# Extending EcoAndroid with Automated Detection of Resource Leaks

Ricardo B. Pereira
INESC-ID & IST, University of Lisbon
Lisbon, Portugal

João F. Ferreira
INESC-ID & IST, University of Lisbon
Lisbon, Portugal

Alexandra Mendes
HASLab / INESC TEC & Faculty of Engineering,
University of Porto, Porto, Portugal

Rui Abreu
INESC-ID & Faculty of Engineering, University of Porto
Porto, Portugal

## ABSTRACT

When developing mobile applications, developers often have to decide when to acquire and when to release resources. This leads to resource leaks, a kind of bug where a resource is acquired but never released. This is a common problem in Android applications that can degrade energy efficiency and, in some cases, can cause resources to not function properly.

In this paper, we present an extension of EcoAndroid, an Android Studio plugin that improves the energy efficiency of Android applications, with an inter-procedural static analysis that detects resource leaks. Our analysis is implemented using 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 leaks related to 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 83.2%. Our tool was able to uncover 191 previously unidentified leaks in this benchmark. These results show that our analysis can help developers identify resource leaks.

## CCS CONCEPTS

• **Software and its engineering → Automated static analysis**.

## KEYWORDS

Energy Consumption, Resource Leaks, Android, Green Software

**ACM Reference Format:**
Ricardo B. Pereira, João F. Ferreira, Alexandra Mendes, and Rui Abreu. 2022. Extending EcoAndroid with Automated Detection of Resource Leaks. In *IEEE/ACM 9th International Conference on Mobile Software Engineering and Systems (MOBILESoft '22), May 17–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3524613.3527815

## 1 INTRODUCTION

Mobile devices are more than ever prevailing in our society, with the estimated number of smartphone users in 2022 to be around 6.6 billion worldwide [26]. Android is the most used operating system, with its market share hitting an estimated 85%, followed by iOS with 15% [10]. The market for Android applications has also grown throughout the years, at times totaling a number of 3 million applications available in the Google Play Store [2, 11].

Recent research has been uncovering energy problems and inefficiencies that decrease the battery life of Android devices [5, 8, 33]. Given the sheer number of mobile devices in use, taking action to solve these energy problems and increasing the overall energy efficiency of Android applications can have a significant impact on energy consumption. Moreover, it can also impact 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 [34].

The diversity of sensors that modern mobile devices provide — such as cameras, fingerprint readers, and GPS receivers — has been growing [1], allowing developers to create applications that interact with users in novel ways. This interaction between applications and sensors can be handled manually by the developer through the API provided by Android; however, if not well implemented, it can have huge costs on the battery life of the device [22]. One problem that may arise from this incorrect use 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 [6, 22, 36], but not always have researchers been able to find resource leaks in applications [12]. More generally, detection of leaks and their localization was identified by Microsoft practitioners as one of the top ten important research ideas [25].

This paper describes an extension of EcoAndroid [29], an Android Studio plugin, with the ability to automatically detect resource leaks in Android applications. Our main contribution is a context- and flow-sensitive inter-procedural static analysis capable of detecting resource leaks in Android applications. Currently, our implementation 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 [22].

We evaluated our analysis on DroidLeaks [22], a publicly available resource leaks benchmark. We detected 203 leaks, where 191 are new and undiscovered leaks. We obtained a precision of 72.5%, a recall of 83.2%, and an F-Score of 77.5%.

When considering all the resource leaks included in DroidLeaks that are of one of the four supported types (50 leaks in total), we obtained a bug detection rate of 18% and a false alarm rate of 2%. While the bug detection rate of our tool can improve substantially when compared with the 8 tools considered by DroidLeaks, the false alarm rate is among the best (only Android Lint is better).

*Contributions summary.* Our main contributions can be summarized as follows:

- We present a new context- and flow-sensitive inter-procedural static analysis capable of detecting resource leaks in Android applications. To the best of our knowledge, it is the first IFDS-based resource leak analysis that supports multiple Android resources.
- We provide an implementation of the analysis that can be executed as a standalone tool or run integrated in IntelliJ or Android Studio (as part of the plugin EcoAndroid).
- We extend the DroidLeaks benchmark, with the addition of 191 new resource leaks identified and described, with most of the resource leaks identified concerning the use of Cursor and SQLite Database resources.

All our source code and data are available in EcoAndroid's Github repository: https://github.com/sr-lab/EcoAndroid. EcoAndroid is also available in the JetBrains Marketplace: https://plugins.jetbrains.com/plugin/15637-ecoandroid

## 2 BACKGROUND

In this section, we present background knowledge about the Android architecture, resource leaks, EcoAndroid, and the frameworks used in our development.

### 2.1 Android Architecture

Android applications are built upon four essential components [16, 21, 32]:

(1) **Activity**. This component represents a screen with a user interface and it handles all user interaction.
(2) **Service**. This component runs in the background to perform time-intensive operations and work related to remote processes. It does not provide a user interface.
(3) **Broadcast Receiver**. This component allows an application to receive events from the user or the system.
(4) **Content Provider**. This component is used to manage shared data between multiple applications.

