Article

LeakPred: An Approach for Identifying Components with Resource Leaks in Android Mobile Applications

Josias Gomes Lima *, Rafael Giusti and Arilo Claudio Dias-Neto

Institute of Computing, Federal University of Amazonas, Manaus 69080-900, Amazonas, Brazil; rgiusti@icomp.ufam.edu.br (R.G.); ariloclaudio@gmail.com (A.C.D.-N.)
* Correspondence: josias@icomp.ufam.edu.br

Abstract: *Context*: Mobile devices contain some resources, for example, the camera, battery, and memory, that are allocated, used, and then deallocated by mobile applications. Whenever a resource is allocated and not correctly released, a defect called a resource leak occurs, which can cause crashes and slowdowns. *Objective*: In this study, we intended to demonstrate the usefulness of the *LeakPred* approach in terms of the number of components with resource leak problems identified in applications. *Method*: We compared the approach's effectiveness with three state-of-the-art methods in identifying leaks in 15 Android applications. *Result*: *LeakPred* obtained the best median (85.37%) of components with identified leaks, the best coverage (96.15%) of the classes of leaks that could be identified in the applications, and an accuracy of 81.25%. The *Android Lint* method achieved the second best median (76.92%) and the highest accuracy (100%), but only covered 1.92% of the leak classes. *Conclusions*: *LeakPred* is effective in identifying leaky components in applications.

Keywords: mobile apps; resource leak; tool evaluation; database; Android; static analysis; application

Citation: Lima, J.G.; Giusti, R.; Dias-Neto, A.C. LeakPred: An Approach for Identifying Components with Resource Leaks in Android Mobile Applications. *Computers* **2024**, *13*, 140. https://doi.org/10.3390/computers13060140

Academic Editors: Phivos Mylonas, Katia Lida Kermanidis and Manolis Maragoudakis

Received: 2 May 2024
Revised: 23 May 2024
Accepted: 31 May 2024
Published: 3 June 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The number of mobile devices used in 2022 was almost 16 billion [1], nearly twice the world population, which was reported as 8.1 billion this year [2]. These devices include resources such as memory, sensors, microphone, GPS (Global Positioning System), bluetooth, camera, and NFC (Near Field Communication), which must be handled by applications to provide functionalities to the users. During the development of a mobile application, developers implement the acquisition, use, and release of such resources. For example, when one opens a *camera* application, the camera resource is acquired and used to take pictures, and, once the application is closed, the camera resource must be released. However, sometimes these resources are acquired by a mobile application and not properly released, resulting in a defect called resource leak, which can lead to unnecessary battery consumption, crashes, and slowdowns [3].

In recent years, some methods have been developed to identify resource leaks in mobile applications. For example, *Android Studio* has the *Android Lint* method that inspects code and shows resource leaks as well as other defects [4]. The *FindBugs* method performs code analysis in order to find structural problems and resource leaks [5].

In a previous work, the authors presented the *LeakPred* approach, which manages to identify resource leaks in Android applications using machine learning (ML) [6]. A study was carried out to verify which is the best ML classifier to be used in *LeakPred* and showed that the KNN (K-Nearest Neighbor) and DNN (Deep Neural Network) classifiers obtained the best results [6].

Continuing with the evolution of this approach, we want to demonstrate the usefulness of *LeakPred* by helping app developers to identify components (in this work, a component is considered a method of a class of an object-oriented program) with resource leak problems. So, we validated this approach with three state-of-the-art methods through a controlled

experiment. These state-of-the-art methods identify a limited number of kinds of resource classes with leaks, and the current number of these resources class is greater. Therefore, this article will present a comparative study.

The three state-of-the-art methods used in the study were *Android Lint* [4], *FindBugs* [5], and *Infer* [7]. This new study was carried out in the form of a controlled experiment, where 15 mobile applications were used, which were randomly selected from the list of applications available at [8].

The results of this study indicate the feasibility of the *LeakPred* approach to assist developers in identifying components with resource leaks, since the approach reached the best median (85.37%) of components with identified resource leaks and had the highest coverage (96.15%) of classes of resource leaks in applications. The main contributions of this work are as follows:

- The *CompLeaks* database was updated with 22 more applications containing 467 components with resource leaks, available at https://bit.ly/3zLmgFj (accessed on 1 May 2024);
- An analysis of the effectiveness of the *LeakPred* approach was conducted in relation to three state-of-the-art methods in identifying leaky components in Android mobile applications;
- The material necessary to replicate this study is available at https://bit.ly/3o8U1gU (accessed on 1 May 2024).

This article is organized as follows: Section 2 shows the related work, Section 3 presents the planning and execution of the study, Section 4 shows the results found, Section 5 presents the study discussions, and Section 6 discusses the threats to the validity of the study. Finally, Section 7 shows the conclusions and future work.

2. Related Work

In recent years, many resource leak identification techniques have been proposed in order to help Android developers to properly manage device resources. However, there was no database of applications with leaks, so in order to evaluate these new techniques, it was necessary to make an effort to find applications to be used in studies [8]. To reduce this problem, Liu et al. [8] presented the *DroidLeaks* database and also revealed some characteristics of leaks in Android applications, as well as some defect patterns in resource management. To demonstrate the usefulness of the database, they performed a study comparing eight methods in relation to detecting leaks in Android mobile applications, where the *Android Lint* and *FindBugs* methods did not have any false positives, but also, they have not achieved the best detection rates [8].

Some state-of-the-art techniques for identifying resource leaks in Android applications are *LeakPred*, *Infer*, *FindBugs*, and *Android Lint*, which will be briefly explained below.

Lima et al. [6] proposed an approach called *LeakPred*, which uses machine learning to identify leaks in Android application components. In ML, a database is needed, and Lima et al. presented the *CompLeaks* database, which was based on the *DroidLeaks* database. Aiming at analyzing which would be the best classifier to be used, a study was carried out comparing six classifiers commonly used in studies for defect prediction, where a KNN and DNN had the best results, and in this study, the DNN classifier was used, which had the best result (78.93%) in the ROC AUC metric. This evaluation metric was chosen because it is recommended for when the database is unbalanced [6].

Facebook [7] uses the *Infer* method, which uses static analysis. It can parse Java and C/C++/Objective-C code. After parsing, it will produce a list of possible defects (including resource leaks). Some of the companies that use this method are *Amazon Web Services*, *Spotify*, *Uber*, *WhatsApp*, *Microsoft*, *Mozilla*, and *Instagram* [7].

Pugh et al. [5] presented the static analysis method *FindBugs*. It can identify more than 200 defect patterns, such as null pointers, infinite recursive loops, resource leaks, the misuse of Java libraries, and deadlocks. The project is open-source, has been downloaded over 230,000 times, and is used by many large companies and financial institutions [5].

