International Journal Of Engineering Research And Development
e- ISSN: 2278-067X, p-ISSN: 2278-800X, www.ijerd.com
Volume 20, Issue 10 (October, 2024), PP 141-168

Consolidating Categories of Resource Leaks in Mobile Applications from Secondary Studies

Josias Gomes Lima [1],*, Joethe Moraes de Carvalho [1],, Oswald Mesumbe Ekwoge [2],, Rafael Giusti [1], and Arilo Claudio Dias Neto [1]

[1]*Federal University of Amazonas, Amazonas, Brazil; josias@icomp.ufam.edu.br, joethe@icomp.ufam.edu.br, rgiusti@icomp.ufam.edu.br and ariloclaudio@gmail.com*

[2] *Sidia Institute of Science and Technology, Amazonas, Brazil; ome@icomp.ufam.edu.br*

**Correspondence: josias@icomp.ufam.edu.br*

Abstract: *Context Resource leak occurs when a developer does not correctly release the resources acquired (e.g., camera and sensors) by the mobile application. This error can lead to performance degradation or crashes. Objective To identify, analyze, and synthesize categories of leak causes. Method A systematic mapping study on resource leak in applications was carried out based on a collection of 57 papers (from the 1760 papers). Results We identified 10 categories of leak causes, and some observed trends were derived, for example, in terms of types of research and types of resources. In addition, an opinion survey with developers was carried out in order to assess the perceived relevance of the 10 categories of leak causes identified in the mapping. Conclusions The results show categories of leak causes in applications that can serve as a guide to areas that require more attention from developers and the research community.*

Keywords: *systematic mapping, mobile apps, resource leak, opinion survey*

Date of Submission: 11-10-2024 Date of Acceptance: 22-10-2024

I. Introduction

Mobile devices have become an important part of the daily lives of many people around the world. The number of smartphone subscriptions worldwide is expected to surpass 7.2 billion in 2024 and that number is expected to grow in coming years [1]. In 2023, users downloaded nearly 257 billion mobile applications on their devices, compared to

104.7 billion in 2016 [2]. The growth in the number of devices with different configurations presents numerous challenges for the correction and good performance of the applications. This is because, in addition to the traditional defects, there are defects related to the use of the limited resources of the devices.

Mobile devices have several resources such as microphone, GPS, memory, camera, NFC (Near Field Communication), sensors, and bluetooth. When developers do not correctly manage these resources in their applications, for example, not releasing the resource after its use, it may cause a failure known as **resource leak**. This error may lead to crashes, poor responsiveness, unnecessary battery consumption, and overall negative user experience [3].

This paper aims to explore and synthesize categories of leak causes in mobile applica- tions. For this purpose, a systematic mapping study was carried out, where 1760 papers (primary studies) were analyzed. After the first and second filters, 57 papers remained, which were synthesized based on 5 facets, including types of contributions, of research, of resource leakage that the study can identify, the categories of causes of resource leaks, and the input artifact. To analyze the relevance of the 10 categories of causes of resource leak identified in the mapping, we conducted an opinion survey with developers. The 6 categories perceived as the most relevant and a new category were identified. Main contributions of this work is **the identification and evaluation of categories of causes of resource leaks in mobile applications**.

The remainder of this paper is organized as follows: related works are presented in Section 2. Section 3 provides an overview of the basics of resource leaks in mobile applications. Section 4 details the main procedures we followed to carry out a systematic mapping study, as well as showing the results obtained. Section 5 presents the planning, execution and results of the survey with the developers. Finally, Section 6 presents the conclusion and future work.

II. Related Works

There are different research studies on resource leaks. This section will show some of the recent research.
[4] present 11 categories of memory leaks in *C* programs from the perspectives of heap memory behavior and program structures. As well, they designed and implemented a standards-based system for generating a set of program data called HPMD (Heap Program Memory Dataset) that contains a variety of memory leaks.

Memory leaks were the subject of research by [5], who carried out a systematic map- ping study over a stipulated period of 35 years, from which 105 papers were selected. Among the results, we highlight the 3 types of memory leak detection techniques found: Dynamic, Static and Dyn-stat, which is a technique that implements the 2 previous char- acteristics. The types of studies detected were analytical, empirical and hybrid, the latter being the most used throughout the research period. The authors also listed the motivations and areas related to the study of the subject in categories, evidencing the search for new algorithms and areas related to Computer Science.

[6] analyze different types of power leaks in Android apps and how these leaks affect device power consumption. As well as discussing how these energy leaks can be avoided in the development phase.

[7] discuss static analysis methods for memory leak detection in *C* and *C++* languages. They noted that these methods often have two stages. First, they use different types of graphs to find memory leaks faster but less accurately. Then, for detected cases, they use path-sensitive analysis to increase accuracy. The authors also listed 9 methods to detect the leaks, namely, *SMOKE*, *PCA*, *SVF*, *Pinpoint*, *Sparrow*, *Fastcheck*, *CSA*, *PML Checker* and *Infer*. Each method has its characteristics and applicability, which were individually tested to find bugs, where false positive and false negative results were found. As a result, *SMOKE* and *CSA* have the best false positive rate for *C* language, *CSA* has the best false positive for *C++* language. As for false negatives, it was mentioned that *SMOKE* and *PCA* were better in *C* and *SMOKE* for *C++*.
Our paper differs from related works because it carried out a systematic mapping in order to identify categories of leak causes in mobile applications.

III. Resource Leak in Mobile Applications

Resource leak is caused when an application does not release resources that it acquired during its execution [8]. Properly managing resources is not a trivial task for developers [9]. This becomes more significant as the capabilities of Android devices and the complexity of their software continue to grow rapidly. This growth presents significant challenges for the software correctness and performance [10]. For a better understanding, 14 types of resources were identified, based on the systematic mapping performed. Below are some examples of resources and types.
Concurrency: are the classes of resources that allow the execution of tasks in an interleaved way in the same time interval. Some examples of classes of this type of resource are: *java.lang.Thread*, *java.util.concurrent.Semaphore* and *android.os.Binder*.
Connectivity: are the resource classes related to connecting to other devices, for example, bluetooth (*android.bluetooth.BluetoothAdapter*), USB (*android.hardware.usb*) and Wifi (*android.net.wifi.WifiManager*).
Database: are resource classes that manipulate application data, such as *android .database.DatabaseUtils* and *android.database.sqlite.SQLiteDatabase.Cursor*.

Files: are resource classes related to writing and reading files, for example, *java.io.InputStream*, *java.util.Scanner* and *java.io.FileOutputStream*.
Localization: are the resource classes that manipulate the user's location, for exam- ple, *android.location.LocationProvider*, *android.location.LocationListener* and *android.location .LocationManager*.
Memory: are the resource classes related to memory consumption, for example, *android.app.Activity*, *android.app.Fragment* and *android.content.Context*.
Multimedia: are the classes of resources that manipulate the device media, for exam- ple, *android.media.MediaPlayer*, *android.media.AudioManager* and *android.hardware.Camera*.
Network: are the classes of resources related to internet connection, for example, *org .apache.http.impl.client.AndroidHttpClient*, *java.net.Socket* and *java.net.Network*.
Screen: are the classes of resources that relate to what is displayed on the user's screen, for example, *android.view.MotionEvent*, *android.text.TextUtils* and *android.appwidget*.
Security: are the resource classes that deal with application security, for example, the package classes *android.security*.
Sensor: are the resource classes related to the device's sensors, for example, *android .hardware.SensorManager*, *android.hardware.Sensor* and *android.os.Vibrator*.
Service: are the classes of resources that help the application run as a service, for example, *android.app.Service*.

WakeLock: are the resource classes that help the application to run even with the cell phone blocked, for example, the class *android.os.PowerManager.WakeLock*.
Energy: are the resources related to power consumption.
Unfortunately, resource management bugs are common in Android programs [11]. It should be noted that, in addition to the traditional defects, it is also important to consider defects that cause excessive consumption of the limited resources available on mobile devices [10], such as media players, memory, camera and sensors. An advantage would be, if developers understood all the relevant API contracts, but even the most experienced developers may fail to free up all resources over the term of all possible invocation sequences from event handlers [11].

IV. Systematic Mapping

In the technical literature, no analysis was found on this topic, so it was decided to perform this Systematic Mapping Study (SMS), which was conducted according to the guidelines outlined in [12] and [13]. The SMS is composed of the following main steps:

- Definition of research questions;
- Identification of studies;
- Definition of data extraction strategy;
- Execution of systematic mapping.

4.1 Methods

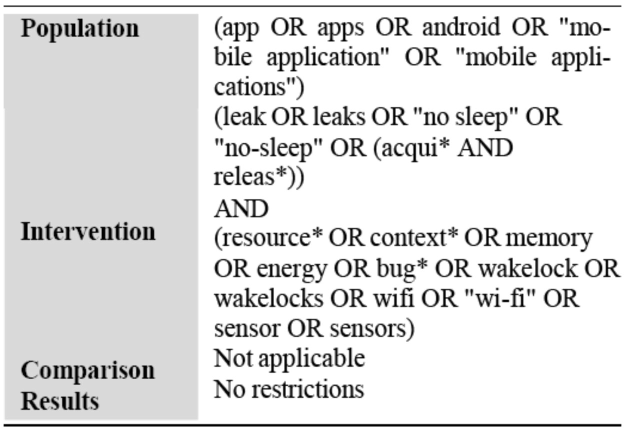
4.1.1 Goal and Research Questions

The objective of this study is to identify, analyze, and synthesize categories of leak causes in scientific publications related to the identification of resource leaks in mobile applications. Based on the research objective, the main research question (RQ 1) was formulated. To extract detailed information, the question was divided into several sub- questions, as described below.
RQ 1 - systematic mapping: What is the space for research of the technical literature in identifying resource leak in mobile applications? To answer this question, the following research subquestions have been proposed:
RQ 1.1 - contribution types: How many papers present methods/techniques, tools, models, metrics, or processes for identifying resource leaks? The SMS guidance document described in [14] proposes the types of contributions mentioned above. The response to this RQ will allow us to assess whether the community, as a whole, was more focused on developing new identification techniques or more focused on developing new identification tools;
RQ 1.2 - research types: What types of research are used in studies in this area? The SMS guidance document described in [14] proposes the following types of research: solution proposal, validation research, evaluation research, and experience reports. The logic behind this RQ is to understand the maturity of the field in the use of empirical approaches;
RQ 1.3 - resource types: What types of resources are identified as having leaks? Some papers detect a leak in only one resource, such as WakeLock. Others detect in various resources like camera, sensor and location;
RQ 1.4 – category of leak causes: What are the categories of leak causes? This RQ helps understand the causes of resource leak;
RQ 1.5 - input artifact: Which artifact do methods/techniques receive as input? Some examples of input artifacts are apk, source code, and running application.

4.1.2 Identification of Studies

The search string was modeled after the PICO analytical structure (*Population*, *Inter- vention*, *Comparison* and *Outcomes*), according to the recommendations described by [12]. In order to improve the string, 8 control papers [3,11,15–20] were used, with keyword synonyms identified. Execution cycles of the search string for refinements in the Scopus[1] digital library were also carried out. The referred string is shown in Table 1.

