Extending EcoAndroid with Automated Detection of Resource Leaks

Ricardo Pereira
ricardobaiapereira@tecnico.ulisboa.pt
Instituto Superior Técnico
Lisbon, Portugal

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.

CCS CONCEPTS

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

KEYWORDS

energy consumption, static analysis, resource leaks, android

1 INTRODUCTION

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 [20]. 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% [7]. The market for Android applications has also grown throughout the years, totaling a number of 3 million applications on the Google Play Store [2, 8].

Recent research has been uncovering energy problems and inefficiencies, created by application developers, that decrease the battery life of Android devices [3, 5, 25]. 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 [26].

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 [1]. 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 API provided by Android; however, if not well implemented, this can have huge costs on the battery life of the device [18]. 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 [4, 18, 28], but not always have researchers been able to find resource leaks in applications [9].

2 OBJECTIVES AND CONTRIBUTIONS

The main goal of this project is to extend EcoAndroid [22] – 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 inter-procedural static analysis capable of detecting resource leaks in Android applications, integrated in an IntelliJ plugin. Currently, our analysis supports the detection of 4 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[18].

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 IDE: IntelliJ and Android Studio
- the extension of the DroidLeaks benchmark, with the addition of 194 new resource leaks identified and described

3 ANDROID ARCHITECTURE

Android applications are built upon four essential components [13, 17, 24]. Figure 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.

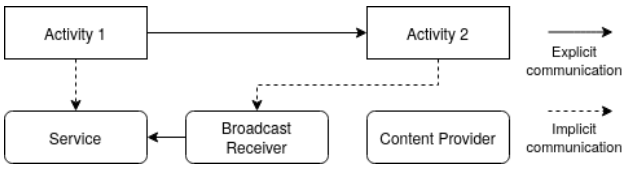

Figure 1: Android component communication (adapted from Li et al. [17])

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()` [14, 24]. The complete lifecycle and state transitions of an activity are illustrated in Figure 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 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 [14].

The Android framework is mainly event-driven [27]. 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

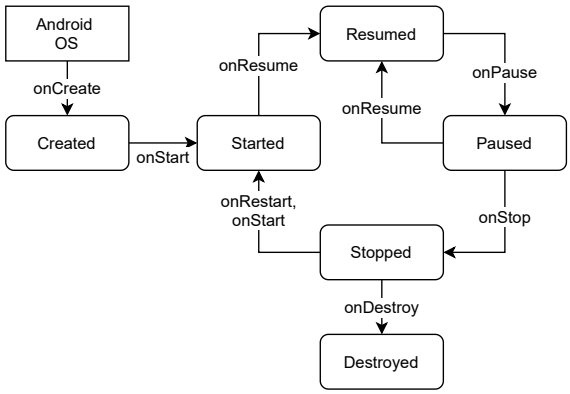

Figure 2: Android activity lifecycle (adapted from Android Guide [14] and Android Fundamentals [12])

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) [15, 18, 19].

4 RESOURCE LEAKS

```
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     //leak! missing call to c.close()
15     return mMetaDb;
16 }
```

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

As introduced in this work, 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 [29]. 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 [18]. Here, we show an example from an older version of AnkiDroid[1]. In Listing 1, we see a resource – in this case, a database cursor – being acquired at the beggining of a function.

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

The developer performs some operations but, at the end of the function, forgets to close the database cursor, creating a resource leak [2]. 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 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 [18, 29].

5 IMPLEMENTATION

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.

5.1 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. It is implemented on top 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. Heros implements a solver for the IFDS framework, and provides the four flow functions we need to implement in our work. Each flow-function serves a different purpose in the IFDS framework. Our implementation of them reflects this fact, as we describe their use in our work next:

- In the `getNormalFlowFunction`: handles acquiring and releasing class-scope resources and to handle the flow of data-flow facts when dealing with `if` statements.
- In the `getCallFlowFunction`: is responsible for handling flow of facts when a method is called
- `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.
- `getCallToReturnFlow`: is responsible for acquiring and releasing method-scope resources and also for their correct flow, in conjunction with `getReturnFlowFunction`.

5.2 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.

5.2.1 Collection of 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 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.

5.2.2 Processing of 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.

With this problem in mind, we developed an algorithm 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 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.

5.2.3 Result Storage and Presentation. We 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 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,