Android [4] provides the *Android Lint* method. This method helps to find code with an inefficient structure that can affect the reliability and efficiency of Android applications (including resource leaks) and make code maintenance difficult. Possible defects and improvements are grouped according to the following criteria: accuracy, security, performance, usability, accessibility, and internationalization [4].

3. Study Planning

This feasibility study aimed to compare the identification of components with resource leaks using the *LeakPred* approach with state-of-the-art methods, in which mobile applications from the Android platform were used. With this, it was expected that the results obtained and the body of knowledge resulting from the study's conduction will provide information that will allow for the evolution of the approach and improve its use in the identification of leaking components in Android mobile applications.

The purpose of this study was to answer the following question: "Is the use of the *LeakPred* approach feasible in analyzing its effectiveness in comparison to representative state-of-the-art methods in identifying components with resource leaks in mobile applications on Android platform?". Effectiveness is understood in this study as the number of components with resource leaks identified in relation to the total number of components with resource leaks in mobile applications. In this context, the list of components identified with resource leaks by the *LeakPred* approach was compared with the list identified by each of the state-of-the-art methods. Therefore, the research question defined for this study was the following:

RQ1. How effective is the *LeakPred* approach in relation to the identification performed by state-of-the-art methods regarding the number of components with identified resource leaks?

To help answer the research question, five metrics were chosen: the True Positive Rate (TPR, also known as Recall), which is the probability that a leaking component will be identified as leaking (Equation (1)); the False Discovery Rate (FDR), which is the expected ratio of the number of non-leaking components identified as leaking (false discoveries) to the total number of leaking components identified (Equation (2)); the accuracy; the precision; and the f1-score. In Equations (1) and (2), TP is the number of components with leaks correctly identified, P is the number of leaks that could be identified, and FP is the number of non-leaking components identified as having a leak.

$$TruePositiveRate = \frac{TP}{P} \tag{1}$$

$$FalseDiscoveryRate = \frac{FP}{FP + TP} \tag{2}$$

3.1. Methodology for Identifying Resource Leaking Components

To identify components with resource leaks, the same protocol used to assemble the *DroidLeaks* database in [8] was followed. Figure 1 shows an overview of the process followed formed by two steps: (1) search for keywords in the commit logs (some example keywords appear in Table 1) and commit code differences (examples of keywords are shown in Table 1); and (2) the manual validation of resource leaks identified in the commits found in step 1. The process presented in this subsection was followed in the applications discussed in the next subsection, and some of the applications with identified resource leaks were used to evaluate the effectiveness of the approach *LeakPred*.

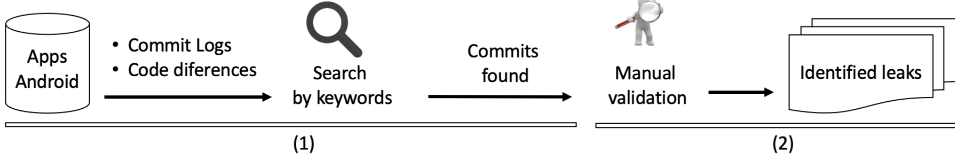

Figure 1. Resource leak identification process.

Table 1. Keywords for commit logs and code differences.

Commit logs		
leak	leakage	release
recycle	cancel	unload
unlock	unmount	unregister
close		
Code differences		
.close(	.stop(	.stopPreview(
.release(	.abandonAudioFocus(	.stopFaceDetection(
.removeUpdates(	.cancel(	.unregisterListener(
.unlock(	.disableNetwork(	

3.2. Mobile Applications Selection

Liu et al. [8] provided a list of 170 mobile applications that meet the following criteria:

1. They have more than 10,000 downloads in the store (application is popular);
2. They have a public defect tracking system (defects are trackable);
3. The application's code repository has over 100 code reviews (application is actively maintained);
4. They have at least 1000 lines of Java source code (the application has a medium or high level of complexity) [8].

For the creation of the *DroidLeaks* database, 34 applications of these 170 were used. Therefore, for this study, 32 applications were randomly selected among the remaining 136 (Figure 2). These 32 selected applications are shown in Table 2, of which 22 were used to increase the database, and 5 of these 22 were also randomly selected to be used in the comparison study between the methods, and the other 17 were used together with the old version of the database (*CompLeaks*) for the training of the *LeakPred* approach. Therefore, 15 applications were used in this study. More information, such as the component name and leaked resource class, is found at https://bit.ly/3o7XJr9 (accessed on 1 May 2024).

Table 2. The 32 apps selected randomly.

ID	App Name	Leak Quantity	Use	Successfully Compiled
1	Aard	8	database/experiment	no
2	Android-Project	20	database	**
3	BART_Runner	9	database	**
4	BeeCount	13	database	**
5	BeTrains-for-Android	52	database/experiment	no
6	BetterWeather	1	database	**
7	Bitcoin	26	database	**
8	BombusMod	78	database	**
9	Dimmer	1	database	**
10	FareBot	3	database/experiment	no
11	FBReader TTS+ Plugin	10	database	**
12	GPSLogger	74	database	**
13	KeepScore	2	database	**

Table 2. *Cont.*

ID	App Name	Leak Quantity	Use	Successfully Compiled
14	Navit	9	database	**
15	OI Notepad	12	database/experiment	yes
16	OI Safe	30	database/experiment	yes
17	Pedometer	41	database	**
18	Public Transport Timisoara	21	database	**
19	RedReader Beta	28	database	**
20	Shattered Pixel Dungeon	25	database	**
21	WiFi Analyzer	3	database	**
22	Yubico Authenticator	1	database	**
23	AnyMemo	12	experiment	yes
24	Avare	22	experiment	yes
25	OpenDocument Reader	-	experiment	no
26	Tinfoil for Facebook	-	experiment	no
27	Wikipedia	-	experiment	no
28	Seafile	42	experiment	yes
29	SMS Backup+	-	experiment	no
30	Chess	-	experiment	no
31	EP Mobile	-	experiment	no
32	Persian Calendar	-	experiment	no

Legend: "-" means that the amount of resource leakage in the application is unknown. Where "**" appears, it means that no attempt was made to compile the application.

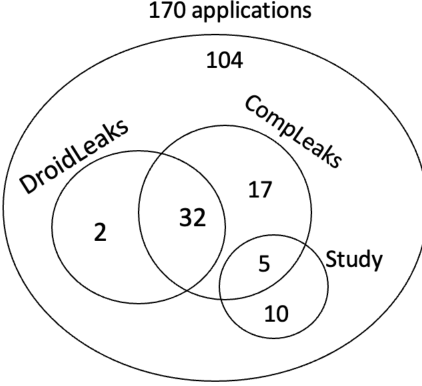

Figure 2. Mobile application selection.