Table 1. The string in the PICO structure for the mapping study performed



Population	(app OR apps OR android OR "mobile application" OR "mobile applications")
Intervention	(leak OR leaks OR "no sleep" OR "no-sleep" OR (acqui* AND releas*)) AND (resource* OR context* OR memory OR energy OR bug* OR wakelock OR wakelocks OR wifi OR "wi-fi" OR sensor OR sensors)
Comparison	Not applicable
Results	No restrictions

The search for studies, using the string, was carried out in the following digital libraries: Scopus, IEEE Xplore[2], Science Direct[3], Engineering Village[4], Web of Science[5] e ACM DL[6]. These digital libraries were selected based on the experience report by [12]. The studies returned by the digital libraries will be evaluated and/or analyzed only if they meet all the inclusion criteria, which are:

- Studies related to the identification of resource leaks in mobile applications;
- Papers available on the web or by contacting the authors. The following exclusion criteria are used:
- Non-English papers;
- Duplicate papers.

1 https://www.scopus.com/search/form.uri?display=basic

2 https://ieeexplore.ieee.org/Xplore/home.jsp

3 https://www.sciencedirect.com/

4 https://www.engineeringvillage.com/search/quick.url

5 http://www.webofknowledge.com

6 https://dl.acm.org

4.1.3 Classification Scheme

A classification scheme is derived from a careful analysis of the primary studies in order to help with the categorization of papers. The classification for RQ 1.1 was based on [21], where a *framework* is a detailed method that has a wide purpose and focuses on multiple questions or areas of research, a *method* usually has a more specific goal and a narrow research question or purpose, a *model* provides an abstract classification or model of a topic and problem, rather than a specific, tangible way to solve a specific problem, the *tool* is when the study provides a tool, *evaluation* which includes papers that evaluate an already published concept and do not introduce new concepts or solutions, the *metric* provides guidelines on how to measure aspects of the application. The *database* category was also added, which refers to an organized collection of information composed of related records.

RQ 1.2 (types of research) was organized according to [14], where papers that have only examples and a good line of argument are categorized as *Solution Proposal*; papers whose techniques have not yet been implemented in practice, that is, only with experiments carried out in laboratory are classified as *Validation Research*; if the technique is implemented in practice and has a technical evaluation showing how the study was conducted, what are the benefits and disadvantages (including the identification of problems in the industry), it is classified as *Evaluation Research*; papers that only report applications or experiences in practice, based on the author's personal experience, are classified as *Experience Reports*.

RQ 1.3 (resource types) was structured in 14 resource types (*concurrency*, *connectiv- ity*, *database*, *files*, *localization*, *memory*, *multimedia*, *network*, *screen*, *security*, *sensor*, *service*, *wakeLock*, *energy*) based on [9–11], which are described in Section 3.

For the construction of the RQ 1.4 classification scheme, an initial version was created from the control studies, then evolved during data extraction, through the extracted at- tributes and iterative refinement steps. Adding new categories or merging existing ones. Therefore, the following 10 categories were defined: *Complete leak*, *Leak on normal paths*, *Leak on exceptional paths*, *Leak on irregular paths*, *Leak caused by race condition*, *Leak caused by unused resource*, *Leak caused by complex application lifecycle*, *Leak caused by improper* *allocation and deallocation*, *Leak caused by improper resource liberation in loop* and *Leak caused by Android SDK*.

RQ 1.5 (input artifacts) was organized into 4 categories: *source code* referring to the source code of the mobile application, *APK* which is generated when compiling the source code, *App in Runtime* being the application running on the device and *UI Test* which is a test that simulates the user using the graphical interface of the application.

4.1.4 Data Extraction Strategy

For each selected paper, the necessary data was extracted to answer the research questions proposed for this study. The data extracted are described in Table 2.

Table 2. Data Extraction Form

RQ	Item	Description
RQ 1		
RQ 1.1	Types of contribution	Framework, method, model, tool, evaluation, metric or database.
RQ 1.2	Types of research	Solution proposal, validation research, evaluation research and experience reports.
RQ 1.3	Types of resources	What resources the paper can identify as leaks.
RQ 1.4	Cause of the leak	What causes the resource leaks
RQ 1.5	Input artifact	Apk, source code, app in runtime or ui test

4.1.5 Execution of the Systematic Mapping Study

The search was performed in the selected digital libraries using the string, returning 1760 papers, as shown in the Table 3. After that, duplicate papers were removed and filter 1 (selection based on title, abstract and keywords) was applied. Then, filter 2 (complete paper analysis) was conducted. In the papers selected in filter 2, the information is described in Table 2. The papers remaining after filter 2 were summarized and analyzed.

Table 3. Number of selected papers

Digital Library	Papers Returned	1º Filter	2º Filter
Scopus	427	25	14
IEEE Xplore	337	9	1
Science Direct	34	1	0
ACM DL	217	13	3
Web of Science	258	26	13
Engineering Village	487	48	26
Total	**1760**	**122**	**57**

4.2 Results of the Systematic Mapping

The 57 selected papers were published between 2012 and 2024. In Figure 1, it can be seen that in 2013 (second year with publications) there was an increase in the number of papers, followed by a decline in the years 2014 and 2015. It is also noted that the same number of papers published in 2013 was reached in 2018 and was increased in 2019. It is worth remembering that the search for papers in digital libraries occurred in July 2024, so it is possible that some 2024 papers have not been included.

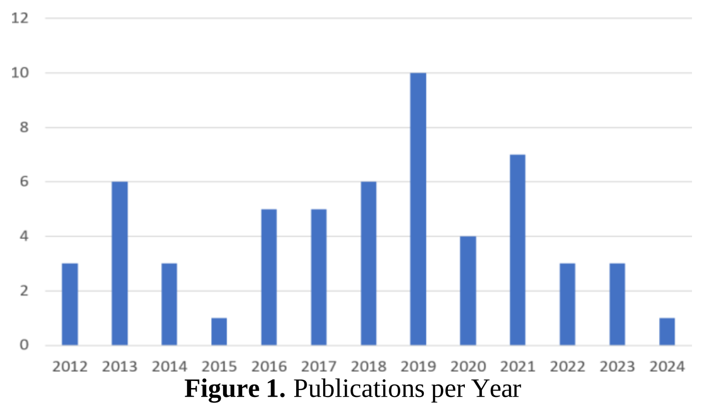


Figure 1. Publications per Year

Figure 2 show the number of papers by type of publication location. It can be seen that the majority of authors (59.6%) published their papers in conferences, 35.1% in journals and, lastly, 5.3% in workshops.
To classify the places of publication, the number of papers published in each place was used as a metric. The ranking of the main sites with at least two papers is shown in Table
4. There are nine publication venues on this list: five journals and six conferences, among which are some of the main means of publication in the area of software engineering. For example, the two conferences with the most articles (three articles, 5.3%) are ICSE and ASE, which are one of the two main events in the area.

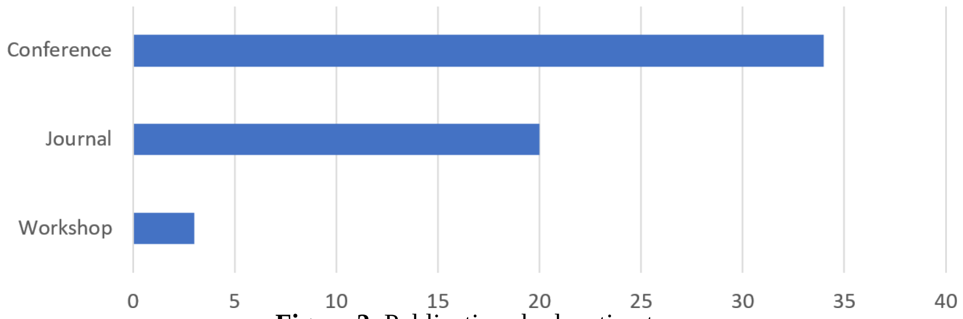


Figure 2. Publications by location type

Table 4. Main places of publication

Publication Venue	**Acronym**	**Qty**
International Conference on Software Engineering	ICSE	3
Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)	LNCS	3
International Conference on Automated Software Engineering	ASE	3
International Computer Software and Applications Conference	COMPSAC	2
ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering	ESEC/FSE	2
IEEE Access	IEEE Access	2
Journal of Computer Science and Technology	JCST	2
International Conference on Mobile Systems, Applications, and Services	MobiSys	2
Software - Practice and Experience	SPE	2
IEEE Transactions on Software Engineering	TSE	2
International Symposium on Software Testing and Analysis	ISSTA	2

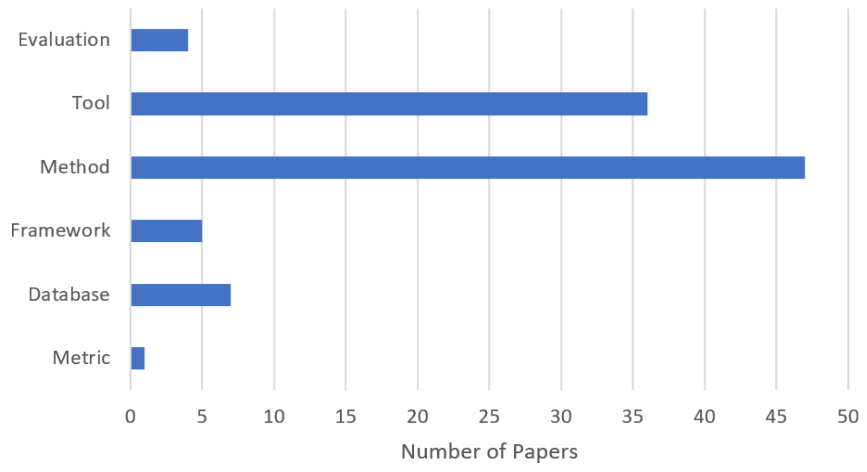
The results obtained from the systematic mapping provide answers for RQ 1: "What is the space for research of the technical literature in identifying resource leak in mobile applications?"

4.2.1 RQ 1.1 - Contribution Types

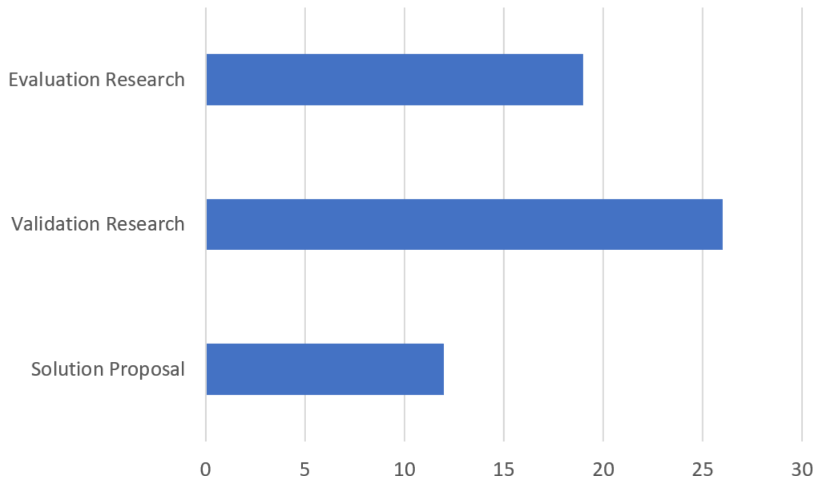
Figure 3 shows the distribution by contribution type for all the 57 papers analyzed in this study. Based on their contributions, some studies were classified into more than one type. For example, [22] presented two contributions: (1) they developed a static analysis technique which encounters inefficiency errors in the use of services, and (2) implemented this technique as a tool called *ServDroid*. Figure 3 also shows that method proposing was the item that attracted more publications, with 47 papers (about 82.5%) focused on this aspect. Then, about 63.2% of the selected papers (36 of 57) proposed new tools.