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()` [17, 32]. The complete lifecycle and state transitions of an activity are illustrated in Figure 1.

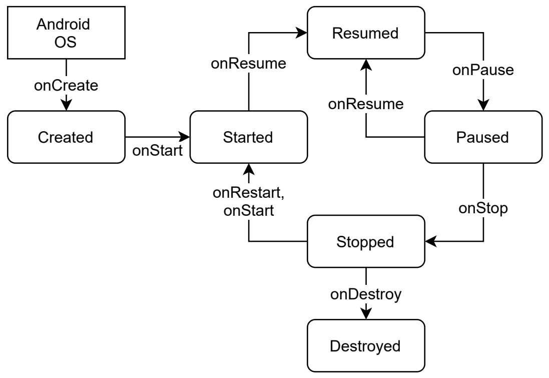


**Figure 1: Android activity lifecycle (adapted from Android Guide [17] and Android Fundamentals [15])**

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 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 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 [17].

The Android framework is mainly event-driven [35]. 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. Event handlers are a more specific type of callback; these 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) [19, 22, 23].

### 2.2 Resource Leaks

As mentioned before, 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 [37]. When a developer wants to use a resource, they must explicitly acquire and release it manually. This is usually done via Android-specific API calls, which vary from resource to resource [22]. A *resource leak* occurs when a programmer forgets to release a resource they previously acquired, after it is done being used. A resource leak causes components to stay active and consume battery, even if they are no longer needed. Apart from the unnecessary battery usage, a resource leak may cause the resource to not function properly for other applications or even cause the Android system to crash [22, 37].

Listing 1 shows an example from an older version of AnkiDroid[1], where a resource — in this case, a database cursor — is *acquired* at

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

```
1  private static SQLiteDatabase upgradeDB(...) {
2      (...)
3      Cursor c = mMetaDb.rawQuery(...);
4      int columnNumber = c.getCount();
5      if (columnNumber > 0) {
6          if (columnNumber < 7) {
7              (...)
8          }
9      } else {
10         mMetaDb.execSQL(...);
11     }
12     mMetaDb.setVersion(databaseVersion);
13     Timber.i(...);
14     // resource leak: missing call to c.close()
15     return mMetaDb;
16 }
```

**Listing 1: Resource leak of a database cursor in an old version of AnkiDroid**

the beginning of a function. The cursor is acquired (line 3), but not closed before exiting the method (line 14)[2].

## 2.3 EcoAndroid

EcoAndroid [29] is an extendable open source Android Studio plugin created to assist developers in creating energy-efficient mobile applications by automatically applying a set of energy patterns to Java source code. At the time of writing, it supports ten different cases of energy-related refactorings, distributed over five energy patterns taken from the literature [13]: *Dynamic Retry Delay*, *Push Over Poll*, *Reduce Size*, *Cache*, and *Avoid Extraneous Graphics and Animations*. In some of these patterns more than one case was implemented (totaling the 10 cases).

EcoAndroid provides two types of warnings: informational warnings and non-informational warnings. The first type does not have an automated refactoring associated. This is because either i) the suggestion is impossible to implement without further information (e.g., in the case of the *Push Over Poll* energy pattern, registration of the class in Firebase is needed) or ii) the required changes affect too much code. For these cases, if the developer wishes to follow EcoAndroid's suggestion and implement the changes manually, the plugin introduces a TODO comment so that the change is listed in the IDE's TODO window. The second type of warning has an automated refactoring associated and will change the code by applying the identified energy pattern.

EcoAndroid is currently available in the JetBrains Marketplace: https://plugins.jetbrains.com/plugin/15637-ecoandroid

## 2.4 Analysis Tools

Our analysis is implemented using Soot [31], FlowDroid [3], and Heros [7], which provide a static-analysis environment capable of processing Android applications and performing inter-procedural analysis with the IFDS framework [28].

**Soot**[3] [31] can be used 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

[2]Commit available at: https://github.com/ankidroid/Anki-Android/commit/3725ce75828aaf4fa0b7bc36416a973f2ea6a157
[3]https://soot-oss.github.io/soot

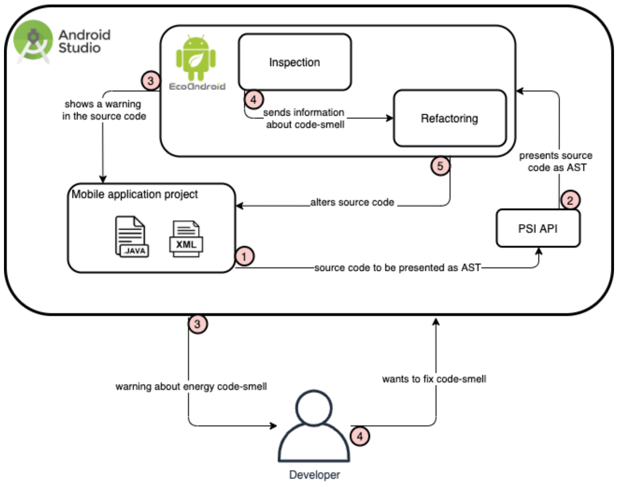

Android Studio
EcoAndroid
Inspection
Refactoring
shows a warning in the source code
sends information about code-smell
presents source code as AST
Mobile application project
alters source code
PSI API
JAVA
XML
source code to be presented as AST
warning about energy code-smell
wants to fix code-smell
Developer
1
2
3
4
5