3.3. Tools

For this study, some of the state-of-the-art methods for resource leak detection in mobile applications were selected, which were selected from a systematic mapping that we performed on techniques related to resource leaks. Those selected are as follows:

1. **Android Lint** provides a code scan that helps identify resource leaks and other structural code issues, using static analysis to check if the code breaks existing lint rules [4]. A lint rule has the following information: id, summary, explanation, category, priority, severity, detector class (responsible for detecting the occurrence of the issue; could be written using UAST—Universal Abstract Syntax Tree), and scope [9].
2. **FindBugs** is a program that uses static analysis to look for defects (including resource leaks) in Java code [5]. It often syntactically matches source code with faulty code patterns, but also uses data flow analysis to check for defects [10].
3. **Infer** checks using static analysis for resource leaks and other defects [7]. It develops a compositional, bottom-up variant of the RHS inter-procedural analysis algorithm [11].

These methods will serve as a basis for comparisons with the *LeakPred* approach [6].

3.4. Execution of the Study

The execution of this study had three steps: (1) compiling the applications (some methods need the application compiled), (2) executing the methods, and (3) listing the classes of resource leaks by method. These steps are presented in Figure 3. For steps 1 and 2, a maximum time of 2 weeks was provided (this amount of time was chosen as it was believed to be reasonable for understanding how to prepare the environment to compile an application or run a method) to try to solve compilation problems or for the execution of each application and method. If even with this amount of time, it was not possible to solve the problem, the application or the method would not be used in the study.

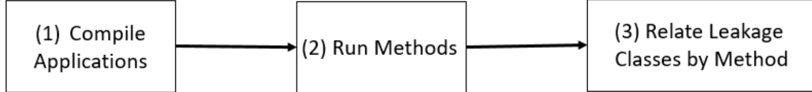

Figure 3. Study execution process of this study.

The first step was to compile the 15 applications randomly selected for this study (the applications contain the word experiment in the use column of Table 2). Applications with gray background in the table and with ids *1*, *5*, *10*, *25*, *26*, *27*, *29*, *30*, *31*, and *32* were not compiled due to library dependencies and design errors. It was only possible to compile the applications that have the green background in the table, namely, *OI Notepad (15)*, *OI Safe (16)*, *AnyMemo (23)*, *Avare(24)*, and *Seafile(28)*. Therefore, of the 15 methods previously selected, only five could be used in this study, as some of the methods require the application to be compiled.

The second step was to implement state-of-the-art methods. The *Android Lint*, *FindBugs*, and *Infer* methods were successfully executed. Thus, the three methods plus the *LeakPred* approach were executed to analyze resource leaks in the five mobile applications.

The third step was to make a list of resource leak classes that each method could identify in the five applications (Table 3), as each method identifies some types of resource leaks and not all methods provide a list of the resource classes that they can identify. For this, we used the list of leaks identified by at least one of the methods in the five applications and the list of leaks that each method could identify mapped in *DroidLeaks* [8]. In the next subsection, the analysis of the results will be shown, and the detection rate of each method was based only on the leaks that it could identify in each application.

Table 3. The resource leak classes that could be identified by each method in the *OI Notepad*, *OI Safe*, *AnyMemo*, *Avare*, and *Seafile* applications.

Target Java Class	LeakPred	Infer	FindBugs	Android Lint
android.app.AlarmManager	X	-	-	-
android.app.NotificationManager	X	-	-	-
android.app.Service	X	-	-	-
android.bluetooth.BluetoothSocket	X	-	-	-
android.content.BroadcastReceiver	X	-	-	-
android.content.res.XmlResourceParser	X	-	-	-
android.content.ServiceConnection	X	-	-	-
android.database.Cursor	X	X	-	X
android.database.sqlite.SQLiteDatabase	X	X	-	-
android.databaseMatrixCursor	-	X	-	-
android.graphics.Bitmap	X	X	X	-
android.hardware.SensorManager	X	-	-	-
android.location.LocationManager	X	-	X	-
android.media.MediaMetadataRetriever	X	-	X	-

Table 3. *Cont.*

Target Java Class	LeakPred	Infer	FindBugs	Android Lint
android.media.MediaPlayer	X	-	-	-
android.media.SoundPool	X	-	X	-
android.os.AsyncTask	X	-	-	-
android.os.CountDownTimer	X	-	-	-
android.speech.tts.TextToSpeech	X	-	-	-
android.support.v4.app.NotificationManagerCompat	X	-	-	-
com.github.kevinsawicki.http.HttpRequest	X	-	-	-
java.io.BufferedInputStream	X	X	X	-
java.io.BufferedOutputStream	X	X	X	-
java.io.BufferedReader	X	-	X	-
java.io.BufferedWriter	X	X	X	-
java.io.DataInputStream	X	X	-	-
java.io.DataOutputStream	X	-	X	-
java.io.File	X	-	-	-
java.io.FileInputStream	X	X	X	-
java.io.FileOutputStream	X	X	X	-
java.io.FileReader	-	X	-	-
java.io.FileWriter	X	X	-	-
java.io.InputStream	X	X	X	-
java.io.InputStreamReader	X	X	X	-
java.io.OutputStream	X	X	X	-
java.io.PrintWriter	X	-	-	-
java.io.RandomAccessFile	X	X	-	-
java.io.Reader	X	X	-	-
java.io.Writer	X	X	-	-
java.lang.Thread	X	-	X	-
java.net.DatagramPacket	X	-	-	-
java.net.DatagramSocket	X	-	-	-
java.net.HttpURLConnection	X	-	-	-
java.net.URL	X	-	-	-
java.net.URLConnection	X	-	-	-
java.text.Format	X	-	-	-
java.util.Timer	X	-	-	-
java.util.zip.ZipFile	X	X	X	-
java.util.zip.ZipInputStream	X	-	-	-
java.util.zip.ZipOutputStream	X	-	-	-
org.nocrala.tools.gis.data.esri.shapefile.ShapeFileReader	X	-	-	-

Legend: "X" means that the method can identify resource class resource leaks, and "-" means that it cannot.

4. Results

The results of this study will be shown for each of the mobile applications, starting with the application *OI Notepad* followed by the results for the applications *OI Safe*, *AnyMemo*, *Avare*, and *Seafile*. More information (for example, resource class or which method identified each leak) about the components with identified resource leaks in each of the applications is available at https://bit.ly/3mq5RTv (accessed on 1 May 2024).

4.1. Results: OI Notepad Application

Table 4 presents the 20 resource leaks identified by the methods in the application *OI Notepad*. For a better understanding of how many resource leaks were identified by more than one method, Figure 4 shows a Venn diagram showing resource leaks and false positives, in which it can be seen that *LeakPred* identified 17 leaks, while identified *FindBugs* 0 (zero), and the methods *Infer* and *Android Lint* identified 10 leaks each. Still, in Figure 4, the detection rates of resource leaks and false positives are presented, where it is possible to observe that the *LeakPred* approach obtained the best detection coverage, with 85%, as well as the highest percentage of false positives, with 37.04%. The *FindBugs* method did not find either of the two leaks it could identify, and it did not have any false positives either.

