is mutation analysis effective at testing android apps?” in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 86–93.

[42] S. Krüger, S. Nadi, M. Reif, K. Ali, M. Mezini, E. Bodden, F. Göpfert, F. Günther, C. Weinert, D. Demmler *et al.*, “Cognicrypt: Supporting developers in using cryptography,” in *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, pp. 931–936.

[43] Google, “Android Reference - Cursor,” https://developer.android.com/reference/android/database/Cursor, accessed: 15-10-2021.

[44] ——, “Android Reference - Content Query Map,” https://developer.android.com/reference/android/content/ContentQueryMap, accessed: 15-10-2021.

[45] ——, “Android Reference - SQLiteDatabase,” https://developer.android.com/reference/android/database/sqlite/SQLiteDatabase, accessed: 15-10-2021.

[46] ——, “Android Reference - Wakelock,” https://developer.android.com/reference/android/os/PowerManager.WakeLock, accessed: 15-10-2021.

[47] ——, “Android Reference - Camera,” https://developer.android.com/reference/android/hardware/Camera, accessed: 15-10-2021.

[48] Center for Assured Software, “CAS Static Analysis Tool Study - Methodology,” National Security Agency, 9800 Savage Road Fort George G. Meade, MD 20755-6738, Tech. Rep., Dec. 2011.