**Figure 2: EcoAndroid detection and refactoring process**

supports call graph construction, point-to analysis, intra- and inter-procedural data-flow analysis, and taint analysis. Soot uses a few data structures to represent objects used in the 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 about the analysis (e.g., control-flow graphs, call graphs). A `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 contains `Units`. `Units` represent statements (e.g., assignment statements, return statements, and conditional statements) in the code. A single datum is expressed as a `Value`, which can be, e.g., a local (`Local`) or an expression (`Expr`). A `Local` represents a variable in Soot's intermediate representations. The four intermediate representations provided by Soot are: Baf, Jimple, Grimp, and Shimple. The most used intermediate representation is Jimple, which is typed and statement-based. Jimple's succinctness is convenient for performing analysis and optimizations (it has a total of 15 statements, compared to more than 200 instructions in Java bytecode). 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 [3]. 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 APK), and allows the creation of call graphs with knowledge of the callbacks of the Android framework.

**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 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 (we show an example of such a graph in Figure 7). 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**: responsible for passing information from callees to call sites.
- **Call-to-return**: responsible for passing information from before a call site to all possible call site's successor statements. Information passing from these edges typically do not concern the callee.
- **Normal edges**: for all other statements.

In this work, we use **Heros**, a generic IFDS/IDE solver that can be plugged into existing Java-based analysis frameworks [7]. Connecting Heros to a program analysis framework only requires the user to implement a version of the inter-procedural 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 (see Section 3.2 for more details).

## 3 RESOURCE LEAK DETECTION

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 the current 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 of these components is responsible for a specific step in the detection of resource leaks. 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 shows this new user interaction process and operation.

### 3.1 Resource and Leak Representation

Currently, our implementation supports four different Android resources: Cursor, SQLiteDatabase, Wakelock, and Camera. However, our analysis is general and parametric on the resources supported. Figure 4 shows the class diagram of the resource representation. To define a resource, one only needs to specify the fully-qualified class name of the resource (`type`); the names of the methods used to acquire the resource (`acquireOp`); the fully-qualified name of the class where the `acquireOp` can be used (`acquireClass`); the names of the methods used to release the resource (`releaseOp`); the fully-qualified name of the class where the `releaseOp` can be used (`releaseClass`); the name of the method used to check

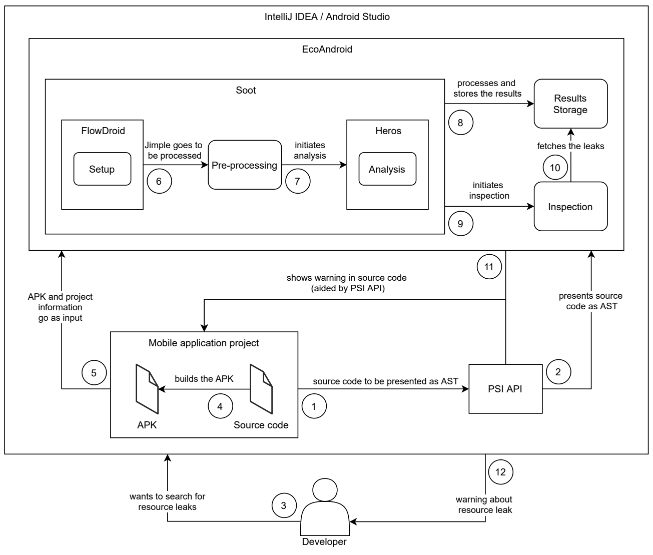

IntelliJ IDEA / Android Studio
EcoAndroid
Soot
FlowDroid
Setup
Jimple goes to be processed
6
Pre-processing
initiates analysis
7
Heros
Analysis
processes and stores the results
8
Results Storage
fetches the leaks
10
initiates inspection
9
Inspection
11
shows warning in source code (aided by PSI API)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

**Figure 3: Extended EcoAndroid detection of resource leaks**

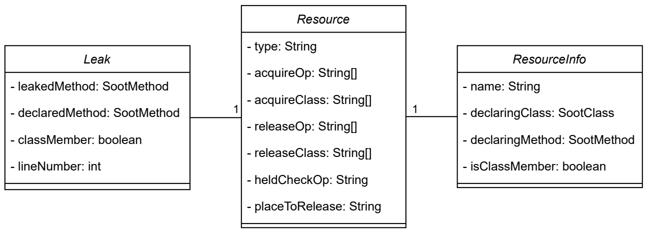


**Figure 4: Class diagram of the implemented data structures: `Resource`, `ResourceInfo` and `Leak` (getters and setters omitted)**

if the resource is acquired (`heldCheckOp`); and the name of the callback method where the resource is supposed to be released (`placeToRelease`) — for example, wakelocks are supposed to be released in `onPause()`.