Table 4. Components with resource leaks in the *OI Notepad* application.

Package	Class	Component	Parameters	Return
org.openintents.notepad.theme	ThemeUtils	addThemeInfos	(PackageManager,String,ApplicationInfo,List)	void
org.openintents.notepad.search	SearchQueryResultsActivity	doSearchQuery	(Intent,String)	void
org.openintents.notepad.search	SearchSuggestionProvider	getSuggestions	(String,String[])	Cursor
org.openintents.notepad.search	SearchSuggestionProvider	refreshShortcut	(String,String[])	Cursor
org.openintents.notepad.search	FullTextSearch	getCursor	(Context,String)	Cursor
org.openintents.notepad.noteslist	NotesList	updateTagList	()	void
org.openintents.notepad.noteslist	NotesList	sendNoteByEmail	(long)	void
org.openintents.notepad.noteslist	NotesList	encryptNote	(long,String)	void
org.openintents.notepad.noteslist	NotesList	saveFile	(Uri,File)	void
org.openintents.notepad.noteslist	NotesList	writeToFile	(File,String)	void
org.openintents.util	ProviderUtils	getAffectedRows	(SQLiteDatabase,String,String,String[])	long[]
org.openintents.notepad.activity	SaveFileActivity	writeToFile	(Context,File,String)	void
org.openintents.notepad.activity	SaveFileActivity	getFilenameFromNoteTitle	(Uri)	Uri
org.openintents.notepad	NoteEditor	onPause	()	void
org.openintents.notepad	NoteEditor	deleteNote	()	void
org.openintents.notepad	NoteEditor	importNote	()	void
org.openintents.notepad	NotePadProvider	query	(Uri,String[],String,String[],String)	Cursor
org.openintents.notepad	NotePadProvider	insert	(Uri,ContentValues)	Uri
org.openintents.notepad	NotePadProvider	delete	(Uri,String,String[])	int
org.openintents.notepad	NotePadProvider	update	(Uri,ContentValues,String,String[])	int

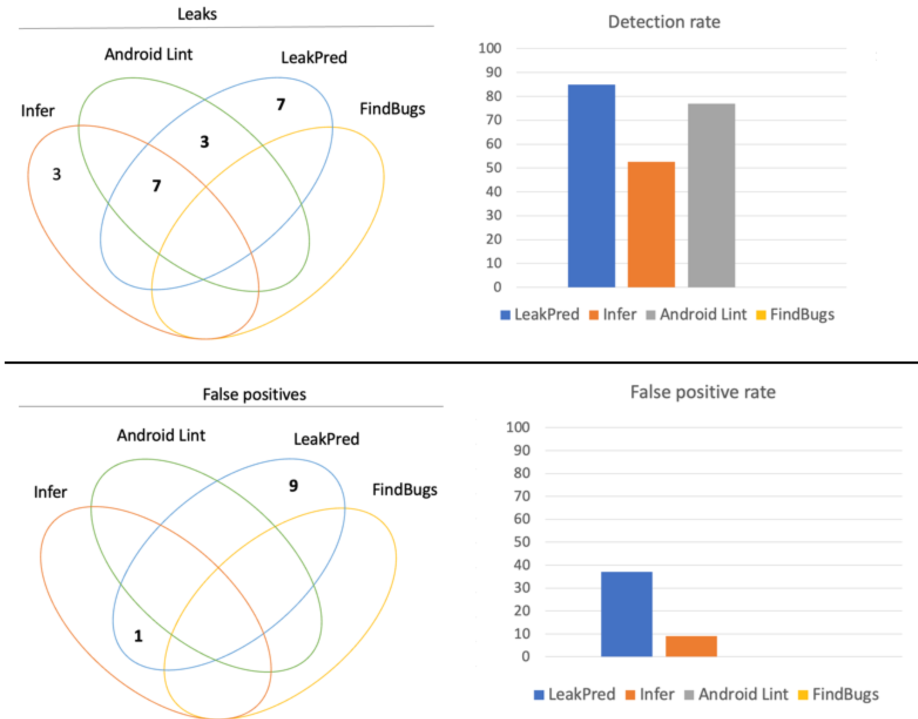

Figure 4. Components with leaks, detection rates, and false positives in *OI Notepad* application.

The three leaks that the *LeakPred* approach failed to identify were two from the *java.io.BufferedWriter* class (the two components are about 81% the same) and one from the *android.database.Cursor*. Regarding the false positives reported by the *LeakPred* approach, nine are from the *android.database.Cursor* class and one from the *java.io.InputStream* class. We can consider that the approach could decrease the percentage of detected false positives.

4.2. Results: OI Safe Application

Table 5 presents the 13 resource leaks discovered by the methods in the *OI Safe* application, and Figure 5 shows the number of resource leaks and false positives that each method identified and how many were identified by more than one method. For example, *LeakPred* identified eight leaks, and *Infer*, seven. Also, in Figure 5, the detection rates of resource leaks and false positives are shown, where the method *Android Lint* scored 100% in coverage, identifying the only leaky component it could identify in this application. Next is the *Infer* method with 70% coverage, and then the *LeakPred* approach with 61.54% coverage, identifying 8 out of 13 possible leaks to be identified.

Table 5. Leaking components in the *Safe* application.

Package	Class	Component	Parameters	Return
org.openintents.util	SecureDelete	delete	(File)	boolean
org.openintents.safe.service	AutoLockService	onDestroy	()	void
org.openintents.safe	CSVWriter	read	(Clob)	String
org.openintents.safe	CSVWriter	close	()	void
org.openintents.safe	CryptoHelper	encryptFileWithSessionKey	(ContentResolver,Uri)	Uri
org.openintents.safe	CryptoHelper	decryptFileWithSessionKey	(Context,Uri)	Uri
org.openintents.safe	DBHelper	DBHelper	(Context)	void
org.openintents.safe	AskPassword	keypadOnDestroy	()	void
org.openintents.safe	Export	exportDatabaseToWriter	(Context,Writer)	void
org.openintents.safe	CategoryList	backupToFile	(String)	void
org.openintents.safe	PRNGFixes	generateSeed	()	byte[]
org.openintents.safe	Backup	write	(String,OutputStream)	boolean
org.openintents.safe	Restore	restoreFromFile	(String)	void

