Noname manuscript No.
(will be inserted by the editor)

DroidLeaks: A Comprehensive Database of Resource Leaks in Android Apps

Yepang Liu · Jue Wang · Lili Wei · Chang Xu · Shing-Chi Cheung · Tianyong Wu · Jun Yan · Jian Zhang

Received: date / Accepted: date

Abstract Resource leaks in Android apps are pervasive. They can cause serious performance degradation and system crashes. In recent years, many resource leak detection techniques have been proposed to help Android developers correctly manage system resources. Yet, there exist no common databases of real-world bugs for effectively comparing such techniques to understand their strengths and limitations. This paper describes our effort towards constructing such a bug database named DroidLeaks. To extract real resource leak bugs, we mined 124,215 code revisions of 34 popular open-source Android apps. After automated filtering and manual validation, we successfully found 292 fixed resource leak bugs, which cover a diverse set of resource classes, from 32 analyzed apps. To understand these bugs, we conducted an empirical study, which revealed the characteristics of resource leaks in Android apps and common patterns of resource management mistakes made by developers. To further demonstrate the usefulness of our work, we evaluated eight resource leak detectors from both academia and industry on DroidLeaks and per-

Yepang Liu
Shenzhen Key Laboratory of Computational Intelligence, Department of Computer Science and Engineering, Southern University of Science and Technology, Shenzhen, China. E-mail: liuyp1@sustech.edu.cn

Jue Wang, Chang Xu
State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology, Nanjing University, Nanjing, China
E-mail: juewang591@gmail.com, changxu@nju.edu.cn

Lili Wei, Shing-Chi Cheung
Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong, China. E-mail: {lweiae, scc}@cse.ust.hk

Tianyong Wu, Jun Yan, Jian Zhang
Institute of Software, Chinese Academy of Sciences, University of Chinese Academy of Sciences, Beijing, China
E-mail: {wuty, yanjun, zj}@ios.ac.cn

formed an detailed analysis of their performance. We release DroidLeaks for public access to support future research.

Keywords Android apps · resource leak · mining code repository · bug database · fault pattern · tool evaluation

1 Introduction

Mobile applications (or apps for short) such as those running on the Android platform are gaining popularity in recent years. People rely on such apps for various daily activities such as work, socializing, and entertainment. Unlike PC software, mobile apps run on resource-constrained mobile devices and are required to consume computational resources (e.g., memory, battery power) more efficiently. However, many apps on the market fail to satisfy this non-functional requirement. They often do not properly release the acquired computational resources after use (Guo et al. 2013). Such software defects are called *resource leaks*. They can gradually deplete the finite computational resources in mobile devices at runtime, leading to severe performance degradation and system crashes.

Ensuring proper resource usage in a program is a non-trivial task for developers (Torlak and Chandra 2010). Over the years, researchers have proposed various techniques to help developers correctly manage resources used by their apps, including static analyzers (e.g., Guo et al. 2013, Liu et al. 2016b), verification (e.g., Vekris et al. 2012, Liu et al. 2014) and testing techniques (e.g., Yan et al. 2013, Wu et al. 2018). Besides, industrial tools such as Infer (Facebook 2018) and the built-in checkers in Android Studio (Google 2018b) can also help pinpoint resource leaks in the code of mobile apps.

Despite the tremendous efforts towards automated resource management and leak detection, there does not exist a widely-recognized database of real-world resource leak bugs in mobile apps. Such bug databases are essential as they can provide a reliable basis to evaluate and compare various resource management and leak detection techniques. Due to the lack of such bug databases, existing techniques such as Relda2 (Wu et al. 2016) can only be evaluated on a small set of open-source or commercial apps. The detected bugs were also rarely confirmed by the original developers. As a result, it is hard to (1) fully reproduce existing studies' results to assess the effectiveness of the proposed techniques in real settings and (2) quantitatively compare such techniques with a fair basis to understand their strengths and limitations. In addition, existing work only studied limited types of resource leaks in Android apps (e.g., those causing energy waste). To the best of our knowledge, there does not exist a comprehensive database of resource leak bugs in mobile apps.

In this work, we make an initial contribution towards benchmarking resource leak bugs for mobile apps and focus on the Android platform. To collect real resource leak bugs in Android apps, we investigated 34 diverse and popular open-source Android apps indexed by F-Droid (F-Droid 2018). A

straightforward approach for bug collection is to search these apps' issue tracking systems. However, in practice, software bugs could be fixed without being properly documented and this approach would miss many such bugs. In order to address the problem, we searched for bugs by examining the revision history of the apps. Our observation is that the patches to fix resource leak bugs (1) usually demonstrate patterns (e.g., developers often add code to invoke certain APIs to release the acquired resources) and (2) are eventually committed to the apps' code repository. Therefore, we can mine the apps' code repository for bug collection. To construct the bug database, we built a tool, which automatically mined 124,215 code revisions of the 34 apps. After automated filtering and manual validation, we successfully located 292 *fixed* resource leak bugs in 32 apps, of which only 14 (4.8% = 14/292) were documented in the corresponding apps' issue tracking system. We call this bug database DroidLeaks and collected the following data for each bug: (1) the type of the leaked system resource (in terms of Java classes such as `android.database.Cursor`), (2) the buggy code, (3) the bug-fixing patches, and (4) the bug report (if any).