4.2.2 RQ 1.2 - Research Types

Figure 4 shows the distribution of selected papers by type of research. It can be seen that the papers on the identification of resource leak are mostly related to validation research with 26 papers (45.6%), which indicates the relatively high level of maturity of that community. Next are the studies that focus on evaluation research with 19 papers (33.3%), which suggests that the community has a special attention to empirical approaches, such as the referred species of research. The surveys that describe proposed solutions total 12 papers (21.1%).



Evaluation: [9], [23], [24], [25]
Tool: [26], [17], [27], [11], [28], [29], [30], [31], [32], [16], [33], [18], [34], [22], [35], [36], [15], [37], [38], [10], [19], [20], [39], [40], [41], [42], [43], [44], [45], [8], [46], [47], [48]
Method: [49], [50], [30], [18], [34], [35], [15], [37], [51], [38], [52], [40], [53], [44], [46], [47], [48], [26], [17], [27], [28], [29], [31], [33], [22], [36], [3], [54], [10], [41], [42], [43], [55], [56], [45], [8], [57], [11], [32], [58], [19], [59], [60]
Framework: [16], [20], [39], [61], [62]
Database: [9], [63], [61], [62], [24], [25], [48]
Metric: [27]
Figure 3. Contribution types (see above mentioned references for more information)



Evaluation Research: [26], [9], [64], [27], [11], [28], [30], [31], [16], [33], [18], [34], [22], [35], [36], [15], [19], [20]
Validation Research: [50], [17], [29], [32], [3], [37], [54], [51], [38], [52], [10], [59], [39], [40], [41], [42], [53], [43], [55], [44], [45], [8], [46], [47], [48]
Solution Proposal: [49], [57], [63], [58], [60], [23], [61], [56], [62], [24], [25]
Figure 4. Research types (see the references mentioned above for more information)

4.2.3 RQ 1.3 - Resource Types

During the analysis of the 57 papers, 14 types of resources were identified. These resources are listed in the columns of Table 5, where the types of resources identified for each paper are shown. The paper that worked with the largest number of resource types was [9] with 9 (64.3%) resources, followed by the work of [11] with 8 (57.1%) resource types. The type of resource most worked on in the papers was *Energy*, with 29 (50.9%) papers. Then there is the resource *WakeLock* with 21 (36.8%) papers.

4.2.4 RQ 1.4 – Category of Leak Causes

The extracted leak causes from each study were grouped for the purpose of identifying categories, which were defined in 10, which are: *Complete leak*, *Leak on normal paths*, *Leak on exceptional paths*, *Leak on irregular* *paths*, *Leak caused by race condition*, *Leak caused by unused resource*, *Leak caused by complex application lifecycle*, *Leak caused by improper allocation and deallocation*, *Leak caused by improper resource liberation in loop* and *Leak caused by Android SDK*. Table 6 shows the list of papers that cited or can identify leaks in the mapped categories, where it is possible to see that the most cited category of leak was *Leak caused by complex application lifecycle* with 34 papers (59.6%), followed by the category *Leak on normal paths* with 22 papers (38.6%). Regarding which categories the works could identify the leaks, the work that it is able to find leaks in more categories was [26], being able to find in 7 of the 10 categories. Each of the categories of leak will be explained below.

Code 1. Exemplo de vazamento completo (com defeito)

```
1   static OtpType getType(String email) {
2     Cursor cursor = DATABASE.query(TABLE_NAME, null, EMAIL_COLUMN + "= ?        ",
3         new String[] {email}, null, null, null);
4     if (cursor != null && cursor.getCount() > 0) {
5       cursor.moveToFirst();
6       Integer value = cursor.getInt(cursor.getColumnIndex(TYPE_COLUMN));
7       return OtpType.getEnum(value);
8     }
9     return null;
10  }
```

Complete leak: Developers completely fail to release the resources acquired after their use. For example, in Code 1 a code snippet of the application *Google Authenticator*, where in line 2 an object of type *Cursor* is initialized. In lines 5 and 6, this is used but then it is not released. The code with the correct release of the resource, implemented by one of the application's developers, is shown in Code 2. In this case, the resource usage code was placed in the *try* block and the method call (line 10), which releases the resource, was included in the *finally* clause, so that the even be released both when the code runs without error and when an error occurs.

Code 2. Exemplo de vazamento completo (correto)

```
1   static OtpType getType(String email) {Cursor
2     cursor = getAccount(email);
3     try {
4       if (!cursorIsEmpty(cursor)) {
5         cursor.moveToFirst();
6         Integer value = cursor.getInt(cursor.getColumnIndex(TYPE_COLUMN));
7         return OtpType.getEnum(value);
8       }
9     } finally {
10      tryCloseCursor(cursor);
11    }
12    return null;
13  }
```

Table 5. Resource types

Paper	Concurrency	Connectivity	Database	Files	Localization	Memory	Multimedia	Network	Screen	Security	Sensor	Service	WakeLock	Energy
[9]	x	x	x	x	x		x	x	x				x	x
[11]		x			x	x	x	x	x		x		x	x
[63]		x	x		x	x	x		x				x	x
[35]		x	x	x	x		x	x			x			
[15]		x		x	x		x	x					x	x
[16]		x			x		x				x		x	x
[18]		x			x		x				x		x	x
[47]	x	x			x		x				x		x	
[17]		x			x	x	x				x			x
[42]			x	x			x	x			x			
[8]		x			x		x				x		x	
[26]					x		x				x		x	x
[30]					x		x	x			x			
[41]			x				x						x	x
[53]		x			x								x	x
[25]			x	x			x		x					
[48]			x	x			x	x						
[29]				x					x	x				x
[40]		x			x		x							
[60]	x												x	x
[56]	x												x	x
[49]	x												x	x
[31]		x												x
[32]													x	x
[34]													x	x
[22]												x		x
[3]	x					x								
[37]					x									x
[54]													x	x
[10]	x					x								
[19]													x	x
[20]			x	x										
[39]													x	x
[43]	x					x								
[55]													x	x
[46]													x	x
[57]						x								x
[50]						x								
[27]						x								
[28]												x		
[33]						x								
[36]						x								
[58]						x								
[51]						x								
[38]						x								
[52]						x								
[59]						x								
[23]														x
[61]														x
[62]														x
[44]						x								
[45]											x			
[24]														x
[65]	x	x	x	x			x	x						
[66]				x		x		x						
[67]						x								
[68]						x		x						

Table 6. List of papers that cited or can identify leaks in categories

Cited	
Category Title	Papers
Complete leak	[26], [49], [57], [9], [63], [17], [11], [32], [18], [22], [37], [41], [53], [45], [8], [47], [65], [68]
Leak on normal paths	[26], [49], [9], [17], [11], [29], [32], [16], [18], [34], [22] [15], [54], [60], [41], [53], [55], [56], [45], [8], [47], [65],
Leak on exceptional paths	[26], [49], [9], [17], [32], [16], [54], [19], [25], [48], [65], [66]
Leak on irregular paths	[17], [11], [18], [15], [47], [68]
Leak caused by race condition	[26], [49], [32], [34], [60], [43], [56], [65], [66]
Leak caused by unused resource	[26], [31], [22], [15]
Leak caused by complex application lifecycle	[26], [57], [9], [50], [17], [11], [28], [30], [32], [16], [33], [18], [34] [22], [35], [36], [15], [58], [3], [37], [51], [38], [10], [19], [20], [40], [41], [42], [43], [44], [8], [46], [47], [65],
Leak caused by improper allocation and deallocation	[26], [57], [9], [50], [31], [22], [58], [3], [38], [10], [47], [66], [67]
Leak caused by improper resource liberation in loop	[29], [16], [65], [68]
Leak caused by Android SDK	[28]

Identified			
Category Title	Static	Dynamic	Static and Dynamic
Complete leak	[26], [49], [17], [11], [30], [18], [34], [22], [35], [36], [15], [37], [54], [38], [40], [60], [41], [42], [55], [45], [8], [46], [48], [65]	[57], [50], [27], [31], [32], [33], [51], [52], [10], [19], [59], [39], [61], [43], [56], [44], [47], [24]	[28], [29], [16], [3], [20], [53], [62]
Leak on normal paths	[26], [49], [17], [11], [18], [34], [22], [15], [54], [60], [41], [42], [55], [45], [8], [65], [68],	[27], [32], [56], [47],	[28], [29], [16], [53]
Leak on exceptional paths	[26], [17], [54], [48], [65], [66]	[32]	[16]
Leak on irregular paths	[17], [11], [18], [15], [68]	-	-
Leak caused by race condition	[26], [49], [34], [60], [65], [66]	[32], [43], [56]	-
Leak caused by unused resource	[26], [22], [15]	[31]	-
Leak caused by complex application lifecycle	[26], [17], [11], [30], [18], [34], [22], [35], [36], [15], [37], [38], [40], [41], [42], [8], [46],	[57], [50], [32], [33], [51], [10], [19], [43], [44], [47],	[28], [16], [3], [20]
Leak caused by improper allocation and deallocation	[26], [22], [38], [66]	[57], [50], [31], [10]	[3], [67]
Leak caused by improper resource liberation in loop	[65], [68]	-	[29], [16]
Leak caused by Android SDK	-	-	[28]

Code 3. Exemplo de vazamento em caminhos normais (com defeito)

```
private void realRun () {
    synchronized (createLock) {
        if (mTask != this) {
            Log.w( THIS_FILE , " unexpected task : " + mNetworkType + (
                mConnected ? " CONNECTED" : "DISCONNECTED"));
            return;
        }
        mTask = null;
        Log.d(THIS_FILE, " deliver change for " + mNetworkType + (mConnected ? "
            CONNECTED" : "DISCONNECTED"));
        // onConnectivityChanged(mNetworkType, mConnected);
        dataConnectionChanged(mNetworkType, true); sipWakeLock.r
        elease(this);
    }
}
```

Leak on normal paths: Resources are released only in a few paths in the code. For example, Code 3 shows the code snippet extracted from the application *CSipSimple*[7], in which the resource in the *sipWakeLock* object is released in line 11. However, if the condition in line 3 is true, the resource will not be released, as there is a *return* statement that ends the execution of the current function. Code 4 shows the code correction implemented by the author of the application[8], where the resource release statement is implemented within *if* on line 5, while the other release statement remains on line 12.

Code 4. Exemplo de vazamento em caminhos normais (correto)

```
1   private  void  realRun ()  {
2           synchronized  (createLock)  {
3               if  (mTask  !=  this)  {
4                   Log.w( THIS_FILE ,  "    unexpected  task :  " + mNetworkType + (
                        mConnected  ?  " CONNECTED"  :  "DISCONNECTED"));
5                   sipWakeLock . release ( this ) ;
6                   return ;
7               }
8               mTask  =  null ;
9               Log.d(THIS_FILE ,  "  deliver  change  for  "  +  mNetworkType  +  (mConnected  ?  "
                    CONNECTED"  :  "DISCONNECTED"));
10              //  onConnectivityChanged (mNetworkType ,  mConnected );
11              dataConnectionChanged (mNetworkType ,  true );  sipWakeLock . r
12              elease(this);
13          }
14  }
```

[7] https://github.com/r3gis3r/CSipSimple/blob/88d62bc9510809ab6d9750b0a0b761ef08c85948/src/com/csipsimple/service/SipService.java

[8] https://github.com/r3gis3r/CSipSimple/blob/686620ba55f584c16388f4c1052a336ee6896bcb/src/com/csipsimple/service/SipService.java

Code 5. Exemplo de vazamento em caminhos excepcionais (com defeito)

```
@Override
protected  Boolean  doInBackground(Void . . .  params)  {
    try  {
        mImportResults  =  StorageImporter . importSettings (mContext ,  mInputStream
            ,
                mEncryptionKey ,  mIncludeGlobals ,  mAccountUuids ,  mOverwrite ) ;
    }  catch  (StorageImportExportException  e)  {
        Log.w(K9.LOG_TAG,  "Exception  during  export",  e);
        return  false ;
    }
    return  true ;
}
```

Leak on exceptional paths: System resources are not released if exceptions occur. For example, in the code snippet extracted from the application *k-9 mail*[9], shown in Code 5, there is no release of *InputStream* in case of error. The correct version of the application is implemented in the code shown in Code 6[10], where the use of the resource was placed in the block *try* and the release of the resource was included in the *finally* clause, thus ensuring the release of the resource, whether an error occurs or not.