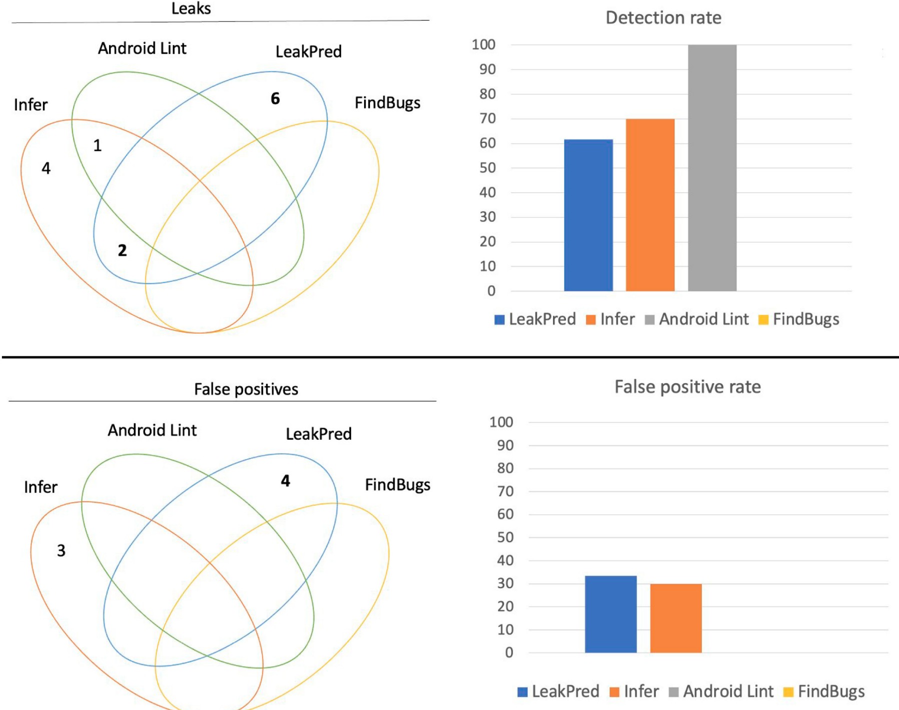

Figure 5. Components with leaks, detection rates, and false positives in textitOI Safe application.

It is worth mentioning that the *LeakPred* approach does not identify resource leaks in an intermediate class that inherits the original resource class, and this application had two resource leaks that were in this situation. The resource leaks were from the *InputStreamData* class, which inherited from the *java.io.InputStream* class, making it impossible to identify this resource leak using the approach *LeakPred*. These 2 resource leaks were counted in the 13 that could be identified.

With regard to false positives, the *LeakPred* approach had the highest percentage, with 33.33%, followed by the *Infer* method with 30%. The *Android Lint* and *FindBugs* methods did not report any false positives. However, the *FindBugs* method did not find any of the six resource leaks that could be identified in this application.

4.3. Results: AnyMemo Application

The 12 leaked components detected by the methods in the *AnyMemo* application are presented in Table 6. For a better understanding of how many leaks were identified by each method, Figure 6 is shown. In this figure, it can be seen that *Infer* identified two leaks and had a false positive. Still, in Figure 6, the detection rates of leaks and false positives are shown, in which it is highlighted that the *LeakPred* approach had the best coverage, identifying 100% of the leaks that the approach could identify. As far as false positives go, 7 out of 18 were in test files. Therefore, an improvement in the approach would be to ignore test files during code analysis.

The *Infer* method had the second best coverage with 50%. The *Android Lint* method, on the other hand, did not have any leaks that it could identify in this application, and the *FindBugs* method had two possible leaks to be identified, but it did not identify any of them and had a false positive (100%).

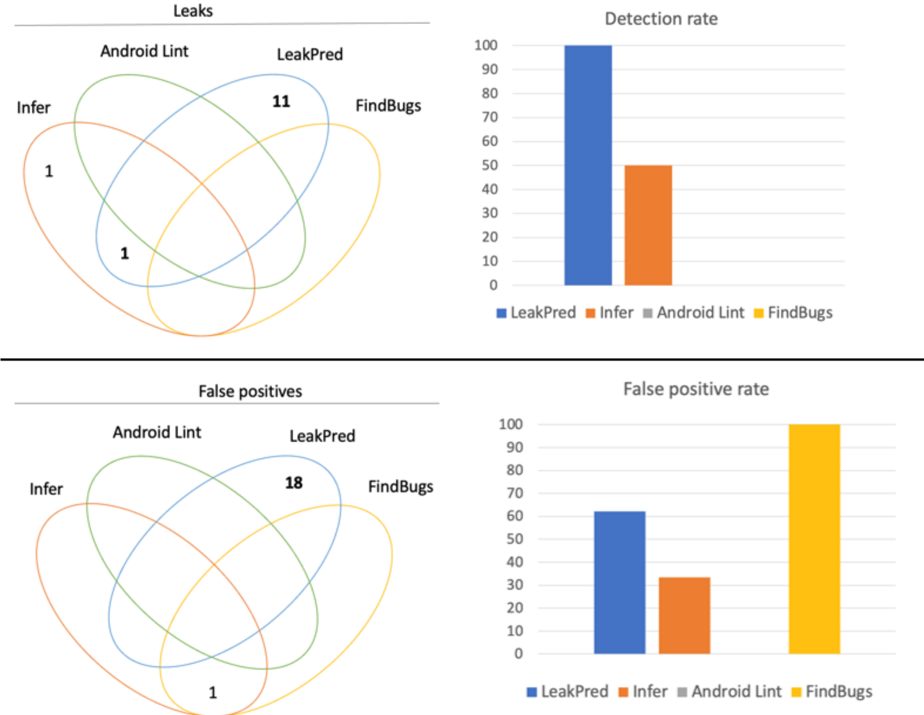

Figure 6. Components with leaks, detection rates, and false positives in the *AnyMemo* application.

Table 6. Components with resource leaks in the *AnyMemo* application.

Package	Class	Component	Parameters	Return
org.liberty.android.fantastischmemo.converter	Supermemo2008XMLImporter	convert	(String,String)	void
org.liberty.android.fantastischmemo.converter	MnemosyneXMLImporter	convert	(String,String)	void
org.liberty.android.fantastischmemo.converter	CSVExporter	convert	(String,String)	void
org.liberty.android.fantastischmemo.converter	SupermemoXMLImporter	convert	(String,String)	void
org.liberty.android.fantastischmemo.converter	Mnemosyne2CardsImporter	xmlToCards	(File)	List<Card>
org.liberty.android.fantastischmemo.converter	QATxtImporter	convert	(String,String)	void
org.liberty.android.fantastischmemo.converter	Mnemosyne2CardsExporter	createMetadata	(String,File)	void
org.liberty.android.fantastischmemo.tts	AnyMemoTTSImpl	stop	()	void
org.liberty.android.fantastischmemo.receiver	SetAlarmReceiver	cancelNotificationAlarm	(Context)	void
org.liberty.android.fantastischmemo.receiver	SetAlarmReceiver	cancelWidgetUpdateAlarm	(Context)	void
org.liberty.android.fantastischmemo.utils	AMZipUtils	zipDirectory	(File,String,ZipOutputStream)	void
org.liberty.android.fantastischmemo.ui	CardImageGetter	getDrawable	(String)	Drawable

4.4. Results: Avare Application

Table 7 shows the 22 resource leaks found by the methods in the application *Avare*. Figure 7 shows the number of leaks identified by each method. The *LeakPred* approach identified 20 leaks and had 39 false positives. Also, in Figure 7, the detection rates of leaks and false positives are shown, and it is observed that the *LeakPred* approach achieved the best coverage, 90.91%, as well as the highest percentage of false positives, 62.10%. Next is the *FindBugs* method with a coverage of 11.11% and a false positive rate of 50%.