To understand the characteristics of these bugs, we performed an empirical study on DroidLeaks and made several interesting observations. For example, developers can easily make resource management mistakes when the apps have complex lifecycles or frequently interact with the running environment. We also found three common patterns of resource management mistakes (e.g., API misuses and losing references to resource objects). Moreover, we observed that bugs in DroidLeaks are representative and comprehensive as they cover the types of resource leak bugs studied by the existing work (Guo et al. 2013, Liu et al. 2016b, Vekris et al. 2012, Wu et al. 2016), and additionally contain many more types. Such findings suggest that our work not only can provide practical programming guidance to Android developers (e.g., the bugs and patches in DroidLeaks can be used for training or educational purposes) but also can support follow-up research on developing or evaluating automated resource leak bug finding and patching techniques. As an example, we implemented a static checker to detect a common misuse of Android database APIs and helped developers find 17 previously-unknown resource leaks in their Android apps, of which 16 were fixed later (see Section 6.2).

To further demonstrate the usefulness of DroidLeaks, we experimentally evaluated eight existing resource leak detectors for Android apps using it. These detectors are freely accessible to Android developers. Some are research prototypes, while others are of industrial strength, e.g., Facebook Infer. All these detectors perform static analysis for bug detection. We did not evaluate dynamic analysis techniques due to the lack of test cases to run the Android apps. The results show that none of the existing detectors support detecting all types of resource leaks indexed by DroidLeaks. These detectors also suffer from high false negative or false positive rates, which would significantly hinder their adoption. To help improve the detectors, we provide a detailed analysis of their limitations with real-world bug examples in Section 5.4. In summary, we make three major contributions in this paper:

– We present DroidLeaks, a large database of real resource leak bugs in popular open-source Android apps, and describe its construction process in detail. DroidLeaks currently features 292 bugs covering 33 different resource classes. To the best of our knowledge, DroidLeaks is the first of its kind and we release it to facilitate future research (https://zenodo.org/record/2589909).
– We performed an empirical study of the bugs in DroidLeaks. The study revealed characteristics of resource leaks in Android apps and found common patterns of resource management mistakes made by Android developers.
– We evaluated eight existing resource leak detectors for Android apps with DroidLeaks and provide a detailed analysis of their strengths and weaknesses, which can shed light on future research to improve these detectors.

Paper organization. Section 2 introduces the preliminaries of Android apps and resource leaks. Section 3 presents our approach to constructing DroidLeaks. Section 4 discusses the characteristics of bugs in DroidLeaks. Section 5 evaluates existing resource leak detectors for Android apps. Section 6 discusses threats to validity, limitations of the work, the usefulness of DroidLeaks, and implications on future techniques. Section 7 reviews related work and Section 8 concludes this paper.

2 Background

Android is a Linux-based open-source mobile operating system. Android apps are mostly written in Java and compiled to Dalvik bytecode, which are then encapsulated into Android app package files (i.e., `.apk` files) for distribution and installation.

App components and event handlers. Android apps are event-driven programs. An app usually consists of four types of components: (1) *activities* contain graphical user interfaces (GUIs) for user interactions, (2) *services* run in background for long-running operations, (3) *broadcast receivers* respond to system-wide broadcast messages, and (4) *content providers* manage shared app data for queries. Each app component can define and register a set of *event handlers*, i.e., callback methods that will be invoked by the Android OS when certain events occur. Developers implement the main logic and functionalities of an app in these event handlers.

System resource management. In order to acquire system resources for computation, Android apps need to invoke designated resource-acquiring APIs provided by the Android SDK. When the computation completes, the apps should release the acquired resources by invoking the resource-releasing APIs. For example, wake lock is a critical system resource for power control on Android devices. Listing 1 on page 5 shows how an app can acquire a partial wake lock by calling the `WakeLock.acquire()` API (Line 3). The partial wake lock will keep CPU running to protect the critical computation from being disrupted by device sleeping. When the critical computation completes, the app releases the wake lock by calling the `WakeLock.release()` API (Line 5).

```
1. PowerManager pm = (PowerManager) getSystemService(POWER_SERVICE);
2. WakeLock wl = pm.newWakeLock(PARTIAL_WAKE_LOCK, "lockTag");
3. wl.acquire(); //acquire a wake lock
4. //performing critical computation when the wake lock is held
5. wl.release(); //release the wake lock
```

Listing 1: Example code for using wake locks

Resource leak. For correct resource management, developers should ensure that acquired resources are released on every possible program execution path, including exceptional ones. Particularly, for reference-counted resources (e.g., in Android, wake locks are by default reference-counted), each call to the resource-acquiring API must be balanced by an equal number of calls to the resource-releasing API.[1] Otherwise, the resources will be leaked (e.g., when developers forget Line 5 of Listing 1), which can cause undesirable consequences such as performance degradation and system crashes. In practice, resource management tasks are error-prone (Torlak and Chandra 2010, Wu et al. 2016). The complex and implicit control flows among Android event handlers further complicate the tasks, giving rise to various resource leak bugs (see Listing 2 and Listing 3 on page 14 for examples).