Code 6. Exemplo de vazamento em caminhos excepcionais (correto)

```
@Override
protected Boolean doInBackground(Void... params) {
    try {
        InputStream is = mContext.getContentResolver().openInputStream(mUri);
        try {
            mImportResults = StorageImporter.importSettings(mContext, is,
                    mEncryptionKey, mIncludeGlobals, mAccountUuids, mOverwrite
                    );
        } finally {
            try {
                is.close();
            } catch (IOException e) { /* Ignore */ }
        }
    } catch (StorageImportExportException e) {
        Log.w(K9.LOG_TAG, "Exception during import", e);
        return false;
    } catch (FileNotFoundException e) {
        Log.w(K9.LOG_TAG, "Couldn't open import file", e);
        return false;
    }
    return true;
}
```

[9] https://github.com/k9mail/k-9/blob/dfa97cd878dceb6821a5e98c722d5bf3c5c5a02d/src/com/fsck/k9/activity/Accounts.java

[10] https://github.com/k9mail/k-9/blob/644571cfe512a21e84764a4adcb34cf40da9a335/src/com/fsck/k9/activity/Accounts.java

Code 7. Exemplo de vazamento em caminhos irregulares (com defeito)

```
1   builder.setNegativeButton(R.string.cancel_action,
2       new DialogInterface.OnClickListener() {
3           @Override
4           public void onClick(DialogInterface dialog, int which) {dialog.
5               dismiss();
6               try {
7                   mInputStream.close();
8               } catch (Exception e) { /* Ignore */ }
9           }
10      });
11  builder.show();
```

Leak on irregular paths: The resources are released in an inappropriate location, that is, the application has a release operation performed only when a specific event occurs, such as *onClick*, *onKeyDown*, and so on. If the user does not trigger any of these events, the related resources cannot be released. For example, in Code 7 a code snippet from the application *k-9 mail*[11] is sampled, where it is noted that the release of the resource of an *InputStream* type (line 7) occurs in the event *onClick* of the dialog's "cancel" button. If the user leaves the application and does not click this button, the resource will not be released. The code with the error correction[12], done by one of the developers of the application, it is shown in Code 8, where the resource release of the event *onClick* was removed to a suitable location. This situation can occur with different types of resources.

Code 8. Exemplo de vazamento em caminhos irregulares (correto)

```
builder.setNegativeButton(R.string.cancel_action,
        new DialogInterface.OnClickListener() {
            @Override
            public void onClick(DialogInterface dialog, int which) {
                dialog.dismiss();
            }
        });
builder.show();
```

11 https://github.com/k9mail/k-9/blob/dfa97cd878dceb6821a5e98c722d5bf3c5c5a02d/src/com/fsck/k9/activity/Accounts.java

12 https://github.com/k9mail/k-9/blob/644571cfe512a21e84764a4adcb34cf40da9a335/src/com/fsck/k9/activity/Accounts.java

Code 9. Exemplo de vazamento causado por condição de corrida

```
    ...
    public void stop() {
        synchronized (createLock) {
            if (mTask != null) {
                Log.d(THIS_FILE, "Delete already pushed task in stack");
                mTask.cancel();
                sipWakeLock.release(mTask);
            }

            if(mTimer != null) { mTimer
                .purge(); mTimer.cancel
                ();
            }
            mTimer = null;
        }
    }
    ...
    protected void onChanged(String type, boolean connected) {
        boolean fireChanges = false;
        synchronized (createLock) {
            // When turning on WIFI, it needs some time for network
            // connectivity to get stabile so we defer good news (because
            // we want to skip the interim ones) but deliver bad news
            // immediately
            if (connected) {
                Log.d(THIS_FILE, "Push a task to connected timer");
                if (mTask != null) {
                    Log.d(THIS_FILE, "We already have a current task in stack"); mTask.
                    cancel();
                    sipWakeLock.release(mTask);
                }
                mTask = new MyTimerTask(type, connected);
                if(mTimer == null) {
```

```
34                    mTimer = new Timer("Connected-timer");
35                }
36                mTimer.schedule(mTask, 2 * 1000L);
37                // hold wakup lock so that we can finish changes before the
38                // device goes to sleep
39                sipWakeLock.acquire(mTask);
40            } else {
41                if ((mTask != null) && mTask.mNetworkType.equals(type)) { mTask.
42                    cancel();
43                    sipWakeLock.release(mTask);
44                }
45                fireChanges = true;
46            }
47        }
48        if(fireChanges) {
49            Log.d(THIS_FILE, "Fire changes right now cause it's a deconnect info")
              ;
50            dataConnectionChanged(type, false);
51        }
52    }
```

Leak caused by race condition: It occurs when the management of a resource can be performed (that is, acquired and released) by different *threads* in the application. In the common case, one *thread* activates the resource and, some time later, another *thread* disables the resource, resulting in the normal behavior of using the component. However, the order in which threads attempt to access the shared resource cannot be expected due to several factors. For example, in the code snippet extracted from the application *CSipSimple*[13], shown in Code 9, it is possible to observe that the acquisition, use and release of some resources are modified by different *threads* (lines 3 and 20). For better management, the

[13] https://github.com/r3gis3r/CSipSimple/blob/88d62bc9510809ab6d9750b0a0b761ef08c85948/src/com/csipsimple/service/SipService.java#L646

keyword *syncronized* was used, which blocks the use of a block of code at any given time, that is, if a *thread* is executing its restricted code block, the too many *trheads* will not be able to execute their code blocks until this *thread* ends.

Code 10. Exemplo de vazamento causado por recurso não utilizado (com defeito)

```
1   private void checkOutgoing() throws MessagingException {
2       if (!(account.getRemoteStore() instanceof WebDavStore)) {
3           publishProgress(R.string.
                account_setup_check_settings_check_outgoing_msg);
4       }
5       Transport transport = Transport.getInstance(K9.app, account); tran
6       sport.close();
7       transport.open();
8       transport.close();
9   }
```

Leak caused by unused resource: The resource is acquired, but not used. As an example, we have the code snippet extracted from the application *k-9 mail*[14], shown in Code 10. Note that in line 6 the resource is released, however, the developer acquires it again in line 7 and then releases it again in line 8, without having been used in the last two operations. The code with the correction suggested by the authors of the paper is shown in Code 11, where the acquisition and release that were not necessary were removed.

Code 11. Exemplo de vazamento causado por recurso não utilizado (correto)

```
private  void  checkOutgoing ()  throws  MessagingException  {
    if  (!(account . getRemote Store ()  instanceof  WebDavStore))  { publish
        Progress (R. string .
            account_setup_check_settings_check_outgoing_msg );
    }
    Transport  transport  =  Transport . getInstance (K9 . app ,  account );
    transport . close () ;
}
```

Code 12. Exemplo de vazamento causado pela complexidade do ciclo de vida da aplicação (com defeito)

```
@Override
protected  void  onPause ()  {
    super . onPause () ;
    if  (mChat Controller  !=  null )  { m
        Chat Controller . onPause () ;
    }

    stop Watching External Storage () ;

}
```

Leak caused by complex application lifecycle: Poor management of resources during the life cycle of a component, whether due to carelessness or ignorance of how the Android component life cycle work. An example of this problem can be seen in the code snippet of the application *Surespot* shown in Code 12[15], in which the instruction to pause the audio capture in the *onPause* method of the life cycle was not implemented. The correction of

[14] https://github.com/k9mail/k-9/blob/acd18291f2ded8f176beb074415f8c42ca829966/k9mail/src/main/java/com/fsck/k9/activity/setup/AccountSetupCheckSettings.java

[15] https://f-droid.org/forums/topic/surespot-complete-whatsapp-alternative-application/

this error made by one of the authors of the application is shown in Code 13[16]. In line 8 there is an instruction that calls the method that pauses the audio capture. One way to reproduce the leak is to alternate between displaying that application and another one, while in different component states (e.g. navigate to the home screen and return to your application).

Code 13. Exemplo de vazamento causado pela complexidade do ciclo de vida da aplicação (correto)

```
1   @Override
2   protected  void  onPause ()  {
3       super . onPause () ;
4       if  (mChat Controller  !=  null )  { m
5           Chat Controller . onPause () ;
6       }
7
8       Voice Controller . pause () ;
9
10      stop Watching External Storage () ;
11
12  }
```

Code 14. Exemplo de vazamento causado por alocação e desalocação imprópria (com defeito)

```
private  void  stopMovement ()  {
    listView . onTouchEvent (MotionEvent . obtain (SystemClock . uptime Millis () ,
            SystemClock . uptime Millis () ,  MotionEvent .ACTION_CANCEL,  0 ,  0 ,  0)) ;
}
```

Leak caused by improper allocation and deallocation: Inadequate allocation and deallocation of space in native memory, misuse of API (*Application Programming Interface*), lack or loss of references to resource objects can cause an unexpected increase in memory used. For example, in the method *stopMovement*[17] (Code 14) of the *Xabber* application, the *MotionEvent* resource is allocated and used. However, it is not released, which causes *View* to remains active and not released. The code with the correction implemented by one of the developers of the application is shown in Code 15[18], where the resource allocation is assigned to a variable (lines 2 and 3), then it is used (line 4) and then on line 5 the *recycle* method is called (allowing the object to be destroyed).

Code 15. Exemplo de vazamento causado por alocação e desalocação imprópria (correto)

```
1   private  void  stopMovement()  {
2       MotionEvent  event  =  MotionEvent.obtain(SystemClock.uptimeMillis(),
3               SystemClock.uptimeMillis(),  MotionEvent.ACTION_CANCEL,  0,  0,  0);
4       listView.onTouchEvent(event);
5       event.recycle();
6   }
```

[16] https://f-droid.org/forums/topic/surespot-complete-whatsapp-alternative-application/

[17] https://github.com/redsolution/xabber-android/blob/926f4ca93ecaebf2c614dd65a8aab39de8757218/src/com/xabber/android/ui/ContactListFragment.java

[18] https://github.com/redsolution/xabber-android/blob/4cf5f2db9a67189b8990899651f1618bd1b0fd1e/src/com/xabber/android/ui/ContactListFragment.java

Code 16. Exemplo de vazamento causado por liberação inadequada de recursos em loop (com defeito)

```
private  boolean  resumeDownload  ( )  {
    BufferedInputStream  in  = null ; File
    OutputStream  fos  =  null ;
    BufferedOutputStream  bout  =  null ;

    try  {
        for  ( ;  downloadIndex  <  fileNames.length ;  downloadIndex++)  {
            . . .
            in  =  new  BufferedInputStream(connection.getInputStream()) ;  fos
            =  (downloaded  ==  0)  ?  new  FileOutputStream(file
                    .getAbsolutePath())  :  new  FileOutputStream(file
                    .getAbsolutePath() ,  true) ;

            bout = new BufferedOutputStream(fos , DOWNLOAD_BUFFER_SIZE) ;
            byte[]  data  =  new  byte[DOWNLOAD_BUFFER_SIZE] ;
            int  x  =  0;

            while  (isRunning  &&  (x  =  in.read(data ,  0,  DOWNLOAD_BUFFER_SIZE))
                >=  0)  {
                bout.write(data ,  0,  x);  downloaded
                +=  x ;
                double  percent  =  100.0  *  ((1.0  *  downloaded)  /  (1.0  *  total)) ;
                updateProgress((int)  percent ,  fileNames.length ,  downloadIndex)
            }        ;
            . . .
        }
    }  catch  (FileNotFoundException  e)  {
        Log.e("quran_srv" ,  "File  not  found:  IO  Exception" ,  e) ;
    }  catch  (IOException  e)  {
        Log.e("quran_srv" ,  "Download  paused:  IO  Exception" ,  e) ;
        return  false ;
        } catch  (Exception  e)  {
            Log.e("quran_srv",  "Download  paused:  Exception",  e);
            return  false;
        }

        return  true;
    }
```

Leak caused by improper resource liberation in loop: A resource is acquired repeat- edly within a loop, but it is not released enough times before exiting the application. For example, the *resumeDownload*[19] (Code 16) of the application *Quran for Android* has the allo- cation of three resources (*BufferedInputStream*, *FileOutputStream* and *BufferedOutputStream*) within a repetition and these resources are not released within the loop. The code with the correction made by one of the developers is shown in Code 17[20], where lines 24 to 28 are used to release resources.