The analysis will result in a collection of resource leaks. As Figure 4 shows, each `Leak` specifies the `Resource` that was leaked (`resource`). It also identifies the method where the resource leak occurred (`leakedMethod`), the method where the resource was declared (`declaredMethod`), a boolean which indicates if the resource is class-scoped (`classMember`), and the line number in the source code where the resource was declared (`lineNumber`).

### 3.2 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. It is implemented on top of Soot, FlowDroid, and Heros.

As mentioned before, these three frameworks are built to be easily integrated with each other, as they are maintained by the same group of developers. Heros implements a solver for the IFDS framework, and requires the definition of four flow functions. Each flow-function serves a different purpose in the IFDS framework. In our context we have:

- The function `getCallFlowFunction` is responsible for handling flow of facts when a method is called.
- The function `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.
- The function `getCallToReturnFlow` is responsible for acquiring and releasing method-scope resources and also for their correct flow.
- The function `getNormalFlowFunction` handles acquiring and releasing class-scope resources and handles the flow of data-flow facts when dealing with `if` statements.

## 3.3 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, collecting the location of possible leaks, processing them, and presenting the final results to the user.

*3.3.1 Collection of Results.* Considering the properties of our problem, 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 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. In terms of our implementation, 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.

*3.3.2 Processing of Collected Results.* This step is focused 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, since 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).

For example, suppose 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, a naive analysis would collect a leak in `methodB`, since this method does not return the resource and there is a data-flow fact regarding *r* in the method's return statement. Figure 5 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 Algorithm 1 to process the 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

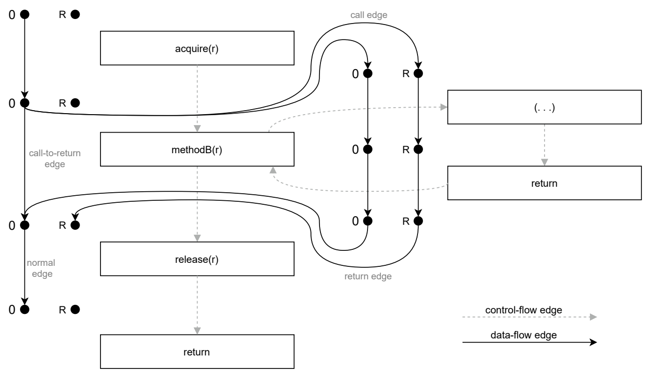


**Figure 5: Example of a false positive of a resource leak**

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.

**Algorithm 1:** 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
```

*3.3.3 Result Storage and Presentation.* We can run our analysis as a standalone application or integrated in an IntelliJ IDEA / Android Studio plugin, such as EcoAndroid. To store the results, we first need to evaluate how we want to present them to the user.

In the standalone version, the results are presented in CSV files. For this purpose, we simply store the leaks in three sets: one for the intra-procedural analysis[4], one for the inter-procedural analysis, and one containing the leaks from both analyses. 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 follow the current methodology of 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

[4]Although we have implemented an intra-procedural analysis, our inter-procedural analysis outperforms it and so the intra-procedural analysis is currently disabled and is not described in this paper.