3 Collecting Resource Leak Bugs

This section presents our semi-automated approach for constructing the DroidLeaks bug database.

3.1 Selecting Open-Source App Subjects

To construct DroidLeaks, we started by selecting representative open-source Android apps for investigation. F-Droid (F-Droid 2018) is a well-known open-source Android app database. It indexed 2,146 apps of different maturity levels at our study time, of which 1,475 have an accessible source code repository and are hosted on GitHub, a leading open-source project hosting site.[2] To search for suitable app subjects, we defined the following four criteria: (1) a selected app should have more than 10,000 downloads on the market (the app is popular), (2) the app should also have a public issue tracking system (bugs are traceable), (3) the app's code repository should contain over 100 code revisions (the app is actively-maintained), and (4) the app should contain at least 1,000 lines of Java source code (the app is non-trivial). These four criteria were chosen to select non-trivial real-world apps and avoid toy-example projects. Our intuition is three-fold. First, if an app is frequently downloaded and has a large user base, users might have already encountered and reported various

[1] https://developer.android.com/reference/android/os/PowerManager.WakeLock
[2] https://github.com/

Table 1: Open-source app subjects and their resource leak bugs

App name	Category	Rating	Downloads	SLOC (Java)	# total revisions	# interesting revisions	# bugs
AnkiDroid	Education	4.5	1M – 5M	47.3K	8,303	223	28
AnySoftKeyboard	Tools	4.4	1M – 5M	25.2K	2,803	46	4
APG	Communication	4.4	100K – 500K	42.0K	4,366	69	10
BankDroid	Finance	4.1	100K – 500K	22.9K	1,202	5	6
Barcode Scanner	Shopping	4.1	100M – 500M	10.6K	3,219	43	3
BitCoin Wallet	Finance	4.0	1M – 5M	18.0K	2,442	52	4
CallMeter	Tools	4.3	1M – 5M	13.5K	2,263	27	10
ChatSecure	Communication	4.0	500K – 1M	37.2K	2,906	128	32
ConnectBot	Communication	4.6	1M – 5M	17.6K	1,349	22	5
CSipSimple	Communication	4.3	1M – 5M	49.0K	1,778	42	7
CycleStreets	Travel & Local	3.7	50K – 100K	18.8K	1,269	18	1
c:geo	Entertainment	4.4	1M – 5M	52.2K	9,338	90	8
FBReader	Books & References	4.5	10M – 50M	70.9K	9,005	76	8
Google Authenticator	Tools	4.3	10M – 50M	3.3K	179	1	5
Hacker News Reader	News & Magazines	4.4	50K – 100K	4.0K	296	2	4
IRCCloud	Communication	4.2	50K – 100K	35.3K	1,866	136	11
K-9 Mail	Communication	4.2	5M – 10M	78.5K	6,132	98	31
OI File Manager	Productivity	4.2	5M – 10M	6.9K	399	9	0
Open GPS Tracker	Travel & Local	4.1	500K – 1M	12.3K	1,096	42	2
Osmand	Maps & Navigation	4.2	5M – 10M	137.7K	29,336	134	13
OsmDroid	Maps & Navigation	3.9	50K – 100K	18.4K	1,881	34	2
OSMTracker	Travel & Local	4.3	100K – 500K	5.9K	400	14	4
ownCloud	Productivity	3.6	100K – 500K	31.6K	4,541	82	8
Quran for Android	Books & References	4.7	10M – 50M	21.7K	1,560	47	19
SipDroid	Communication	4.0	1M – 5M	24.5K	293	7	1
SMSDroid	Communication	3.9	1M – 5M	4.7K	813	14	2
SureSpot	Social	4.2	100K – 500K	41.0K	1,572	77	11
Terminal Emulator	Tools	4.4	10M – 50M	11.7K	1,035	11	2
Transdroid	Tools	4.3	100K – 500K	23.5K	427	8	2
Ushahidi	Communication	3.7	10K – 50K	35.7K	948	16	4
VLC-Android	Video Players & Editors	4.4	100M – 500M	18.1K	3,481	60	14
WebSMS	Communication	4.4	100K – 500K	4.4K	1,648	10	0
WordPress	Social	4.2	5M – 10M	74.9K	14,805	162	29
Xabber	Communication	4.1	1M – 5M	38.2K	1,264	6	2
Total					124,215	1,811	292

Notes: (1) 1K = 1,000 and 1M = 1,000,000; (2) For app downloads, we only considered data from the Google Play store; (3) The data in the table were initially obtained in October, 2016 and were last updated in August, 2017; (4) The links to the apps' code repositories can be found in the appendix.