[19] https://github.com/quran/quran_android/blob/121cd5803b610eeb79bd42c78ece53df7043cc16/src/com/quran/labs/androidquran/service/QuranDataService.java

[20] https://github.com/quran/quran_android/blob/4e4a60249a27cbf834d59db4611eb9f1843240c0/src/com/quran/labs/androidquran/service/QuranDataService.java

Code 17. Exemplo de vazamento causado por liberação inadequada de recursos em loop (correto)

```
1      private  boolean  resumeDownload  ( )  {
2          Buffered Input Stream  in  = null ; File
3          Output Stream  fos  = null ;
4          BufferedOutputStream  bout  =  null ;
5
6          try  {
7              for  (;  downloadIndex  <  fileNames.length ;  downloadIndex++)  {
8                  ...
9                  in  =  new  BufferedInputStream(connection.getInputStream(),
                       DOWNLOAD_BUFFER_SIZE);
10                 fos  =  (downloaded  ==  0)  ?  new  FileOutputStream(file
11                             .getAbsolutePath())  :  new  FileOutputStream(file
12                             .getAbsolutePath(),  true);
13
14                 bout = new  BufferedOutputStream(fos,  DOWNLOAD_BUFFER_SIZE);
15                 byte[]  data  = new  byte[DOWNLOAD_BUFFER_SIZE];
16                 int  x  =  0;
17
18                 while  (isRunning  &&  (x  =  in.read(data,  0,  DOWNLOAD_BUFFER_SIZE))
                       >=  0)  {
19                     bout.write(data,  0,  x);  downloaded
20                     +=  x;
21                     double  percent  =  100.0  *  ((1.0  *  downloaded)  /  (1.0  *  total));
22                     updateProgress((int)  percent,  fileNames.length,  downloadIndex)
                           ;
23                 }
24                 bout.flush();
25                 bout.close();
26                 fos.close();
27                 ...
28             `}
29         }  catch  (FileNotFoundException  e)  {
30             Log.e("quran_srv",  "File  not  found:  IO  Exception",  e);
31         }  catch  (IOException  e)  {
32             Log.e("quran_srv",  "Download  paused:  IO  Exception",  e);
33             return  false;
34         }  catch  (Exception  e)  {
35             Log.e("quran_srv",  "Download  paused:  Exception",  e);
36             return  false;
37         }
38
39         return  true;
40     }
```

Code 18. Exemplo de vazamento causado pelo Android SDK (com defeito)

```
import android . os . Bundle ;
import android . view . View ;
import android . view . ViewGroup ;
import android . view . inputmethod . InputMethodManager ;
import android . widget . Button ;
import android . widget . EditText ;
import android . widget . LinearLayout ;
import android . widget . TextView ;
import java . lang . reflect . Field ;
import static android . view . ViewGroup . LayoutParams .MATCH_PARENT;
import static android . view . ViewGroup . LayoutParams .WRAP_CONTENT;
import static android . widget . LinearLayout . VERTICAL ;
public class MainActivity extends Activity {
  private TextView immView ;

  @Override protected void onCreate (Bundle savedInstanceState) {
    super . onCreate ( savedInstanceState ) ;
    final LinearLayout layout = new LinearLayout ( this ) ;
    layout . setOrientation (VERTICAL) ;
    immView = new TextView ( this ) ;
    final EditText leaking = new EditText ( this ) ;
    Button button = new Button ( this ) ;
    button . setText ( "Remove EditText " ) ;
    button . setOnClickListener (new View . OnClickListener () {
      @Override public void onClick (View v) {
        layout . removeView ( leaking ) ;
        layout . removeView (v) ;
      }
    }) ;
    layout . addView (immView, new ViewGroup . LayoutParams (MATCH_PARENT,
        WRAP_CONTENT)) ;
    layout . addView ( button , new ViewGroup . LayoutParams (MATCH_PARENT,
        WRAP_CONTENT)) ;
    layout . addView ( leaking , new ViewGroup . LayoutParams (MATCH_PARENT,
        WRAP_CONTENT)) ;
    setContentView ( layout ) ;
    updateImmView () ;
  }

  private void updateImmView () {
    logServedView () ;
    immView . postDelayed (new Runnable () {
      @Override public void run () {
        updateImmView () ;
      }
    } , 100) ;
  }

  private void logServedView () {
    try {
      Field sInstanceField = InputMethodManager . class . getDeclaredField (" sInst
          ance") ;
      sInstanceField . setAccessible ( true ) ;
      Object imm = sInstanceField . get ( null ) ;
      if (imm == null) {
        // Not set yet .
        return ;
```

```
            }
            Field mServedViewField = InputMethodManager.class.getDeclaredField("
                mServedView");
            mServedViewField.setAccessible(true);
            View servedView = (View) mServedViewField.get(imm);
            immView.setText("InputMethodManager.mServedView: "
                + servedView.getClass().getName()
                + "\nAttached:"
                + servedView.isAttachedToWindow());
          } catch (NoSuchFieldException e) {
            throw new RuntimeException(e);
          } catch (IllegalAccessException e) {
            throw new RuntimeException(e);
          }
        }
      }
```

Leak caused by Android SDK: Leaks can be caused by the Android SDK and the developer has few ways to fix them. Some known leak patterns are defined in AndroidExcludedRefs.java[21] provided by LeakCanary[22], such as the leak of the last focused *View* that happens in the API 22 from Android. The steps to reproduce this error are: (1) set up a hierarchy of *View* with only one focusable; (2) request focus on the focusable *View*; (3) remove this *View* (or one of its ancestors) from the hierarchy of *View*. A possible code to reproduce this error is shown in Code 18[23], where *TextView* (lines 39 to 44) displays the reference in *InputMethodManager.mServedView*, updated every 100 milliseconds. By clicking on the button declared on lines 23 to 30, the *EditText* is removed. After that, note that it is still referenced by *InputMethodManager.mServedView*, although it is detached. A developer found a way to prevent this leak, which was implemented in the form of a method called *fixInputMethod* (shown in Code 19) which must be called in the cycle *onDestroy* method of life.

Code 19. Exemplo de vazamento causado pelo Android SDK (correto)

```
    public static void fixInputMethod(Context context) {
        if (context == null) return;
        InputMethodManager inputMethodManager = null;
        try {
            inputMethodManager = (InputMethodManager) context.get
                ApplicationContext()
                  .getSystemService(Context.INPUT_METHOD_SERVICE);
        } catch (Throwable th) {
            th.printStackTrace();
        }
        if (inputMethodManager == null) return;
        String[] strArr = new String[]{"mCurRootView", "mServedView", "
            mNextServedView"};
        for (int i = 0; i < 3; i++) {
            try {
                Field declaredField = inputMethodManager.getClass().get
                    DeclaredField(strArr[i]);
                if (declaredField == null) continue;
                if (!declaredField.isAccessible()) {
                    declaredField.setAccessible(true);
                }
                Object obj = declaredField.get(inputMethodManager);
                if (obj == null || !(obj instanceof View)) continue;
                View view = (View) obj;
                if (view.getContext() == context) {
                    declaredField.set(inputMethodManager, null);
                } else {
                    return;
                }
            } catch (Throwable th) {
                th.printStackTrace();
            }
        }
    }
```

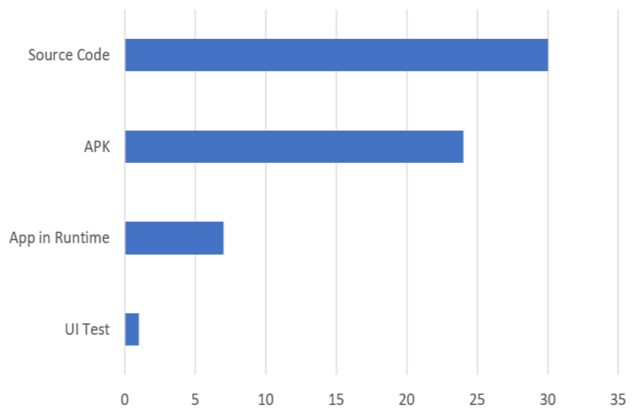
4.2.5. RQ 1.5 – Input Artifact

The distribution of the input artifacts of the selected papers is shown in Figure 5. It can be seen that the source code artifact is accepted in thirty primary studies (52.6%), followed by the APK artifact accepted in twenty-four studies (42.1%). In turn, only the papers [9], [53], [24] and [37] accept both source code and APK as input artifact. Seven (12.3%) studies receive the running application as input artifact. One (1.8%) of the study requires the application's source code and user interface (UI) tests as input.

[21] https://github.com/hehonghui/leakcanary-for-eclipse/blob/master/Leakcanary-lib/src/com/squareup/leakcanary/AndroidExcludedRefs.java

[22] https://square.github.io/leakcanary/

[23] https://issuetracker.google.com/issues/37043700#comment1



Source Code: [49], [9], [63], [50], [30], [32], [16], [35], [58], [3], [37], [54], [38], [10], [59], [40], [60], [41], [53], [61], [56], [62], [47], [24], [25], [48], [65], [66], [67], [68]
APK: [26], [64], [17], [11], [28], [33], [18], [34], [22], [36], [15], [51], [9], [37], [42], [23], [53], [43], [55], [44], [45], [8], [46], [24]
App in Runtime: [57], [27], [29], [31], [52], [19], [39]
UI Test: [32]
Figure 5. Input artifact

4.3 Discussions

The main purpose of this systematic mapping was to identify categories of leak causes in mobile applications. Currently, to the best of our knowledge, there is no systematic mapping with this objective in this important and current area. As a result, a systematic mapping on the area can contribute to software developers by helping them understand the categories of leak causes and consequently avoid leaks in their applications. Furthermore, it can assist researchers focus their efforts on categories that have not yet been explored much; for example, the *Leak caused by Android SDK* category has only one method that can identify leaks and the *Leak caused by improper resource release in loop* has only four.

Regarding the most critical categories, we can highlight two, *Complete leak* and *Leak caused by complex application lifecycle*. The *Complete leak* category, which is cited by 18 studies (31.6%), is the most critical, as the resource is acquired and never released, which means the application will probably present some inconsistency due to this leak. The *Leak caused by complex application lifecycle*, which is cited by 34 studies (59.6%), is the second most critical category, as many resources need to be acquired and released in the lifecycle callbacks, as these callbacks are called several times during the use of the application and if the developer has not correctly implemented acquisition and release in these callbacks, the probability of an application failure is high.