The *Infer* method could identify 11 leaks but did not identify any of them, and the *Android Lint* method had no leaks that could be identified in this application. Regarding the false positives of the *LeakPred* approach, it can be noted that 14 of the 39 were from resources in class-level variables and not from the component; in other words, generally, these resources are not closed in the component where they are used.

Table 7. Components with resource leaks in the *Avare* application.

Package	Class	Component	Parameters	Return
com.ds.avare.externalFlightPlan	SkvPlanParser	parse	(String,FileInputStream)	ExternalFlightPlan
com.ds.avare.utils	ZipFolder	zipFiles	(String,OutputStream)	boolean
com.ds.avare.utils	ZipFolder	unzipFiles	(String,InputStream)	boolean
com.ds.avare.utils	NetworkHelper	getVersionNetwork	(String)	String
com.ds.avare.utils	Helper	readFromFile	(String)	String
com.ds.avare.utils	Helper	writeFile	(String,String)	boolean
com.ds.avare.shapes	Layer	parse	(String,String)	void
com.ds.avare.shapes	ShapeFileShape	readFile	(String)	ArrayList<ShapeFileShape>
com.ds.avare	LocationActivity	onDestroy	()	void
com.ds.avare	ChecklistActivity	onPause	()	void
com.ds.avare	PlanActivity	onPause	()	void
com.ds.avare	WnbActivity	onPause	()	void
com.ds.avare	ToolsFragment$ImportTask	doInBackground	()	String
com.ds.avare	ToolsFragment$ExportTask	doInBackground	()	String
com.ds.avare.adapters	ChartAdapter$ViewTask	doInBackground	()	Boolean
com.ds.avare.network	Download$DownloadTask	copyInputStream	(InputStream,OutputStream)	void
com.ds.avare.network	Download$DownloadTask	run	()	void
com.ds.avare.adsb	AudibleTrafficAlerts	stopAudibleTrafficAlerts	()	void
com.ds.avare.gps	Gps	stop	()	void
com.ds.avare.instruments	FuelTimer	stop	()	void
com.ds.avare.instruments	UpTimer	stop	()	void
com.ds.avare	StorageService	destroy	()	void

4.5. Results: Seafile Application

Table 8 displays the 42 leaking components that the methods identified in the *Seafile* application. To distinguish how many components were identified by more than one method, Figure 8 is shown. It shows that the same leak was identified by the *LeakPred*, *Infer*, and *Android Lint* methods. The *LeakPred* approach had the best coverage, 85.37%, and the highest false positive rate, 40.68%. Right after this method is the *Infer* method, with 37.50% coverage and a false positive rate of 25%.

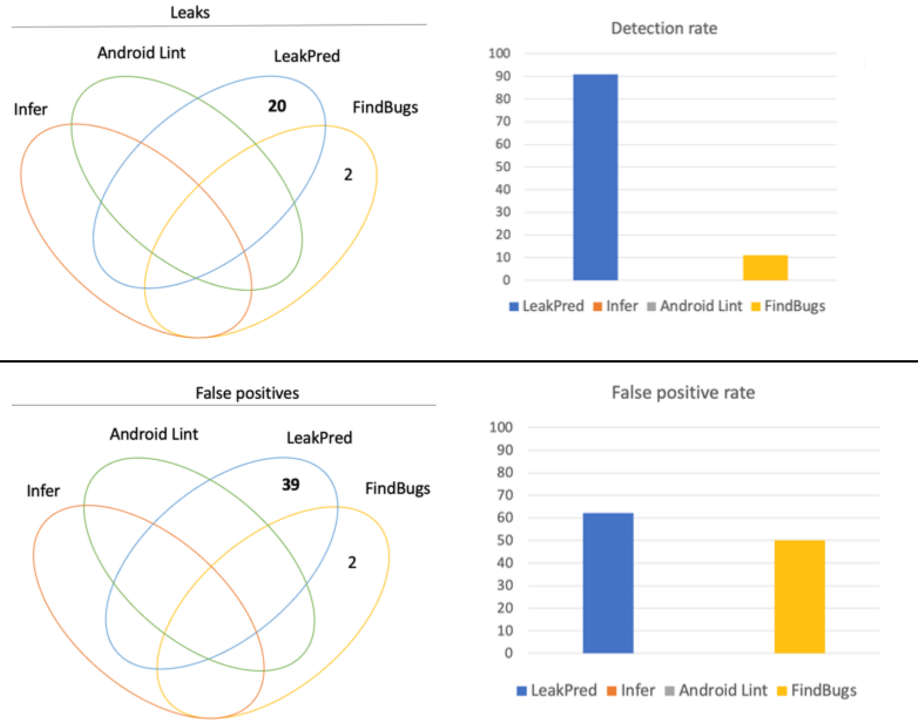

Figure 7. Components with leaks, detection rates, and false positives in the *Avare* application.

The *Android Lint* method did not have any false positives, but it only identified 2 out of 21 possible leaks (9.52%), and the *FindBugs* method did not have any leaking components that could be identified in this application.