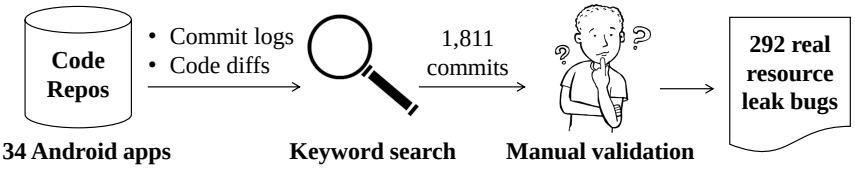

Fig. 1: Resource Leak Bug Collection Process

Table 2: Keywords for mining commit logs

leak	leakage	release	recycle	cancel
unload	unlock	unmount	unregister	close

Table 3: keywords for mining code diffs

`.close(`	`.release(`	`.removeUpdates(`
`.unlock(`	`.stop(`	`.abandonAudioFocus(`
`.cancel(`	`.disableNetwork(`	`.stopPreview(`
`.stopFaceDetection(`		`.unregisterListener(`

quality issues to the app's developers. The app is likely of a higher quality if developers have reacted to such user feedback. Second, if an app has a repository with a large number of revisions, the app is likely to be mature and more optimized than others. Third, if an app has at least thousands of lines of code, it is more likely that the app might have encountered performance issues than those smaller-scale ones.

170 of the 1,475 apps hosted on GitHub satisfied the above constraints. From them we randomly selected 34 (20%) apps as the subjects for our study. Table 1 (page 6) provides their basic information, including: (1) the app name, (2) category, (3) user rating on the Google Play store (5.0 is the highest rating), (4) number of downloads on the Google Play store, (5) app size (lines of Java source code), and (6) number of code revisions. As we can see, our subjects are diverse (covering 14 different app categories), of different sizes (from 3.3 KLOC to 137.7 KLOC with an average of 36.3 KLOC), popular (with millions of downloads), and well-maintained (with 3,653 code revisions on average).

3.2 Keyword Search

In order to ensure the quality of DroidLeaks bug database, we decided only to include those bugs that have already been confirmed and fixed by developers based on a common-sense assumption: *if developers take actions to fix an issue, it is likely that the issue is worth fixing and the actions help improve app quality*. To collect fixed resource leak bugs, we mined the code repositories of the 34 app subjects. Fig. 1 illustrates the overall process, which is semi-automated and contains two major steps: (1) keyword search, and (2) manual validation. This subsection introduces the first step and the next subsection introduces the second step.

The purpose of keyword search is to find interesting code revisions (or commits) that contain fixes to resource leak bugs. A code repository may contain a large number of code revisions. When committing each code revision, developers usually provide a short natural language message to summarize their changes, a.k.a. the *commit log* or *commit message*. The version control systems (e.g., Git) can help compute and visualize the differences between a committed revision and its parent revision(s), a.k.a. the *code diff*. When making commits, developers may mention that they fixed certain resource leaks in the commit logs. Since fixing resource leaks typically requires adding code to invoke designated APIs to release resources, we defined two sets of keywords to search for interesting commit logs and code diffs, respectively. The keywords are listed in Table 2 (page 7) and Table 3 (page 7). The keywords in Table 3 are formulated from the existing work for Android resource leak detection (Wu et al. 2016), which provides a list of frequently-used resource acquiring and releasing APIs. The keywords in Table 2 are general natural language words related to resource management.[3] Such natural language keywords are also needed due to two reasons. First, there is no guarantee that the set of resource releasing APIs (Table 3) provided by existing work is complete. Second, developers may wrap the resource releasing API calls in self-defined methods and invoke them to release resources.

To search for interesting commit logs in an app's code repository, we first transformed all commit logs into a canonical form that contains only lower case letters and no punctuation marks. We then removed certain patterns of phrases, which accidentally include our keywords but are irrelevant to resource leak bugs, from each commit log. For instance, we removed the phrases that match these two regular expressions: "`release (v|ver)?[0-9]+(\.[0-9]+)*`" and "`close issue #?[0-9]+`" as phrases such as "release v1.0.1" and "close issue #168" frequently occur in commit logs.[4] Next, we split each processed commit log into a vector of words and stemmed each word into its root form. Stemming (Lovins 1968) is necessary because the natural language words may be in different forms. For example, the verb "release" may be in its gerund form "releasing" in certain commit logs and we need to stem it into its root from "releas". After stemming, we applied the stemmed form of the keywords in Table 2 for searching.

To search for interesting code diffs, we looked for those diffs that contain lines (1) starting with the "+" symbol (representing code additions), and (2) containing a keyword from Table 3 (for matching resource-releasing API calls).

With the above two searching steps, we obtained a set of code revisions that contain either interesting commit logs or interesting code diffs. Column 7 of Table 1 (page 6) lists the number of such code revisions we found for each of the 34 open-source app subjects.

[3] We do not claim the completeness of our keyword set. With the current keywords, we successfully located a large number of real resource leak bugs in the code repositories of 32 of our 34 app subjects, which are sufficient for our later studies.

[4] In our mining scripts, we defined 32 removal patterns after randomly sampling 1,000+ commit logs. We skip the details in this paper.