Regarding the problems that can occur when there is a resource leak, we can mention the following:

- **Huge battery drain** according to Naseer *et al.* [53] and Banerjee *et al.* [69] can be categorized into energy-bugs and energy-hotspots. An application is energy inefficient due to an energy-bug when it prevents the device from being idle even after its execution is complete and there is no user activity. An application is energy inefficient due to energy-hotspot access in a scenario where the application is running on a device and starts consuming a large amount of battery power;
- **Degradation of usability and responsiveness** can happen when a resource becomes partially

or completely unavailable to the user, preventing the use of a certain func- tionality of the application and/or mobile device [8];

- **Huge memory consumption** is when the application allocates resources in the de- vice's memory and does not release them, which can cause a memory overflow, and consequently failure to use the mobile device [44];
- **Performance degradation** through the gradual depletion of the finite computational resources of the mobile device at runtime, causing slowdowns when using the device and the mobile application [9];
- **Application crash** can occur when the application terminates unexpectedly due to the use of a mismanaged resource [43];
- **Problem in another application** can occur when a resource that can only be used in one application at a time is not released and another application tries to access it. For example, an application allocates the camera and when exiting the application the resource has not been released, so another application tries to use it, but cannot use the camera [47].

4.4 Threats to Validity

In this section, the main threats to the validity of systematic mapping are discussed, organized into four categories: construct, internal, external, and conclusion.

Construct validity: a well-defined search string is important for returning consistent results. To minimize this threat, a control group of 8 papers related to identifying resource leaks in mobile applications was used. To improve the search string, execution cycles of the search string were performed for refinements in Scopus and we extracted synonyms of previously identified keywords.

Internal validity: during the execution of the study, some subjective decisions may be made, such as the selection of primary studies and data extraction. To minimize this threat, the inclusion and exclusion criteria informed in the planning were followed. As well as the systematic mapping protocol, it was reviewed by researchers with extensive experience in Experimental Software Engineering and planning/execution of systematic mappings.

External validity: it is important that the study can be reproduced. This threat was minimized, due to the systematic procedure followed during the mapping study, therefore, it is believed that this study can be repeated.

Conclusion validity: the results found need to be related to the data extracted from the primary studies. To lessen this threat, we included papers references in each of the analyzes that were performed.

4.5 Conclusions of Systematic Mapping

This section presented the study of the systematic mapping of scientific publications on the identification of resource leaks in mobile applications. In the study, a total of 57 papers published between 2012 and 2024 were included. From the analysis of the papers it was possible to identify 10 categories of leak causes, which can help developers avoid these problems in their applications. The results also showed 14 types of resources in which the papers acted, with the types of resources *Energy* (50.9%) and *WakeLock* (36.8%) were the two most explored in the papers. In the next section, a study will be presented that will analyze the categories of causes of leaks identified in the systematic mapping.

V. Survey with Experts

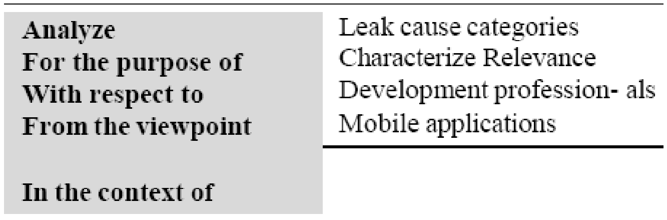
In the systematic mapping study previously presented, we extracted 10 categories of causes of leaks in mobile applications. In order to validate these categories, an opinion survey was planned whose details are presented below.

5.1 Methods

5.1.1 Goal

The objective of this study is outlined from the GQM paradigm (Goal, Question, and Metric) [70], which is presented in Table 7. The execution of this opinion poll had as main objective to characterize the relevance of the categories of causes of leaks identified in the systematic mapping.

Table 7. Objective of the survey according to the GQM paradigm



Analyze	Leak cause categories
For the purpose of	Characterize Relevance
With respect to	Development profession- als
From the viewpoint	Mobile applications
In the context of	

Leak cause categories Characterize Relevance
Development profession- als

Mobile applications

5.1.2 Research Question
Based on the objective, the following research question was formulated:
What is the relevance of each of the categories to represent causes of resources leak for mobile applications?
Metrics: Likert scale value attributed by the participant and the participant's weight.
5.1.3 Instrumentation
Questions were proposed to the participants in order to identify their profile with regard to their experience with resource leaks in mobile applications, in addition to the Free and Informed Consent Term - FICT. In addition, a questionnaire was prepared to assess the relevance of the categories of causes of leaks in mobile applications according to the research question mentioned in the previous subsection.
The Likert ordinal scale will be used, offering the options: (0) No relevance, (1) Very low relevance, (2) Low relevance, (3) Medium relevance, (4) High relevance and (5) Very high relevance. In addition, there are the following open questions: "What is your opinion about the categories presented? Are there any other categories of causes of leaks? ". The evaluation questionnaire is available at: https://drive.google.com/drive/folders/12zC6TUQhtm-KU1DGsnfuYvLyzAETfrln.

5.1.4 Procedure for Analysis of Relevance
To define the level of relevance of each category, the participant's years of experience and the mobile platforms on which he works or worked were based. According to [71] it is necessary to differentiate the responses of the participants by associating a weight to each of them, considering, for example, years of experience. The weight of each participant for this study is calculated according to Equation 1.

$$Weight(i) = NPW(i) + \frac{YE(i)}{Median(YEAP)} \tag{1}$$

Where, *Weight(i)* is the weight assigned to the participant *i*; *YE* are the years of experience of the participant *i*; *YEAP* are years of experience for all participants; and *NPW(i)* is the number of platforms worked on by the participant *i*.

Finally, the level of relevance is being calculated as a value between 0% and 100% by normalizing the value obtained by each category, as shown in Equation 2.

$$Relevance(c) = \frac{\sum_{i=1}^{N} (Likert(i, c) * Weight(i))}{\sum_{i=1}^{N} (Weight(i) * 5)} \tag{2}$$

Where, *Relevance(c)* is the level of relevance for the category *c*; *Likert(i, c)* is the relevance value, according to the Likert scale (between 0 and 5), defined by the participant *i* for the *c* category; *Weight(i)* is the weight assigned to the participant *i*; *N* is the total number of participants who answered the survey; and the constant *5* is the maximum value for the relevance level scale.

After this step, the values will be sorted in descending order. The most relevant categories will be those whose variable *Relevance(c)* has the highest values.

5.1.5 Study Execution
To participate in this study, developers with experience in resource leakages in mobile applications were invited. The developer base was obtained from a search on the social network Linkedin[24], as well as in development communities on Facebook[25]. To participate in the study, the professionals had to express interest, agreeing with a Free and Informed Consent Term and, after that, answered the proposed questionnaire.
The questionnaire was active for a period of 2 months and 12 days. The form, which was sent to 147 developers, received responses from 22 of them, reaching an 81% confidence level in the number of participants, according to the formula of [72] and a response rate of 15%.

Table 8. Opinion Survey participants

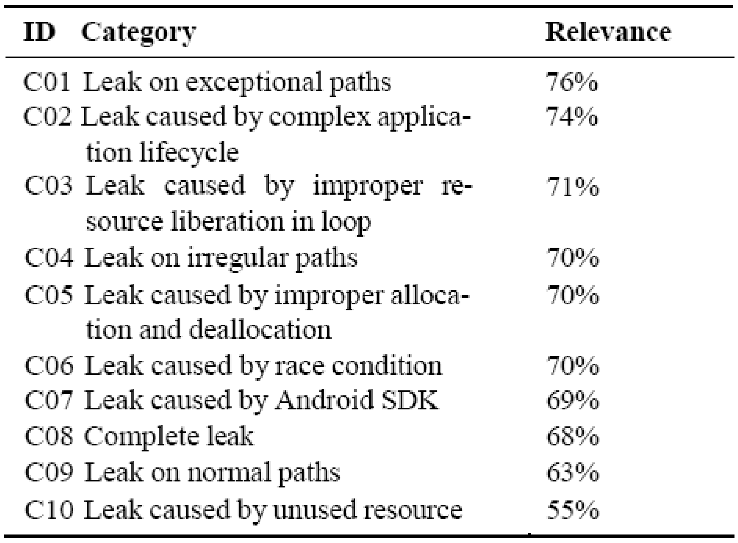
ID	Platforms	Years of Experience	Country	Weight
P01	Android	4	Pakistan	2
P02	Android, iOS	5	India	3,25
P03	Android	6	India	2,5
P04	Android, iOS	5	Brazil	3,25
P05	Android, iOS	2	Brazil	2,5
P06	Windows	7	Brazil	2,75
P07	Android, iOS	6	Brazil	3,5
P08	Android	5	Pakistan	2,25
P09	Android	2	Brazil	1,5
P10	Android, iOS	2	Brazil	2,5
P11	Android	3	Brazil	1,75
P12	Android	3	Brazil	1,75
P13	Android	3	Brazil	1,75
P14	Android, iOS	8	Brazil	4
P15	Android	4	Brazil	2
P16	Android, Windows	10	Brazil	4,5
P17	Android, iOS	5	Brazil	3,25
P18	Android, iOS	1	Brazil	2,25
P19	Android, iOS	4	Brazil	3
P20	Android	1	Brazil	1,25
P21	Android	5	Brazil	2,25
P22	Android, iOS	2	Brazil	2,5

24 http://linkedin.com

25 https://www.facebook.com

Due to privacy concerns, the personal data of the participants will not be presented. Table 8 presents the characterization information of the developers, including the weight already calculated for each one. It can be seen that they work or have worked on at least one of the following platforms: Android, iOS or Windows. Participants also live in different countries like Brazil, India or Pakistan. The median time of experience with resource leaks in mobile applications from the developers in this study is 4 years.

Table 9. Relevance of leak categories *Results Analysis*



ID	Category	Relevance
C01	Leak on exceptional paths	76%
C02	Leak caused by complex application lifecycle	74%
C03	Leak caused by improper resource liberation in loop	71%
C04	Leak on irregular paths	70%
C05	Leak caused by improper allocation and deallocation	70%
C06	Leak caused by race condition	70%
C07	Leak caused by Android SDK	69%
C08	Complete leak	68%
C09	Leak on normal paths	63%
C10	Leak caused by unused resource	55%

The data obtained, according to the opinion of the study participants, indicate 6 cat- egories (C01, C02, C03, C04, C05 and C06) as the most relevant (with a value equal to or greater than the calculated median of 70%) to represent leak causes in mobile applica- tions, as shown in Table 9. The two most relevant categories were C01 and C02, which indicates that developers should be concerned with releasing resources both in the application's exception possibilities and within the *callback* methods (for example, *onStop()* or *viewWillDisappear()*) of the application lifecycle.

In relation to the answers to the question "What is your opinion on the categories presented? Are there any other categories of causes of leaks?", below is the full literal response of the participants who answered this optional question. Some observations can be highlighted, for example, participant P05 confirmed that developers may completely fail to release application resources.

"Well, I agree with C08 (developers completely fail to release the resources acquired after use), that's true, we don't usually look at these things." **P05**

Looking at the response of participant P07, it can be inferred that the Android docu- mentation needs to better highlight the need for developers to explicitly release resources so as not to leak resources.