```
1  public static String getContactName(
2      final Context context, final String address)
3  {
4    //(...)
5    Cursor cursor = getContact(context, address);
6    //(...)
7    return cursor.getString(
8      ContactsWrapper.FILTER_INDEX_NAME);
9  }
10
11 public Cursor getContact(
12     final ContentResolver cr, final String number)
13 {
14   //(...)
15   final Uri uri = Uri.withAppendedPath(
16     Contacts.Phones.CONTENT_FILTER_URL, n);
17   final Cursor c = cr.query(
18     uri, PROJECTION_FILTER, null, null, null);
19   //(...)
20   return c;
21 }
```

**Listing 2: Resource leak (simplified) of an older version of SMSDroid**

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 for checking, in the reported results, if there are any leaks in the visited PsiMethods. At the end of the detection process, we force IntelliJ's 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. Figure 6 shows an example of a leak reported by the extended EcoAndroid.

## 3.4 Illustrative 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 2. This is a cursor leak that spans two different methods, getContact and getContactName, in a version of SMSDroid[5]. In getContact, the cursor c is acquired (line 17) and returned (line 20). The method getContactName then calls getContact (line 5), 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 7, we see the exploded super-graph of this example. The graph provides an overview of all the different types of edges defined in the IFDS framework, and how data flows through 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 other 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.

[5]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

## 4 EVALUATION

This section describes our evaluation methodology and the results obtained. We address the two following research questions:

**RQ1:** How does our tool compare with other resource leak detectors when considering the four types of resource leaks supported?
**RQ2:** Is our tool capable of finding new resource leaks?

## 4.1 Methodology

This subsection describes the datasets used, the data collection and analysis procedures, and the experimental setup.

*4.1.1 Resource Leak Dataset.* To evaluate our work, we use DroidLeaks [22]. 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 of DroidLeaks collected a total of 292 resource leaks from 33 resource classes, which include the 4 resource classes that our implementation currently supports: Cursor, SQLite Database, Wakelock, and Camera.

There is a publicly available website[6] that contains all the information about the dataset. From the available information, there is a spreadsheet[7] containing the 292 identified leaks together with their relevant information: (1) name of the application where the leak was found; (2) the concerned class, i.e. the resource class; (3) the version of the application where the problem was discovered, and the version where the problem was resolved; (4) the problematic method, and the file where this method is implemented; (5) the bug report, if it exists; (6) for the 8 evaluated resource leaks detectors, whether they detected the resource leak or not; (7) information regarding the leak: if it 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 APKs used in the evaluation they performed. There is a total of 137 APKs publicly available[8], including the versions where the leaks were found and the versions where the leaks were fixed.

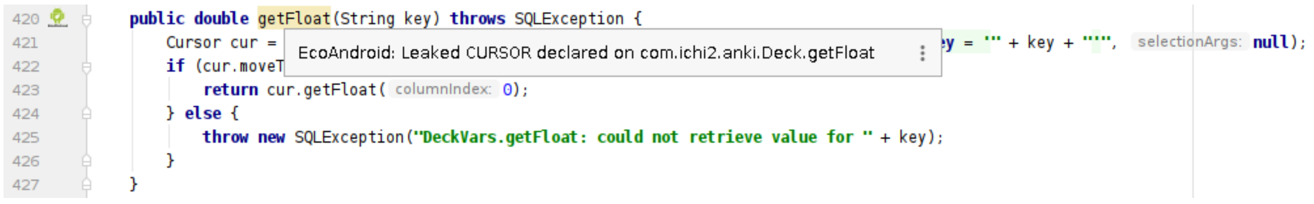
The authors of DroidLeaks also evaluated 8 resource leak detectors — namely, Code Inspections, FindBugs, Infer, Android Lint, PMD[9], Relda2, Elite, and Verifier — with the dataset, with the goal of helping future researchers to create and improve resource leak detection tools. For the evaluation of each tool $t$, the authors defined two metrics: the *Bug Detection Rate*, denoted $BDR(t)$, and the *False Alarm Rate*, denoted $FAR(t)$. These are calculated as follows:

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

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

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 detect since the leak is fixed).

[6]http://sccpu2.cse.ust.hk/droidleaks/
[7]http://sccpu2.cse.ust.hk/droidleaks/project_data/droidleaks.xlsx
[8]http://sccpu2.cse.ust.hk/droidleaks/bugs/apks.php
[9]No data is available for PMD since it does not support any type of system resource covered by DroidLeaks.



```
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 6: Extended EcoAndroid report of a resource leak in the source code**

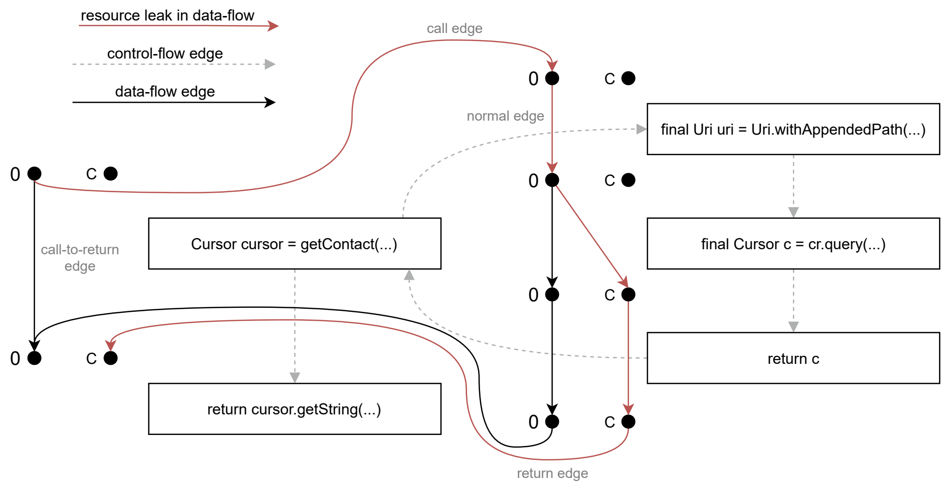

resource leak in data-flow
control-flow edge
data-flow edge
call edge
normal edge
call-to-return edge
return edge
0
c
final Uri uri = Uri.withAppendedPath(...)
Cursor cursor = getContact(...)
final Cursor c = cr.query(...)
return c
return cursor.getString(...)