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

```
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

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.

5.3 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 [3]. 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 3, 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 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.

[3]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

6 EVALUATION

6.1 Methodology

6.1.1 Resource Leak Dataset. Researchers have created public datasets containing resource leaks in multiple applications. We have chosen DroidLeaks [18] 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 ahppens 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:

$$BDR(t) = \frac{\text{\# bugs detected by t on buggy app versions}}{\text{\# bugs experimented on t}} \quad (1)$$

$$FAR(t) = \frac{\text{\# false alarms reported by t on patched app versions}}{\text{\# bugs experimented on t}} \quad (2)$$

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 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 [4] that contains all the information about the dataset. From the available information, there is a spreadsheet [5] 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

[4]http://sccpu2.cse.ust.hk/droidleaks/

[5]http://sccpu2.cse.ust.hk/droidleaks/project_data/droidleaks.xlsx

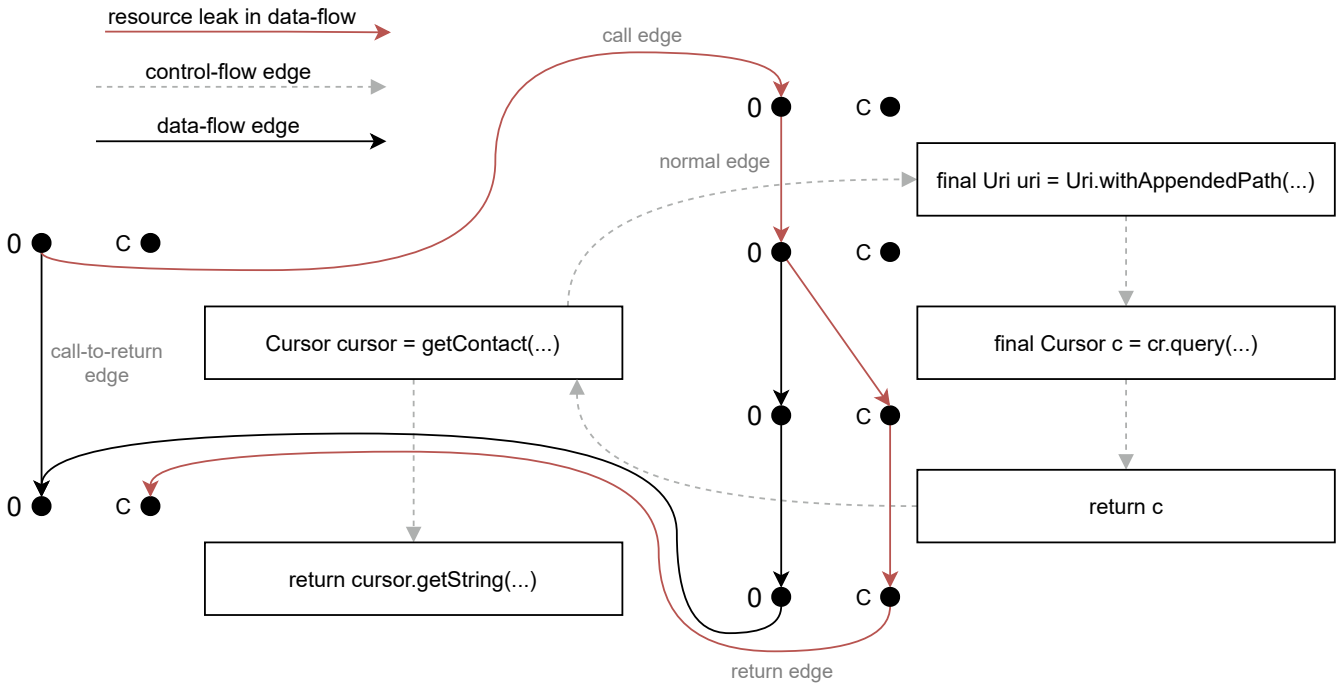

Figure 3: Exploded super-graph for the example in Listing 2

Resource class	# leaks	Related applications
`Cursor`	38	, 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: Distribution of the subset of resource leaks evaluated

- 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[6] – 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.

6.1.2 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 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 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 [6]. This 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

[6] http://sccpu2.cse.ust.hk/droidleaks/bugs/apks.php

(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.

6.2 Results

6.2.1 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 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 2: Applications that suffered from call graph generation failure

6.2.2 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 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 3: Causes for failing to detect leaks in reduced dataset

As mention before, the authors of DroidLeaks performed an evaluation of 8 resource leak detectors with their dataset, calculating their detection rate. Table 4 shows how the tools evaluated in DroidLeaks and EcoAndroid performed on the reduced dataset.

6.2.3 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%.

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 summarizes the results obtained and the efficency metrics calculated regarding the full analysis.

Table 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.

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

	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: Results obtained from full analysis

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 7 shows this recorded times, as well as total time, presented in milliseconds and in minutes.

7 CONCLUSION

The main objective for this work 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 EcoAndroid, in IntelliJ IDEA or Android Studio, and 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.

8 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. 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.

2*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 full 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

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.

9 RELATED WORK

Cruz and Abreu [10] 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.

Jiang et al. [16] 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 [21] 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).

Liu et al. [18] create DroidLeaks, a benchmark of 292 leaks from 33 different resource classes, contained in 32 pouplar and large-scale open-source 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.

Guo et al. [15] 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. [28] develop a tool called Relda2 (the successor of the aforementioned Relda [15]) 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. [24] 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.

The Automated Android Energy-Efficiency InspectiON (AEON) [23] 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. [11] to design the WakeLock Release Deletion mutation operator, used to mimic an energy bug.

ACKNOWLEDGMENTS

To Professor João Ferreira for his guidance during this work. To my family, friends, and girlfriend Sara for their immensurable support throughout the years.

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. [n. d.]. Number of Android apps on Google Play. https://www.appbrain.com/stats/number-of-android-apps. Accessed: 19-12-2020.
[3] 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.

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

[5] 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.

[6] 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.

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

[8] J Clement. [n. d.]. 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.

[9] 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.

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

[11] 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.

[12] Google. [n. d.]. Android fundamentals 02.2 - Activity lifecycle and state. https://developer.android.com/codelabs/android-training-activity-lifecycle-and-state#0. Accessed: 15-12-2020.

[13] Google. [n. d.]. Application Fundamentals - Android Developers. https://developer.android.com/guide/components/fundamentals. Accessed: 15-12-2020.

[14] Google. [n. d.]. Understanding the Application Lifecycle - Android Developers. https://developer.android.com/guide/components/activities/activity-lifecycle. Accessed: 15-12-2020.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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*. 1013–1024.

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

[21] 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.

[22] Ana Ribeiro, Joao F Ferreira, and Alexandra Mendes. 2021. 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.

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

[24] 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*.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.