"I think the Android platform documentation assumes developers that the operating system will take care of the leak, and this may lead the developer to think that resource leakage will not be an issue for their applications." **P07**

Participants P02, P10 and P19 consider the categories presented in this research to be satisfactory.

"Good." **P02**

"No, all of this is represented in the research." **P10**

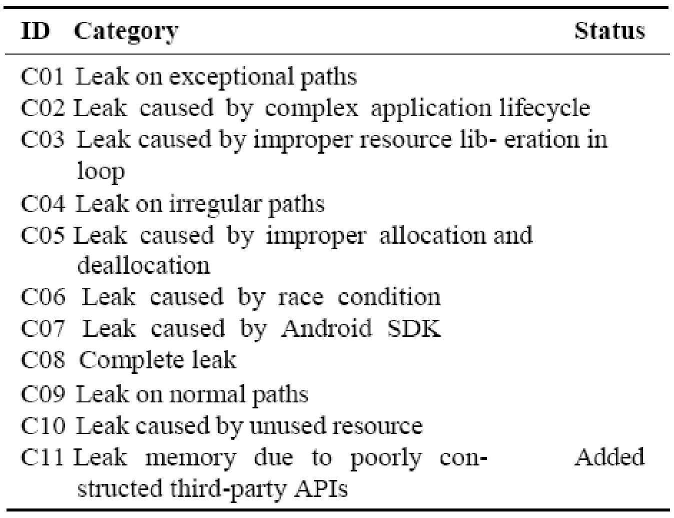
"I think the categories presented summarize the most important ones very well." **P19**

Participant P12 considers that some of these categories are not common to occur, as he believes that some are easy to be detected. However, based on the response of participants P05 and P07, it is possible to infer that some developers are still unaware of the importance of the correct release of application resources and, consequently, fail to do so.

"Some of them are not so common, for example, it is easy to capture when the resource is not "closed", even the code analyst can check and suggest. The race condition is really common and complicated, it takes time to identify and it is not so easy to debug. I think this is the most common in the projects I worked on." **P12**

In addition to the categories present in the initial set, a new category proposed by participant P06 was added with the name *"Leak memory due to poorly constructed third-party APIs"*. The same argued: *"And something that is very common is the choice of third-party APIs (Android-Arsenal has thousands) without many criteria ... and these have a considerable amount of problems like Memory leak. Incidentally, I dare say that this is the main source of memory leaks."* Although participant P06 only worked with *Windows*, this new category also applies to *Android* and *iOS*. In the Table 10 is shown the final list of categories, including the added category.

Table 10. Final list of categories of leak causes



ID	Category	Status
C01	Leak on exceptional paths	
C02	Leak caused by complex application lifecycle	
C03	Leak caused by improper resource lib- eration in loop	
C04	Leak on irregular paths	
C05	Leak caused by improper allocation and deallocation	
C06	Leak caused by race condition	
C07	Leak caused by Android SDK	
C08	Complete leak	
C09	Leak on normal paths	
C10	Leak caused by unused resource	
C11	Leak memory due to poorly con- structed third-party APIs	Added

Based on the responses obtained from the study, it is possible to state that the relevance ranking helped to establish a more comprehensive view of the leak causes in mobile applications.

5.2 Threats to Validity

The threats to the validity of the study were organized into 3 categories: construct validity, internal and external.

Construct validity: the study is characterized by analyzing the relevance of the categories of leak causes in the context of mobile applications. The categories were extracted from primary studies, through a systematic mapping study, involving experiments with mobile applications.

Internal validity: it was proposed to select development professionals who work in the development of mobile applications and who have experience in resource leaks. Thus, it was assumed that they are representative for the study population and that they can give the developers perspective on the categories of leak causes.

The instrument to be used (online form) went through a review and was submitted to a pilot study that pointed out improvements that were implemented.

External validity: study participants in general can be considered representative for the population of development professionals, as the questionnaire data on the participants' experience was used to filter only participants with the profile expected for this study.

5.3 Conclusions from the Study

The relevance analysis of each of the 10 categories, extracted by means of systematic mapping, by the participants of the opinion survey allowed to obtain a base set of 6 categories (*Leak in exceptional paths*, *Leak caused by the complexity of the life cycle of application, Leak caused by improper release of resources in loop, Leak on irregular paths, Leak caused by improper allocation and deallocation and Leak caused by race condition*) most relevant to the representation of leaks causes in mobile applications. It was also possible to add a new category (*Leak memory due to poorly constructed third-party APIs*) of leaks causes. This information is important for a better understanding of the origins of leaks, which allows us to propose an effective strategy to detect the majority of resource leaks.

VI. Conclusions and future work

In this paper, we explored and synthesized categories of leak causes in mobile applica- tions through a systematic mapping study. We characterize the studies we found based on 5 facets: (1) the types of contributions; (2) the types of research (proposal, validation, evaluation and report); (3) the types of resource leak that the study can identify; (4) the categories of causes of resource leaks; and (5) the input artifacts. These 5 facets belong to our research question **RQ 1**: What is the space for research of the technical literature in identifying resource leak in mobile applications?

To collect all relevant studies in our research scope, we searched six digital libraries, namely, *Scopus*, *IEEE Xplore*, *Science Direct*, *Engineering Village*, *Web of Science* and *ACM DL*. 1760 papers were found, and after the first filter (having read the title, abstract and keywords), 122 papers were left. After that, we performed the second filter (reading the entire paper) and as a result there were 57 papers. Through a detailed reading of this body of research, we derive a framework of attributes that, consequently, were used to characterize the studies in a structured way. The resulting systematic mapping study can be beneficial both for researchers in the area of resource leaks and for mobile application developers. We can highlight that the papers work with 14 different types of resources, with *Energy* (50.9%) and *WakeLock* (36.8%) being the most explored. We also point out that 10 categories of leak causes have been identified, which can serve as a guide for developers.

In order to analyze the relevance of these 10 categories, an opinion survey study was carried out with developers, who were found through searches on Linkedin social network as well as in development communities on Facebook. The survey was sent to 147 developers, and 22 of them responded. As a result, 6 most relevant categories were identified, namely, Leak on exceptional paths, Leak caused by complex application lifecycle, Leak caused by improper resource liberation in loop, Leak on irregular paths, Leak caused by improper allocation and deallocation and Leak caused by race condition. We also note that a new category (Leak memory due to poorly constructed third-party APIs) has been identified.

These results provide a basis to assist researchers in planning future work, identifying areas of research that need more attention, as well as helping developers to avoid resource leaks in their applications.

As future work, it is intended to extend the systematic mapping to include more digital libraries, as well as to extend the survey to include a greater number of developers.

Author Contributions: *Josias Gomes Lima:* Contributing to the conceptualization of the research, developing the methodology, analyzing the results and to the writing of the manuscript. *Joethe Moraes de Carvalho and Oswald Mesumbe Ekwoge:* Worked on most of the parts, introduction and empirical studies and to the writing of the manuscript. *Rafael Giusti and Arilo Claudio Dias-Neto:* Supervising the research.

Funding: This research was funded by CAPES (Coordination for the Improvement of Higher Edu- cation Personnel), Brazil, Finance Code 001; and Research Support Foundation State of Amazonas (FAPEAM) - PAPAC Project (Edital 005/2019).

Informed Consent Statement: According to the authors who collected and made the data publicly available, informed consent was obtained from all subjects involved in the study.

Data Availability Statement: Data will be made available on request.

Conflicts of Interest: The authors declare no conflict of interest.

References

[1]. Statista. Number of smartphone mobile network subscriptions worldwide from 2016 to 2023, with forecasts from 2023 to 2028, 2024. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (Accessed 19 July 2024).

[2]. Statista. Number of mobile app downloads worldwide from 2016 to 2023, 2024. https://www.statista.com/statistics/271644

[3]. /worldwide-free-and-paid-mobile-app-store-downloads/ (Accessed 19 July 2024).

[4]. Zhang, H.; Wu, H.; Rountev, A. Automated test generation for detection of leaks in Android applications. In Proceedings of the Proceedings of the 11th International Workshop on Automation of Software Test, 2016, pp. 64–70.

[5]. Zhang, S.; Zhu, J.; Liu, A.; Wang, W.; Guo, C.; Xu, J. A novel memory leak classification for evaluating the applicability of static analysis tools. In Proceedings of the 2018 IEEE International Conference on Progress in Informatics and Computing (PIC). IEEE, 2018, pp. 351–356.

[6]. de Sena, G.O.; Matias, R. A Systematic Mapping Review of Memory Leak Detection Techniques. In Proceedings of the 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). IEEE, 2018, pp. 264–270.

[7]. Khan, M.U.; Abbas, S.; Lee, S.; Abbas, A. Energy-leaks in android application development: Perspective and challenges. Journal of Theoretical and Applied Information Technology **2020**, 98, 3591–3601.

[8]. Aslanyan, H.; Gevorgyan, Z.; Mkoyan, R.; Movsisyan, H.; Sahakyan, V.; Sargsyan, S. Static analysis methods for memory leak detection: A survey. In Proceedings of the 2022 Ivannikov Memorial Workshop (IVMEM). IEEE, 2022, pp. 1–6.

[9]. Bhatt, B.N.; Furia, C.A. Automated repair of resource leaks in android applications. Journal of Systems and Software **2022**,

[10]. 192, 111417.

[11]. Liu, Y.; Wang, J.; Wei, L.; Xu, C.; Cheung, S.C.; Wu, T.; Yan, J.; Zhang, J. DroidLeaks: a comprehensive database of resource leaks in Android apps. Empirical Software Engineering **2019**, 24, 3435–3483.

[12]. Yan, D.; Yang, S.; Rountev, A. Systematic testing for resource leaks in Android applications. In Proceedings of the 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2013, pp. 411–420.

[13]. Guo, C.; Zhang, J.; Yan, J.; Zhang, Z.; Zhang, Y. Characterizing and detecting resource leaks in Android applications. In Proceedings of the 2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2013, pp. 389–398.

[14]. Petersen, K.; Vakkalanka, S.; Kuzniarz, L. Guidelines for conducting systematic mapping studies in software engineering: An update. Information and Software Technology **2015**, 64, 1–18.

[15]. Kitchenham, B.; Charters, S. Guidelines for performing systematic literature reviews in software engineering **2007**.

[16]. Petersen, K.; Feldt, R.; Mujtaba, S.; Mattsson, M. Systematic mapping studies in software engineering. In Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE) 12, 2008, pp. 1–10.

[17]. Xu, Z.; Wen, C.; Qin, S. State-taint analysis for detecting resource bugs. Science of Computer Programming **2018**, 162, 93–109.

[18]. Banerjee, A.; Chong, L.K.; Ballabriga, C.; Roychoudhury, A. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. IEEE Transactions on Software Engineering **2017**, 44, 470–490.

[19]. Jiang, H.; Yang, H.; Qin, S.; Su, Z.; Zhang, J.; Yan, J. Detecting energy bugs in Android apps using static analysis. In Proceedings of the International Conference on Formal Engineering Methods. Springer, 2017, pp. 192–208.

[20]. Wu, T.; Liu, J.; Xu, Z.; Guo, C.; Zhang, Y.; Yan, J.; Zhang, J. Light-weight, inter-procedural and callback-aware resource leak detection for android apps. IEEE Transactions on Software Engineering **2016**, 42, 1054–1076.

[21]. Kim, K.; Cha, H. WakeScope: runtime WakeLock anomaly management scheme for Android platform. In Proceedings of the 2013 Proceedings of the International Conference on Embedded Software (EMSOFT). IEEE, 2013, pp. 1–10.