3.3 Manual Validation of the Collected Bugs

In total, keyword search located 1,811 interesting code revisions. We then carefully investigated each of them to check whether it fixes resource leaks by understanding the relevant code, the purpose of code changes, and referring to the relevant API specifications (Android API Guides 2018, Java API Specification 2018). During the checking, we also analyzed bug reports (if any), commit logs, and developer comments. The process involved four people. First, two authors performed independent checking of each code revision and discussed with each other to reach consensus once any disagreement occurred. The other two authors then further checked the results for consistency. With such manual validation, we successfully found 292 resource leak bugs from 171 code revisions (some code revisions fix multiple resource leaks). The remaining code revisions are irrelevant but retrieved because their commit logs accidentally contain our search keywords or their code diffs contain the addition of resource-releasing API calls for other purposes (e.g., for refactoring or when new code that uses and correctly manages resources is introduced). We observe that 70 of the 292 bugs were found due to our code diff analysis. For example, in WordPress revision `64d7687c23`,[5] which fixes a leak of database cursor, the commit log only mentions "fixing bugs for RC build", but the code diffs contain the added code "`cursor.close()`". This bug can be found by analyzing code diffs but cannot be found by commit log analysis as the message is too general. We also observe that 18 of the 292 bugs were found due to our commit log analysis. For example, in AnkiDroid revision `b27f423f73`, the code diffs contain the added code "`closeOpenedDeck()`" to close an opened database. The method will trigger a chain of method calls until reaching the call to the standard `SQLiteDatabase` closing API in `com.ichi2.anki.AnkiDb` class. This bug cannot be found by analyzing only code diffs, but the commit log of the revision mentions "close database properly to avoid errors". Therefore, the bug can be found by our commit log analysis. The remaining 204 of the 292 bugs can be found by either code diff analysis or commit log analysis.

The last column of Table 1 (page 6) lists the number of real resource leak bugs we found for each app subject. As we can see, 32 of the 34 randomly-selected open-source Android apps contain snapshots where resources are not properly released after use, which suggests the pervasiveness of resource leak bugs in real-world Android apps.[6] Then for each bug, we further collected the following data to construct DroidLeaks: (1) the buggy code, (2) the bug-fixing patch, and (3) the bug report if we can find it in the issue tracking system of the concerned app.

[5] For all open-source projects referenced in this paper, we provide the links to their code repository in Table 10 of the appendix (on page 48). The readers can find the discussed revisions in the code repository.

[6] Note that it is hard to confirm whether these snapshots had been released to market and affected users due to the lack of data. There is a possibility that the "fixes" of resource leak bugs found by our approach were actually committed to the code repositories to address the warnings generated by IDEs or issues noticed by developers themselves instead of patching observed bugs.

4 Characteristics of Collected Resource Leak Bugs

To understand the characteristics of the bugs in DroidLeaks, we conducted an empirical study. We aim to answer the following three research questions:

- **RQ1 (Resource type and consequence of leak):** *What types of system resources (in terms of Java classes) are leaked due to these bugs? What are the consequences of these resource leaks? Are the leaked resources specific to the Android platform?*
- **RQ2 (Resource leak extent):** *Did the developers completely forget to release the concerned system resources on all program execution paths or only forget to release the resources on certain program execution paths or exceptional paths? Does the concerned resource escape local context?*
- **RQ3 (Common fault patterns):** *Are there common patterns of faults made by developers, which can result in resource leaks?*

To answer these questions, we carefully studied each bug in DroidLeaks and examined the relevant code (e.g., patches) and data (e.g., bug reports), API specifications (Android API Guides 2018, Java API Specification 2018). This section reports our observations.

4.1 RQ1: Resource Type and Consequence of Leak

RQ1 aims to identify the resource classes involved in the resource leak bugs in Android apps and understand the consequences of the bugs. This subsection presents how we analyzed our dataset to investigate RQ1 and discusses our major findings.

4.1.1 Resource Types

To identify the concerned resource classes, we studied the code related to each bug in DroidLeaks. Overall, we found that the 292 bugs in DroidLeaks cover 33 different resource classes listed in Table 4 (page 11). As we can see from the table, 61.3% of the bugs (179 of 292) concern resource classes that are specific to the Android platform. For instance, the SQLite database is widely-used in Android apps and we found 143 bugs in DroidLeaks leaking SQLite database cursors (see Listing 6 on page 21 for examples). The remaining 113 bugs (38.7%) leak general Java platform resources, of which I/O streams account for the majority. It is not surprising that the percentage of Java platform resource leaks is high (nearly 40%) since the majority of Android apps are implemented in Java and can use various Java libraries to leverage system resources for computational purposes.