**Figure 7: Exploded super-graph for the example shown in Listing 2**

*4.1.2 Data Collection and Analysis.* To answer the research questions, we divided our evaluation into two analyses: a *filtered dataset analysis* and a *full dataset analysis*. The first addresses RQ1 and uses a filtered version of DroidLeaks considering only the four types of resource leaks supported. The second addresses RQ2 and uses the entire DroidLeaks dataset.

*Filtered dataset analysis.* The authors of DroidLeaks evaluate only 116 of the 292 resource leaks found, due to the labor-intensive work of compiling all APKs found. From those 116 resource leaks, only 60 are of one of the four types supported by our tool. However, only 50 of these have been used in their experiments. We call the dataset consisting of these 50 leaks the "filtered dataset". Since DroidLeaks provides data on the performance of other resource leak detectors regarding these 50 leaks, we can use this dataset to answer RQ1. For this comparison, we measure the Bug Detection Rate and the False Alarm Rate. Table 1 shows, regarding the filtered dataset, the number of leaks from each resource class, as well as the applications where they were identified.

*Full dataset analysis.* To address RQ2, we run our analysis on the 137 APKs provided by DroidLeaks and count the number of previously unidentified leaks (i.e. leaks identified and confirmed by us as true positives that are not listed in DroidLeaks).

We also measure three metrics: precision, recall, and F-Score [9]. Finally, regarding performance, we calculate the average and median time that our tool takes to analyze the provided applications.

*Experimental setup.* We ran our tool 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 APKs from DroidLeaks
(2) Collect and organize the obtained results into a spreadsheet

| 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 1: Filtered dataset: distribution of resource classes**

(3) Compare the obtained results with the filtered 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 filtered dataset)
(5) Calculate the analysis' detection rate and compare with the tools evaluated in DroidLeaks, from the filtered dataset
(6) Calculate the remaining efficiency metrics — precision, recall, and F-Score — for the full dataset analysis and false negatives obtained from DroidLeaks
(7) Calculate performance metrics — average and median duration of the analysis — for the full dataset analysis.

## 4.2 Results

*4.2.1 Errors in the Analysis.* From the 137 APKs 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 are shown in Table 2. For these applications, our analysis is unable to run and detect resource leaks. Regarding evaluation on the filtered 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 full dataset analysis, this means that we only consider 107 out of the 137 APKs provided by DroidLeaks.

*4.2.2 Filtered Dataset Analysis.* For the 50 resource leaks in the filtered 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 these results and observed that, for the 41 that our tool failed to detect, the main reason is due to Soot and Heros not analyzing the method where the resource was leaked, which happened in 25 (61%) of the leaks. Table 3 shows the causes for failure to detect the leaks in the filtered 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).

| 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 2: Applications where the generation of call graphs failed**

| 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 3: Causes for failing to detect leaks in filtered dataset**

As mentioned before, the authors of DroidLeaks performed an evaluation of 8 resource leak detectors using their dataset. Table 4 shows how the tools evaluated in DroidLeaks and our tool (named EcoAndroid) performed on the filtered dataset. Note that Relda2 supports two analysis modes: flow-sensitive and flow-insensitive. In the table, Relda2-FS and Relda2-FI represent the two modes, respectively. Also, FindBugs is not included in the table since there is no data available about this tool for the leaks in the filtered dataset. All the 9 leaks detected by EcoAndroid are also detected by at least one other tool; however, there is no other tool that detects all of these 9 leaks.

**Answer to RQ1. How does our tool compare with other resource leak detectors when considering the four types of resource leaks supported?**
While the bug detection rate of our tool can improve substantially, the false alarm rate is among the best (only Android Lint is better). We noticed that the main problem is with Soot and Heros not analyzing methods where resources are leaked. Further work to fix this single problem can have a big impact on the performance of our tool.

*4.2.3 Full Dataset Analysis.* Given the errors mentioned above, we analyzed a total of 107 APKs. 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 the number of false negatives calculated using information from the filtered dataset) of 83.2%, and an F-Score of 77.5%.

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 the number

| **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 4: Filtered dataset analysis: tool performance**

| | **Full reported leaks** | **Unique reported leaks** |
|---|---|---|
| Total apps analyzed | 107 | 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 filtered dataset) | |
| Precision | 0.725 | 0.754 |
| Recall | 0.832 | 0.677 |
| F-Score | 0.775 | 0.714 |

**Table 5: Results obtained from the full dataset analysis**

of false negatives calculated using information from the filtered dataset) of 67.7%, and an F-Score of 71.4%. Table 5 summarizes the results obtained. Notice that these results indicate that our tool detected 191 previously unidentified leaks (74 when considering unique reported leaks). This is because the 203 true positives detected by EcoAndroid include the 9 leaks that were detected in the filtered dataset. Note that the 50 leaks in the filtered dataset are the only ones identified in DroidLeaks as leaks of the four types supported by EcoAndroid. Therefore, the 191 leaks detected further to the 9 leaks of the filtered dataset where not identified before. Moreover, there are 3 leaks detected by EcoAndroid for which there is no data regarding other tools.