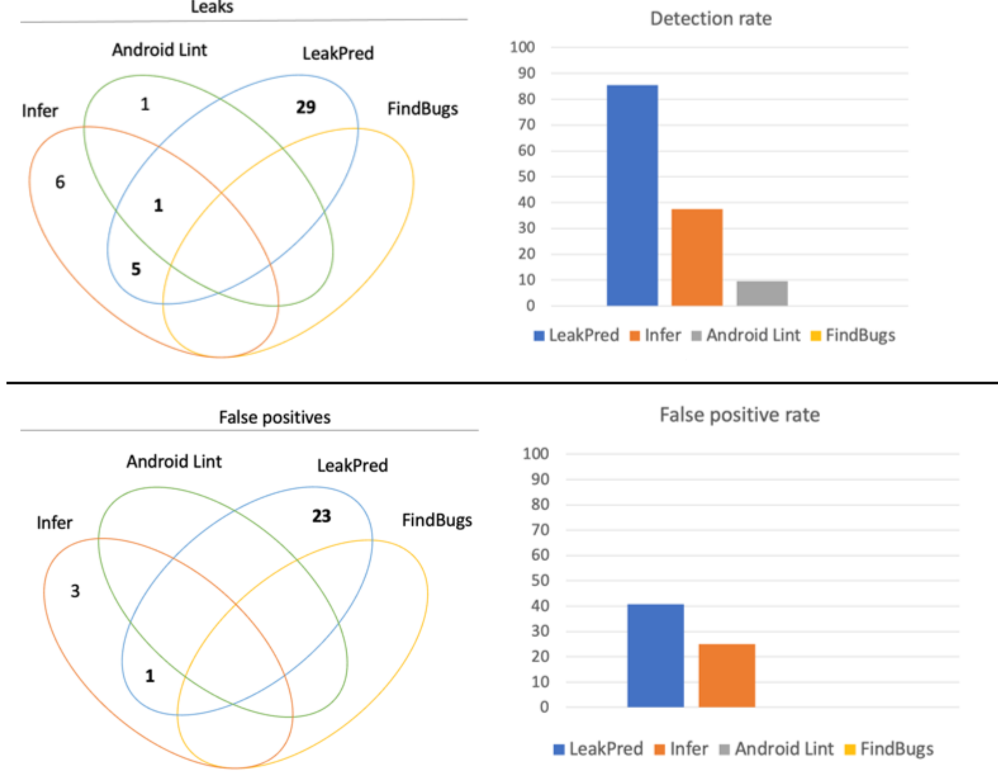

Figure 8. Components with leaks, detection rates, and false positives in *Seafile* application.

Table 8. Components with resource leaks in *Seafile* application.

Package	Class	Component	Parameters	Return
com.seafile.seadroid2	SeafConnection	realLogin	(String,String,boolean)	boolean
com.seafile.seadroid2.gallery	BitmapManager	cancelThreadDecoding	(Thread,ContentResolver)	void
com.seafile.seadroid2.gallery	MultipleImageSelectionActivity	onPause	()	void
com.seafile.seadroid2.gallery	ImageManager	isMediaScannerScanning	(ContentResolver)	boolean
com.seafile.seadroid2.gallery	ImageBlockManager$ImageBlock	recycle	()	void
com.seafile.seadroid2.notification	BaseNotificationProvider	notifyCompleted	(int,String,String)	void
com.seafile.seadroid2.notification	BaseNotificationProvider	notifyCompletedWithErrors	(int,String,String,int)	void
com.seafile.seadroid2.notification	BaseNotificationProvider	cancelNotification	()	void
com.seafile.seadroid2.data	DatabaseHelper	getFileCacheItem	(String,String,DataManager)	SeafCachedFile
com.seafile.seadroid2.data	DatabaseHelper	getFileCacheItems	(DataManager)	List<SeafCachedFile>
com.seafile.seadroid2.data	DatabaseHelper	getRepoDir	(Account,String)	String
com.seafile.seadroid2.data	DatabaseHelper	getCachedStarredFiles	(Account)	String
com.seafile.seadroid2.data	DatabaseHelper	repoDirExists	(Account,String)	boolean
com.seafile.seadroid2.data	DatabaseHelper	getCachedDirents	(String,String)	String
com.seafile.seadroid2.data	DatabaseHelper	getCachedDirentUsage	(String)	int
com.seafile.seadroid2.data	DatabaseHelper	getEnckey	(String)	Pair<String,String>
com.seafile.seadroid2.ui.activity	SeafilePathChooserActivity	onDestroy	()	void
com.seafile.seadroid2.ui.activity	ShareToSeafileActivity	getSharedFilePath	(Uri)	String
com.seafile.seadroid2.ui.activity	BrowserActivity$SAFLoadRemoteFileTask	doInBackground	()	File[]
com.seafile.seadroid2.util	Utils	copyFile	(File,File)	void
com.seafile.seadroid2.util	Utils	getFilenamefromUri	(Context,Uri)	String
com.seafile.seadroid2.util	Utils	getPath	(Context,Uri)	String
com.seafile.seadroid2.util	SeafileLog	writeLogtoFile	(String,String,String)	void
com.seafile.seadroid2.provider	SeafileProvider$Runnable	run	()	void
com.seafile.seadroid2.ui.dialog	SslConfirmDialog	onCreateDialog	(Bundle)	Dialog
com.seafile.seadroid2.ssl	CertsDBHelper	getCertificate	(String)	X509Certificate
com.seafile.seadroid2.avatar	AvatarDBHelper	hasAvatar	(Account)	boolean
com.seafile.seadroid2.avatar	AvatarDBHelper	getAvatarList	()	List<Avatar>
com.seafile.seadroid2.monitor	SeafileObserver	startWatching	()	void
com.seafile.seadroid2.monitor	FileMonitorService	onDestroy	()	void
com.seafile.seadroid2.monitor	MonitorDBHelper	getAutoUploadInfos	()	List<AutoUpdateInfo>
com.seafile.seadroid2.account	AccountDBHelper	getAccountList	(SQLiteDatabase)	List<Account>
com.seafile.seadroid2.account	AccountDBHelper	getServerInfo	(SQLiteDatabase,String)	ServerInfo
com.seafile.seadroid2.cameraupload	CameraSyncAdapter	onPerformSync	(Account,Bundle,String,ContentProviderClient,SyncResult)	void

Table 8. *Cont.*

Package	Class	Component	Parameters	Return
com.seafile.seadroid2.cameraupload	GalleryBucketUtils	getVideoBucketsBelowApi29	(Context)	List<Bucket>
com.seafile.seadroid2.cameraupload	GalleryBucketUtils	getImageBuckets	(Context)	List<Bucket>
com.seafile.seadroid2.cameraupload	CameraUploadDBHelper	isUploaded	(File)	boolean
com.seafile.seadroid2.gesturelock	LockPasswordUtils	checkPassword	(String)	boolean
com.seafile.seadroid2.gesturelock	LockPatternUtils	saveLockPattern	(List)	void
com.seafile.seadroid2.gesturelock	LockPatternUtils	checkPattern	(List)	boolean
com.seafile.seadroid2.account	AccountDBHelper	migrateAccounts	(Context)	void
com.seafile.seadroid2.editor	EditorActivity	readToString	(File)	String

5. Discussion

The *LeakPred* approach achieved the best coverage (mean of 84.56% and median of 85.37%) of leaks identified and with a coverage of 96.15% of the classes of leaks that could be identified in the five applications. This approach also had the highest rate of false positives (mean of 47.84% and median of 40.68%). Therefore, the need for further refinement to reduce false positives was noticed.

In the false positives of the *LeakPred* approach, we have three patterns that we can highlight. (1) When a class-level variable is being instantiated in a component and released in another, a possible way of improvement would be to find metrics that could represent this situation. (2) Test files were analyzed, which caused some false positives in these files. One solution would be to ignore the test files during code analysis. (3) It was observed that during file manipulation, a resource class is instantiated and passed as a parameter to instantiate another resource class and so on. Sometimes, it is necessary to close only one of them for the resource to be released correctly. A way to solve this problem would be to define a heuristic to deal with this phenomenon. Another way to decrease false positives would be to increase the amount of database leakage.