Table 5 (page 12) lists the types of resources studied by existing Android app resource leak analysis work. As the table shows, existing work only studied the resource leaks related to a limited number of resource classes (e.g.,

Table 4: Information of the resource leak bugs in DroidLeaks

Concerned Java class (consequence of leak)	# bugs	Example resource acquiring API	Example resource releasing API
Android platform resources			
android.database.Cursor (I)	143 (27)[1]	SQLiteDatabase.query()[2]	Cursor.close()
android.database.sqlite.SQLiteDatabase (I)	13 (8)	SQLiteDatabase.openDatabase()	SQLiteDatabase.close()
android.os.PowerManager.WakeLock (II)	8 (8)	WakeLock.acquire()	WakeLock.release()
android.media.MediaPlayer (I)	5 (5)	MediaPlayer.start()	MediaPlayer.stop()
android.net.wifi.WifiManager.WifiLock (II)	2 (2)	WifiLock.acquire()	WifiLock.release()
android.location.LocationListener (II)	2 (2)	LocationManager.requestLocationUpdates()	LocationManager.removeUpdates()
android.net.http.AndroidHttpClient (I)	2 (0)	AndroidHttpClient.newInstance()	AndroidHttpClient.close()
android.view.MotionEvent (I)	1 (1)	MotionEvent.obtain()	MotionEvent.recycle()
android.os.ParcelFileDescriptor (I)	1 (1)	ParcelFileDescriptor.open()	ParcelFileDescriptor.close()
android.os.Parcel (I)	1 (0)	Parcel.obtain()	Parcel.recycle()
android.hardware.Camera (III)	1 (1)	Camera.open()	Camera.release()
General Java platform resources			
java.io.InputStream (I)	32 (26)	InputStream.<init>()[3]	InputStream.close()
java.io.FileInputStream (I)	12 (3)	FileInputStream.<init>()	FileInputStream.close()
java.io.FileOutputStream (I)	10 (2)	FileOutputStream.<init>()	FileOutputStream.close()
java.io.BufferedReader (I)	9 (2)	BufferedReader.<init>()	BufferedReader.close()
java.io.FilterOutputStream (I)	9 (2)	FilterOutputStream.<init>()	FilterOutputStream.close()
java.io.OutputStream (I)	6 (5)	OutputStream.<init>()	OutputStream.close()
java.io.FilterInputStream (I)	5 (0)	FilterInputStream.<init>()	FilterInputStream.close()
org.apache.http.impl.client.DefaultHttpClient (I)	4 (1)	DefaultHttpClient.<init>()	DefaultHttpClient.close()
java.io.BufferedOutputStream (I)	3 (0)	BufferedOutputStream.<init>()	BufferedOutputStream.close()
java.util.concurrent.Semaphore (III)	3 (3)	Semaphore.acquire()	Semaphore.release()
java.io.BufferedWriter (I)	2 (0)	BufferedWriter.<init>()	BufferedWriter.close()
java.io.ByteArrayOutputStream (I)	2 (0)	ByteArrayOutputStream.<init>()	ByteArrayOutputStream.close()
java.io.OutputStreamWriter (I)	2 (0)	OutputStreamWriter.<init>()	OutputStreamWriter.close()
java.net.Socket (I)	2 (1)	Socket.<init>()	Socket.close()
java.util.Scanner (I)	2 (0)	Scanner.<init>()	Scanner.close()
java.io.ObjectInputStream (I)	2 (0)	ObjectInputStream.<init>()	ObjectInputStream.close()
java.io.ObjectOutputStream (I)	2 (0)	ObjectOutputStream.<init>()	ObjectOutputStream.close()
java.io.PipedOutputStream (I)	2 (2)	PipedOutputStream.<init>()	PipedOutputStream.close()
java.io.DataOutputStream (I)	1 (0)	DataOutputStream.<init>()	DataOutputStream.close()
java.io.InputStreamReader (I)	1 (1)	InputStreamReader.<init>()	InputStreamReader.close()
java.util.Formatter (I)	1 (0)	Formatter.<init>()	Formatter.close()
java.util.logging.FileHandler (I)	1 (1)	FileHandler.<init>()	FileHandler.close()

[1] We report the number of cases where the concerned resource escapes the local context in the bracket (see Section 4.2. [2] Due to limited space, for each API example, we only provide a simple class name and a method name. Complete API information, including fully qualified class names and method parameter types, can be uniquely found in Android API Guides. [3] We denote public constructors of resource classes as "<init>()".

Table 5: Types of resources studied by existing work on Android app resource leak analysis

Concerned Java class	Pathak et al. 2012	Vekris et al. 2012	Yan et al. 2013	Liu et al. 2014	Wu et al. 2016	Wu et al. 2018	Banerjee et al. 2018
android.database.Cursor			✓				
android.os.PowerManager.WakeLock	✓	✓		✓	✓		✓
android.media.MediaPlayer					✓		
android.net.wifi.WifiManager.WifiLock					✓		✓
android.location.LocationListener	✓			✓	✓		✓
android.hardware.Camera	✓				✓		✓
android.hardware.Sensor	✓				✓	✓	✓
android.media.MediaRecorder	✓						✓
android.graphics.Bitmap			✓				
android.os.Binder			✓				
android.bluetooth.BluetoothAdapter							✓
android.media.AudioManager					✓		✓
android.os.Vibrator					✓		
java.lang.Thread			✓				

Underlined resource classes are not covered in the current version of DroidLeaks.