Table 6 shows the results obtained from the full dataset 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.

To evaluate the performance of our tool, we recorded the time it 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 7 shows these recorded times, as well as total time, presented in milliseconds and in minutes.

**Answer to RQ2. Is our tool capable of finding new resource leaks?**
Yes, our tool found 191 resource leaks previously unidentified in DroidLeaks (74 when considering unique reported leaks). Most of the resource leaks identified concern the use of Cursor and SQLite Database resources. Concerning the Camera resource, there were no resource leaks found.
We obtained a precision of 72.5%, a recall (with the number of false negatives calculated using information from the filtered dataset) of 83.2%, and an F-Score of 77.5%. When considering unique reported leaks, these values changed to 75.4%, 67.7%, and 71.4%, respectively.

## 5 RELATED WORK

Jiang et al. [20] list typical energy bugs, divided into resource leaks (also called no-sleep bugs) and layout defects. Pathak and Jindal [27] divide 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 is carried out by different threads in the application), and no-sleep dilation (i.e. when a component is put to sleep later than necessary). Cruz and Abreu [13] present 22 energy patterns for Android applications, with some involving resource leaks.

Regarding leak detectors, Vekris et al. [32] created a tool to verify if an Android application complies with a set of energy policies, focused only on the acquiring and releasing of wakelocks. Guo et al. [19] created Relda, which 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. Wu et al. [36] present Relda2, which unlike most tools that are built on top of frameworks like Soot and WALA, analyzes Dalvik bytecode directly, leveraging only Androguard to disassemble the app into the Dalvik bytecode. The Automated Android Energy-Efficiency InspectiON (AEON) [30] is an IntelliJ IDEA plugin capable of inspecting energy problems related to the Android API. The plugin focuses on wakelocks. AEON was used in the work of Deng et al. [14] to design the WakeLock Release Deletion mutation operator, used to mimic an energy bug. Wu et al. [35] detect two patterns related to resource leaks: activity adding a listener but not removing it, and activity adding a listener but putting it in a long-wait state.

| 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 6: Results obtained from the full dataset analysis, organized per resource**

| | 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 7: Time performance of the analysis**

Liu et al. [24] created a technique called Elite capable of detecting common wakelock misuses. Elite first decompiles the application's APK files to Java bytecode using Dex2jar, and then performs an analysis with the help of Soot and Apache Byte Code Engineering Library (BCEL). Jiang et al. [20] built a tool called SAAD. 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. Banerjee et al. [4] expand on their previous work [5] and create a framework for detecting resource leaks and implement it into an Eclipse plugin called EnergyPatch. More recently, Bhatt and Furia [6] implement PlumbDroid, which builds several resource-flow graphs (an abstraction based on control-flow graphs) that captures information about the acquiring and releasing of resources. 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.

When compared to the above work, our analysis is the only IFDS-based resource leak analysis that supports multiple Android resources.

## 6 CONCLUSION

This paper presents an extension of EcoAndroid with automated detection of resource leaks in Android applications. We designed and implemented a 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 as part of EcoAndroid, in IntelliJ IDEA or Android Studio, or as a command-line tool. When using our tool to analyze 107 Android applications from the DroidLeaks dataset, we have been able to detect 191 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.

### 6.1 Future Work

*Improve the use of the static analysis frameworks.* While static analysis frameworks like Soot and Heros provide tools to build static analyses, it might not be trivial to implement analyses that work for all cases. For example, 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 we observed is that some generated call graphs were incorrect (e.g., they were incomplete and did not contain the method where the resource leak occurred). When this happens it is still possible to run the analysis but this can cause false positives or false negatives. As the next step, we plan to improve this and fix this problem.

*Support diverse 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. For example, the `ContentQueryMap` is such a mechanism and it is used to cache the contents of a cursor into a map. It works by passing the cursor to the `ContentQueryMap` constructor, performing all the operations needed, and then closing the `ContentQueryMap` [18]. We plan to improve our tool to take into account as many of these mechanisms as possible.

*Repair of resource leaks through refactoring.* A useful improvement to our work would be to automatically repair the detected leaks through refactoring. This would require a careful analysis of where to release each resource, so that the refactoring would not impact the rest of the application. The same mechanism used in EcoAndroid to refactor energy patterns could be used.

*User study.* Since our tool can run as part of EcoAndroid, we plan to perform a user study to assess the usability of our extension and to collect feedback that can be used to improve our work.

## ACKNOWLEDGMENTS

This work was partially funded by national funds through FCT under projects UIDB/50021/2020 and PTDC/CCI-COM/29300/2017.

## REFERENCES

[1] Shaukat Ali, Shah Khusro, Azhar Rauf, and Saeed Mahfooz. 2014. Sensors and mobile phones: evolution and state-of-the-art. *Pakistan journal of science* 66, 4 (2014), 385.

[2] Appbrain. 2022. Number of Android apps on Google Play. https://www.appbrain.com/stats/number-of-android-apps. Accessed: 01-02-2022.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices* 49, 6 (2014), 259–269.