Table 9 shows a summary of metrics. The *Android Lint* method had the second highest coverage (mean of 62.15% and median of 76.92%) and the highest accuracy (median of 100%) of components with identified resource leaks and no false positives. However, it can only identify the resource class *android.database.Cursor* (1.92%) among the leak classes of the five applications in the study. The *Infer* method had the second highest accuracy (median of 95.16%), and the *LeakPred* method had the lowest accuracy (median of 81.25%).

The *LeakPred* approach can identify leaks in several resource classes. This is an advantage, since the fact that a method can identify several categories of leaks reduces the number of methods to be configured and executed, which can help in its use during the development of mobile applications and, consequently, in reducing costs and/or time throughout the project.

Table 9. Metric summary.

Tool	**OI Notepad**					**OI Safe**				
	TPR	**FDR**	**Accuracy**	**Precison**	**f1-Score**	**TPR**	**FDR**	**Accuracy**	**Precison**	**f1-Score**
LeakPred	85	37.04	79.03	62.96	72.34	61.54	33.33	92.91	66.67	64
Infer	52.63	9.09	83.61	90.91	66.67	70	30	95.16	70	70
Lint	76.92	0	94.55	100	86.96	100	0	100	100	100
Find Bugs	0	0	95.45	0	0	0	0	95	0	0
	AnyMemo					Avare				
	TPR	FDR	Accuracy	Precison	f1-score	TPR	FDR	Accuracy	Precison	f1-score
LeakPred	100	62.07	81.25	37.93	55	90.91	66.10	88.48	33.90	49.38
Infer	50	33.33	96.63	66.67	57.14	0.00	0.00	96.81	0.00	0
Lint	0	0	100	0	0	0	0.00	100	0	0
Find Bugs	0	100	96.55	0	0	11	50	94.89	50	18.18
	Seafile					Median				
	TPR	FDR	Accuracy	Precison	f1-score	TPR	FDR	Accuracy	Precison	f1-score
LeakPred	76.81	59.18	51.55	40.82	53.31	85	59.18	81.25	40.82	55
Infer	0	46.15	53.85	53.85	0	50	30	95.16	66.67	57.14
Lint	0	0	0	0	0	0	0	100	0	0
Find Bugs	0	0	18.18	0	0	0	0	95	0	0
	Mean									
	TPR	FDR	Accuracy	Precison	f1-score					
LeakPred	82.85	55.97	79.09	44.03	55.34					
Infer	34	27.90	87.52	51.44	36.86					
Lint	20	0	80	20	20					
Find Bugs	2.22	30	79.92	10	3.64					

6. Threats to Validity

This study has internal, building, and external threats that must be examined. This section details each one as follows:

Internal Validity: The sample of projects was not completely random, as they were randomly selected from the list of open-source applications presented in [8]. As this was a feasibility study, it is believed that this issue does not pose a significant threat.

Construction Validity: This feasibility study used five Android platform applications from different categories developed in the Java language. This study may not be representative for other categories of mobile applications. Commits containing resource leak fixes were identified using keywords, and to ensure that the commit was related to a leak, a manual validation step was included.

External Validity: To reduce this threat, five applications from four different categories were used, namely, productivity, maps and navigation, tools, and education. Likewise, the *LeakPred* approach was compared with more than one state-of-the-art method. In the future, it is intended to increase the number of applications of different categories. We also chose the median, as it provides a direct understanding of the central point of the data and is not as influenced by outliers as the mean.

7. Conclusions

In this work, a feasibility study for the *LeakPred* approach was presented, aiming to analyze it through a controlled experiment with respect to its effectiveness in identifying components with resource leaks compared to state-of-the-art methods. The results show the possibility of using the *LeakPred* approach to identify resource leaks in mobile applications, as it obtained the highest median coverage percentage of identified resource leaks with 85.37%, as well as having the highest percentage of leak class coverage, 96.15%. However, it had the lowest accuracy (median of 81.25%). There is also the possibility of refinement to reduce the number of false positives.

The results found provide a basic understanding of the feasibility of the *LeakPred* approach. As future work, a study could be carried out that increases the number of open-source mobile applications and uses company applications. Another possibility is to evaluate the efficiency of the methods. Another interesting future research would be removing the limitation of the *LeakPred* approach in identifying leaks in an intermediate class that inherits the original leak class and removing the parsing of the test files. Finally, there is the possibility to adapt the approach and carry out a feasibility study for the iOS platform.

Author Contributions: J.G.L. contributed to the conceptualization of the research, developing the methodology, analyzing the results, and the writing of the manuscript. R.G. and A.C.D.-N. supervised the research. All authors have read and agreed to the published version of the manuscript.

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

Data Availability Statement: We made the replication package available at https://bit.ly/3o8U1gU, and the *CompLeaks* database updated is found at https://bit.ly/3zLmgFj.

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

References

1. Statista. Number of Mobile Devices Worldwide 2020–2025. 2024. Available online: https://www.statista.com/statistics/245501/multiple-mobile-device-ownership-worldwide/ (accessed on 2 May 2024).
2. World. Current World Population. 2024. Available online: https://www.worldometers.info/world-population/ (accessed on 2 May 2024).
3. Zhang, H.; Wu, H.; Rountev, A. Automated test generation for detection of leaks in Android applications. In Proceedings of the 11th International Workshop on Automation of Software Test, Austin, TX, USA, 14–15 May 2016; pp. 64–70.
4. Android. Android Lint. 2024. Available online: https://developer.android.com/studio/write/lint (accessed on 2 May 2024).

5. Pugh, B.; Loskutov, A.; Lea, K. FindBugs. 2024. Available online: https://findbugs.sourceforge.net/bugDescriptions.html (accessed on 2 May 2024).
6. Lima, J.G.; Giusti, R.; Neto, A.C.D. Resource Leak Prediction in Android Applications Using Machine Learning. *Braz. J. Dev.* **2021**, *7*, 47820–47837.
7. Facebook. Infer. 2024. Available online: https://fbinfer.com/ (accessed on 2 May 2024).
8. 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. *Empir. Softw. Eng.* **2019**, *24*, 3435–3483. [CrossRef]
9. Android. Writing Custom Lint Rules. 2024. Available online: https://googlesamples.github.io/android-custom-lint-rules/ (accessed on 2 May 2024).
10. Rutar, N.; Almazan, C.B.; Foster, J.S. A comparison of bug finding tools for java. In Proceedings of the 15th International Symposium on Software Reliability Engineering, Bretagne, France, 2–5 November 2004; pp. 245–256.
11. Calcagno, C.; Distefano, D.; Dubreil, J.; Gabi, D.; Hooimeijer, P.; Luca, M.; O'Hearn, P.; Papakonstantinou, I.; Purbrick, J.; Rodriguez, D. Moving fast with software verification. In Proceedings of the NASA Formal Methods Symposium, Pasadena, CA, USA, 27–29 April 2015; Springer: Cham, Switzerland, 2015; pp. 3–11.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.