Wu et al. 2016 studied leaks related to eight resource classes). DroidLeaks contains resource leak bugs related to 33 different resource classes, 11 of which are specific to the Android platform. As a comparison, none of the existing work studied leaks related to such a large number of diversified resource classes. Therefore, we can see that DroidLeaks is large-scale. It can serve as a benchmark to evaluate resource leak detection techniques as we will show in Section 5. However, since DroidLeaks is constructed by analyzing the repositories of 34 open-source Android apps, it does not contain leaks related to all possible resource classes. For example, three resource classes studied by multiple pieces of existing work (i.e., `android.hardware.sensor`, `android.media.MediaRecorder`, and `android.media.AudioManager`) are not covered by the current version of DroidLeaks. In the future, we will analyze more Android apps and further improve the completeness and diversity of the resource leak bugs in DroidLeaks.

4.1.2 Consequence of Resource Leaks

Resource leaks are generally considered as non-functional issues that do not cause immediate fail-stop consequences such as app crashes. To understand the consequences of the bugs in DroidLeaks, we studied various data sources including the bug reports, commit logs of the bug-fixing revisions, developers' comments in the code, and API specifications (Android API Guides 2018, Java API Specifications 2018). We observed three types of major consequences:

– Most of the resource leak bugs (276 of 292, marked with "I" in Table 4) mainly lead to resource occupation and memory waste, which can gradually slow down the whole system. For example, an `android.database.Cursor` object has native file handles behind it because the SQLite database uses indexed files for providing query functions. Forgetting to close a cursor object

will prevent its associated file handles from being released as well as causing memory waste. While the consequence of a single instance of such resource leak bugs may not seem serious, in cases where certain operations that leak resources are repeatedly performed or apps run on low-end devices, users can experience obvious system slowdowns or even crashes (e.g., due to the out-of-memory exceptions). For example, after fixing multiple resource leak bugs in revision `25256904da`, the developers of AnkiDroid left the following commit log:

> *"Surround all cursor statements with try...finally to make sure that the cursor is closed at all times, even when an unexpected exception occurs. If a cursor is not closed, java will throw an exception, but it is silently (well, it can be seen in the log) discarded and does not cause a force close. However, it does cause a noticeable slow-down in the execution."*

– The second common consequence is energy waste, concerning 12 bugs in DroidLeaks (marked with "II" in Table 4). These bugs leak wake lock, Wi-Fi lock, and sensor-related resources, which are specific to the Android platform. For example, as we mentioned in Section 2, wake locks provide a mechanism to indicate that an app needs the device to stay awake for long-running operations (e.g., large file downloading). Calling the `acquire()` API on an `android.os.PowerManager.WakeLock` object will force the device to stay awake. While this interface is convenient, the Android API Guides also warns developers to carefully use wake locks:

  > *"Call release() when you are done and don't need the lock anymore. It is very important to do this as soon as possible to avoid running down the device's battery excessively".*

  Nonetheless, developers still make mistakes and we found nine wake lock leaks in DroidLeaks. Besides wake locks, DroidLeaks also contains resource leak bugs related to Wi-Fi locks, which are used to keep the Wi-Fi radio on for network communications, and sensor listeners, which are registered to obtain continuous updates from phone sensors. Such bugs can also lead to serious energy waste.

– The remaining four bugs (marked with "III" in Table 4 on page 11) concern exclusive resources: camera and semaphore (for restricting the number of concurrent threads). Forgetting to release them can affect app functionalities. For example, Android API Guides asks developers to release cameras when their apps finish using them to avoid affecting their own or other apps:

  > *"If your application does not properly release the camera, all subsequent attempts to access the camera, including those by your own application, will fail and may cause your or other applications to be shut down".*

  Similarly, forgetting to release semaphores can block acquiring threads and the corresponding computational tasks, leading to unexpected app behavior or even crashes in case apps stop responding to user interactions (Android ANR Errors 2018).

```
//FBReader revision 7907a9a13b
1.  public class LibraryService extends Service {
2.    public void onCreate() {
3.      ... //some set up work and open database
4.      myDatabase = SQLiteBooksDatabase.Instance(…);
5.    }
6.    public void onDestroy() {
7.      ... //some tear down work
8. +    myDatabase.close();
9.    }
10.  }
```

Listing 2: A resource leak involving complex app component lifecycle

4.1.3 Typical Resource Leak Examples

In the following, we discuss recurring examples of resource leaks in Android apps to ease understanding.