[4] Abhijeet Banerjee, Lee Kee Chong, Clément Ballabriga, and Abhik Roychoudhury. 2017. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. *IEEE Transactions on Software Engineering* 44, 5 (2017), 470–490.

[5] Abhijeet Banerjee, Lee Kee Chong, Sudipta Chattopadhyay, and Abhik Roychoudhury. 2014. Detecting energy bugs and hotspots in mobile apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 588–598.

[6] Bhargav Nagaraja Bhatt and Carlo A Furia. 2020. Automated Repair of Resource Leaks in Android Applications. *arXiv preprint arXiv:2003.03201* (2020).

[7] Eric Bodden. 2012. 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*. 3–8.

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

[9] Center for Assured Software. 2011. *CAS Static Analysis Tool Study - Methodology.* Technical Report. National Security Agency, 9800 Savage Road Fort George G. Meade, MD 20755-6738.

[10] Melissa Chau and Ryan Reith. 2020. Smartphone market share. https://www.idc.com/promo/smartphone-market-share/os. Accessed: 19-12-2020.

[11] J Clement. 2021. 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: 01-02-2022.

[12] Marco Couto, João Saraiva, and João Paulo Fernandes. 2020. 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, 217–228.

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

[14] Lin Deng, Jeff Offutt, and David Samudio. 2017. Is mutation analysis effective at testing android apps?. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 86–93.

[15] Google. 2021. Android fundamentals 02.2 - Activity lifecycle and state. https://developer.android.com/codelabs/android-training-activity-lifecycle-and-state#0. Accessed: 01-02-2022.

[16] Google. 2021. Application Fundamentals - Android Developers. https://developer.android.com/guide/components/fundamentals. Accessed: 01-02-2022.

[17] Google. 2021. Understanding the Application Lifecycle - Android Developers. https://developer.android.com/guide/components/activities/activity-lifecycle. Accessed: 01-02-2022.

[18] Google. 2022. Android Reference - Content Query Map. https://developer.android.com/reference/android/content/ContentQueryMap. Accessed: 01-02-2022.

[19] Chaorong Guo, Jian Zhang, Jun Yan, Zhiqiang Zhang, and Yanli Zhang. 2013. Characterizing and detecting resource leaks in Android applications. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 389–398.

[20] Hao Jiang, Hongli Yang, Shengchao Qin, Zhendong Su, Jian Zhang, and Jun Yan. 2017. Detecting energy bugs in Android apps using static analysis. In *International Conference on Formal Engineering Methods*. Springer, 192–208.

[21] Li Li, Tegawendé F Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Le Traon. 2017. Static analysis of android apps: A systematic literature review. *Information and Software Technology* 88 (2017), 67–95.

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

[23] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*. 1013–1024.

[24] Yepang Liu, Chang Xu, Shing-Chi Cheung, and Valerio Terragni. 2016. Understanding and detecting wake lock misuses for android applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 396–409.

[25] David Lo, Nachiappan Nagappan, and Thomas Zimmermann. 2015. How Practitioners Perceive the Relevance of Software Engineering Research. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 415–425. https://doi.org/10.1145/2786805.2786809

[26] S O'Dea. 2020. Number of smartphone users worldwide from 2016 to 2021. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide. Accessed: 19-12-2020.

[27] Abhinav Pathak, Abhilash Jindal, Y Charlie Hu, and Samuel P Midkiff. 2012. 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*. 267–280.

[28] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61.

[29] Ana Ribeiro, João F. Ferreira, and Alexandra Mendes. 2021. EcoAndroid: An Android Studio Plugin for Developing Energy-Efficient Java Mobile Applications. In *2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS)*. 62–69. https://doi.org/10.1109/QRS54544.2021.00017

[30] David Samudio. 2016. Automated Android Energy-Efficiency InspectiON. https://plugins.jetbrains.com/plugin/7444-aeon-automated-android-energy-efficiency-inspection/.

[31] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.

[32] Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. 2012. 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*.

[33] Jingtian Wang, Guoquan Wu, Xiaoquan Wu, and Jun Wei. 2012. 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*. 1–5.

[34] Claas Wilke, Sebastian Richly, Sebastian Götz, Christian Piechnick, and Uwe Aßmann. 2013. 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, 134–141.

[35] Haowei Wu, Shengqian Yang, and Atanas Rountev. 2016. Static detection of energy defect patterns in Android applications. In *Proceedings of the 25th International Conference on Compiler Construction*. 185–195.

[36] Tianyong Wu, Jierui Liu, Xi Deng, Jun Yan, and Jian Zhang. 2016. 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, 762–767.

[37] Tianyong Wu, Jierui Liu, Zhenbo Xu, Chaorong Guo, Yanli Zhang, Jun Yan, and Jian Zhang. 2016. Light-weight, inter-procedural and callback-aware resource leak detection for Android apps. *IEEE Transactions on Software Engineering* 42, 11 (2016), 1054–1076.