[22]. Liu, Y.; Xu, C. Veridroid: Automating android application verification. In Proceedings of the Proceedings of the 2013 Middleware Doctoral Symposium, 2013, pp. 1–6.

[23]. Shahrokni, A.; Feldt, R. A systematic review of software robustness. Information and Software Technology **2013**, 55, 1–17.
[24]. Song, W.; Zhang, J.; Huang, J. ServDroid: detecting service usage inefficiencies in Android applications. In Proceedings of the Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2019, pp. 362–373.
[25]. Song, S.; Wedyan, F.; Jararweh, Y. Empirical Evaluation of Energy Consumption for Mobile Applications. In Proceedings of the 2021 12th International Conference on Information and Communication Systems (ICICS). IEEE, 2021, pp. 352–357.
[26]. Palomba, F.; Di Nucci, D.; Panichella, A.; Zaidman, A.; De Lucia, A. On the impact of code smells on the energy consumption of mobile applications. Information and Software Technology **2019**, 105, 43–55.
[27]. Nguyen, T.T.; Vu, P.M.; Nguyen, T.T. An empirical study of exception handling bugs and fixes. In Proceedings of the Proceedings of the 2019 ACM Southeast Conference, 2019, pp. 257–260.
[28]. Pathak, A.; Jindal, A.; Hu, Y.C.; Midkiff, S.P. What is keeping my phone awake? Characterizing and detecting no-sleep energy bugs in smartphone apps. In Proceedings of the Proceedings of the 10th international conference on Mobile systems, applications, and services, 2012, pp. 267–280.
[29]. Vilk, J.; Berger, E.D. BLeak: automatically debugging memory leaks in web applications. ACM SIGPLAN Notices **2018**, 53, 15–29.
[30]. Ma, J.; Liu, S.; Jiang, Y.; Tao, X.; Xu, C.; Lu, J. LESdroid: a tool for detecting exported service leaks of Android applications. In Proceedings of the Proceedings of the 26th Conference on Program Comprehension, 2018, pp. 244–254.
[31]. Ferrari, A.; Gallucci, D.; Puccinelli, D.; Giordano, S. Detecting energy leaks in Android app with POEM. In Proceedings of the 2015 IEEE International Conference on Pervasive Computing and Communication Workshops (PerCom Workshops). IEEE, 2015,
[32]. pp. 421–426.
[33]. Zein, S.; Salleh, N.; Grundy, J. Static analysis of android apps for lifecycle conformance. In Proceedings of the 2017 8th International Conference on Information Technology (ICIT). IEEE, 2017, pp. 102–109.
[34]. Zhang, L.; Gordon, M.S.; Dick, R.P.; Mao, Z.M.; Dinda, P.; Yang, L. Adel: An automatic detector of energy leaks for smartphone applications. In Proceedings of the Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis, 2012, pp. 363–372.
[35]. Wu, L.; Lu, Y.; Qi, J.; Cai, S.; Deng, B.; Ming, Z. Bug Analysis of Android Applications Based on JPF. In Proceedings of the International Conference on Smart Computing and Communication. Springer, 2016, pp. 173–182.
[36]. Jun, M.; Sheng, L.; Shengtao, Y.; Xianping, T.; Jian, L. LeakDAF: An automated tool for detecting leaked activities and fragments of Android applications. In Proceedings of the 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC). IEEE, 2017, Vol. 1, pp. 23–32.
[37]. Vekris, P.; Jhala, R.; Lerner, S.; Agarwal, Y. Towards verifying android apps for the absence of no-sleep energy bugs. In Proceedings of the Presented as part of the 2012 Workshop on Power-Aware Computing and Systems, 2012.
[38]. Hoshieah, N.; Zein, S.; Salleh, N.; Grundy, J. A static analysis of android source code for lifecycle development usage patterns.
[39]. Journal of Computer Science **2019**, 15, 92–107.
[40]. Toffalini, F.; Sun, J.; Ochoa, M. Practical static analysis of context leaks in Android applications. Software: Practice and Experience **2019**, 49, 233–251.
[41]. Wu, H.; Yang, S.; Rountev, A. Static detection of energy defect patterns in Android applications. In Proceedings of the Proceedings of the 25th International Conference on Compiler Construction, 2016, pp. 185–195.
[42]. Chang, B.Y.E. Refuting Heap Reachability. In Proceedings of the International Conference on Verification, Model Checking, and Abstract Interpretation. Springer, 2014, pp. 137–141.
[43]. Chen, B.; Li, X.; Zhou, X. PowerSensor: A method for power optimization of smartphone through sensing wakelock application. In Proceedings of the 2017 9th International Conference on Wireless Communications and Signal Processing (WCSP). IEEE, 2017,
[44]. pp. 1–6.
[45]. Ghanem, T.; Zein, S. A Model-based approach to assist Android Activity Lifecycle Development. In Proceedings of the 2020 4th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT). IEEE, 2020, pp. 1–12.
[46]. Pereira, R.B.; Ferreira, J.F.; Mendes, A.; Abreu, R. Extending EcoAndroid with Automated Detection of Resource Leaks **2022**.
[47]. Lu, Y.; Pan, M.; Pei, Y.; Li, X. Detecting resource utilization bugs induced by variant lifecycles in Android. In Proceedings of the Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022, pp. 642–653.
[48]. Liu, Q.; Pan, L.; Cui, B.; Yan, J.; Zhang, J. Dynamic Detection of AsyncTask Related Defects. In Proceedings of the 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS). IEEE, 2021, pp. 357–366.
[49]. Amalfitano, D.; Riccio, V.; Tramontana, P.; Fasolino, A.R. Do memories haunt you? An automated black box testing approach for detecting memory leaks in android apps. IEEE Access **2020**, 8, 12217–12231.
[50]. Wu, H.; Zhang, H.; Wang, Y.; Rountev, A. Sentinel: generating GUI tests for sensor leaks in Android and Android wear apps.
[51]. Software Quality Journal **2020**, 28, 335–367.
[52]. Khan, M.U.; Lee, S.U.J.; Wu, Z.; Abbas, S. Wake Lock Leak Detection in Android Apps Using Multi-Layer Perceptron. Electronics **2021**, 10, 2211.
[53]. Riganelli, O.; Micucci, D.; Mariani, L. Controlling interactions with libraries in android apps through runtime enforcement. ACM Transactions on Autonomous and Adaptive Systems (TAAS) **2019**, 14, 1–29.
[54]. Nguyen, T.; Vu, P.; Nguyen, T. Code recommendation for exception handling. In Proceedings of the Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 1027–1038.
[55]. Hall, S.; Nataraj, S.; Kim, D.K. Detecting no-sleep energy bugs using reference counted variables. In Proceedings of the Proceedings of the 5th International Conference on Mobile Software Engineering and Systems, 2018, pp. 161–165.
[56]. Qian, J.; Zhou, D. Prioritizing test cases for memory leaks in android applications. Journal of Computer Science and Technology **2016**,
[57]. 31, 869–882.
[58]. Shahriar, H.; North, S.; Mawangi, E. Testing of memory leak in Android applications. In Proceedings of the 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering. IEEE, 2014, pp. 176–183.
[59]. Araujo, J.; Alves, V.; Oliveira, D.; Dias, P.; Silva, B.; Maciel, P. An investigative approach to software aging in android applications. In Proceedings of the 2013 IEEE international conference on systems, man, and cybernetics. IEEE, 2013, pp. 1229–1234.
[60]. Naseer, A.; Nadeem, A.; Zaman, Q.U. A GUI Based Approach to Detect Energy Bugs in Android Applications. In Proceedings of the 2021 16th International Conference on Emerging Technologies (ICET). IEEE, 2021, pp. 1–6.
[61]. Alam, F.; Panda, P.R.; Tripathi, N.; Sharma, N.; Narayan, S. Energy optimization in Android applications through wakelock placement. In Proceedings of the 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE). IEEE, 2014, pp. 1–4.
[62]. Khan, M.U.; Lee, S.U.J.; Abbas, S.; Abbas, A.; Bashir, A.K. Detecting Wake Lock Leaks in Android Apps Using Machine Learning.

IEEE Access **2021**, 9, 125753–125767.
[63]. Sakhare, P.B.; Kim, D.K.; Hamdi, M. Detecting No-Sleep Bugs Using Sequential Reference Counts. In Proceedings of the 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC). IEEE, 2019, Vol. 1, pp. 940–941.
[64]. Xia, M.; He, W.; Liu, X.; Liu, J. Why application errors drain battery easily? A study of memory leaks in smartphone apps. In Proceedings of the Proceedings of the Workshop on Power-Aware Computing and Systems, 2013, pp. 1–5.
[65]. Santhanakrishnan, G.; Cargile, C.; Olmsted, A. Memory leak detection in android applications based on code patterns. In Proceedings of the 2016 International Conference on Information Society (i-Society). IEEE, 2016, pp. 133–134.
[66]. Ahn, S. Automation of Memory Leak Detection and Correction on Android JNI (poster). In Proceedings of the Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services, 2019, pp. 533–534.
[67]. Le, H.A. Analyzing energy leaks of android applications using event-b. Mobile Networks and Applications **2021**, 26, 1329–1338.
[68]. Saju, N.; Garg, J.; Sehgal, R.; Nagpal, R. Green Mining for Android Based Applications Using Refactoring Approach. In Proceedings of the 2021 9th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO). IEEE, 2021, pp. 1–6.
[69]. Sehgal, R.; Mehrotra, D.; Nagpal, R.; Sharma, R. Green software: Refactoring approach. Journal of King Saud University-Computer and Information Sciences **2020**.
[70]. Riganelli, O.; Micucci, D.; Mariani, L. From source code to test cases: A comprehensive benchmark for resource leak detection in android apps. Software: Practice and Experience **2019**, 49, 540–548.
[71]. Wu, H.; Wang, Y.; Rountev, A. Sentinel: Generating GUI tests for Android sensor leaks. In Proceedings of the 2018 IEEE/ACM 13th International Workshop on Automation of Software Test (AST). IEEE, 2018, pp. 27–33.
[72]. Wang, C.; Liu, J.; Peng, X.; Liu, Y.; Lou, Y. LLM-based Resource-Oriented Intention Inference for Static Resource Leak Detection, 2024, [arXiv:cs.SE/2311.04448].
[73]. Cui, B.; Wang, M.; Zhang, C.; Yan, J.; Yan, J.; Zhang, J. Detection of Java Basic Thread Misuses Based on Static Event Analysis. In Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2023, pp. 1049–1060.
[74]. Nanavati, J.; Patel, S.; Patel, U.; Patel, A. Critical Review and Fine-Tuning Performance of Flutter Applications. In Proceedings of the 2024 5th International Conference on Mobile Computing and Sustainable Informatics (ICMCSI). IEEE, 2024, pp. 838–841.
[75]. Shahoor, A.; Khamit, A.Y.; Yi, J.; Kim, D. LeakPair: Proactive repairing of memory leaks in single page web applications. In Proceedings of the 2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2023, pp. 1175–1187.
[76]. Banerjee, A.; Chong, L.K.; Chattopadhyay, S.; Roychoudhury, A. Detecting energy bugs and hotspots in mobile apps. In Proceedings of the Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering, 2014, pp. 588–598.
[77]. Basili, V.R.; Caldiera, G.; Rombach, H.D. The experience factory. Encyclopedia of Software Eng.: Vol **1994**, 1, 469–476.
[78]. Dias-Neto, A.C.; Travassos, G.H. Surveying model based testing approaches characterization attributes. In Proceedings of the Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, 2008, pp. 324–326.
[79]. Hamburg, M. Basic Statistics: A Modern Approach. Journal of the Royal Statistical Society **1980**, 143.