Complex app component lifecycle. As we mentioned in Section 2, Android apps consist of four types of app components. Each app component is required to follow a prescribed lifecycle that defines how this component is created, used, and finally destroyed. At runtime, the lifecycle event handlers (i.e., callback methods) defined in an app component will be invoked by the Android OS when the component enters certain lifecycle stages after user interactions. For instance, when a background service component is started, its `onCreate()` handler will be invoked. When the service finishes its allocated computational task, it will be destroyed and the `onDestroy()` handler will be invoked. Typically, when starting a service component, certain system resources need to be acquired for later computation. Listing 2 gives an example. The `LibraryService` component of the E-book reading app FBReader opens a database connection when it is launched (Line 4). The acquired resources should be released when the service is destroyed. However, since that the resources are acquired and released in different lifecycle stages (i.e., the acquiring and releasing operations are in different callback methods), developers can easily forget to release the resources properly. In DroidLeaks, we observed 14 such bugs and they are all leaks of Android-specific resources, including `SQLiteDatabase`, `WakeLock`, `MediaPlayer`, `LocationListener`. To fix the bug in FBReader, developers added the database closing statement in the `onDestroy()` handler (Line 8) in revision `7907a9a13b`.[7]

Complex GUI widget lifecycle. Besides the four types of top-level app components, the complex lifecycles of interactive GUI widgets also make resource management difficult. For example, the `SurfaceView` class is popularly used to create a window inside a view hierarchy that can be rendered by a secondary thread (so as not to block the main thread of an app). When using `SurfaceView`, developers need to carefully deal with the lifecycle changes of

[7]The code in all listings in this article has been simplified for readability. The readers can refer to the corresponding code repositories, whose URLs are provided in Table 10 in the appendix, to check the original code via the Git commit hash.

```
//IRCCloud revision d7a441e3a6
1.  public class ImageViewerActivity {
2.    private MediaPlayer player;
3.    private void loadVideo(String url) {
4.      player = new MediaPlayer();
5.      final SurfaceView v = (SufaceView) findViewById(…);
6.      v.getHolder.addCallback(new SufaceHolder.Callback() {
7.        public void surfaceCreated(…) {
8.          ...
9.          player.prepare();
10.         player.start();
11.       }
12.       public void surfaceDestroyed(…) {
13. +       if(player != null) {
14. +         player.stop();
15. +         player.release();
16. +       }
17.       }
18.     }
19.   }
20.   public void onDestroy() {
21.     super.onDestroy();
22.     if(player != null) {
23.       player.release();
24.     }
25.   }
26. }
```

Listing 3: A resource leak involving complex widget lifecycle

a `SurfaceView` instance and that of its enclosing activity component. Unfortunately, this is a non-trivial task and developers can easily make mistakes. Listing 3 provides an example. The app IRCCloud, a group chatting app, contains an activity `ImageViewerActivity` to view images and videos. The activity class uses `SurfaceView` to implement a floating window for playing videos. When the floating window pops up, the underlying media player is started (Lines 9–10) to play video. When users quit the activity (e.g., by clicking the return button), the media player is released (Lines 22–24). However, if users switch to another app or screen and put the `ImageViewerActivity` on pause at background, the video player would not be properly stopped and its associated resources cannot be released in a timely fashion (in such cases, the `onDestroy()` callback will not be invoked because the activity is still alive). To fix the resource leak, IRCCloud developers added the resource releasing code to the `surfaceDestroyed()` callback (Lines 13–16), which will be invoked during app or screen switching, in revision `d7a441e3a6`.

Environment interplay. Besides handling user inputs, Android apps also need to frequently react to environmental changes (e.g., changes of user location) to provide context-aware services. Similar to user inputs, environmental conditions are hard to predict. Developers can make resource management mistakes when handling environmental changes. Listing 4 gives an example. The app CSipSimple, an Internet calling app, uses a reference-counted wake

```
//CSipSimple revision da248d1132, in SipService class
1.  protected void onChanged(String type, boolean connected) {
2.    if(networkConnected()) {
3.      if(mTask != null){
4.        mTask.cancel();
5. +      sipWakeLock.release();
6.      }
7.      mTask = new MyTimerTask(type, connected);
8.      if(mTimer == null) mTimer = new Timer();
9.      mTimer.schedule(mTask, 2 * 1000L);
10.     sipWakeLock.acquire();
11.   }
12.  }
```

Listing 4: A resource leak involving environment interplay

lock to keep device awake for phone calls. The background service `SipService` monitors the network status by registering the `ServiceDeviceStateReceiver` broadcast receiver, which actively listens to network state changes. Whenever the network connection is restored, the broadcast receiver will invoke the `onChanged()` callback defined in the `SipService` class to acquire a wake lock for performing computational tasks (Lines 7–10) and cancel the existing task if any (Line 4). Under stable network conditions, there is usually just one acquisition of the wake lock and everything will work smoothly (in such cases, `onChanged()` will only be called once when the app connects to the network for the first time). However, when the network condition is poor (i.e., frequent disconnections and reconnections), `onChanged()` and its enclosing code for acquiring the wake lock will be executed many times, leading to repetitive acquisitions of the held wake lock. The consequence is that this reference-counted wake lock will not be properly closed and the device will stay awake indefinitely, causing huge energy waste. This is because to properly release a reference-counted wake lock, the number of calls to the releasing API should be equal to the number of calls to the corresponding acquiring API. Later, developers fixed the resource leak by releasing the held wake lock when canceling an existing timer task (Line 5) to balance the calls to the wake lock releasing and acquiring APIs in revision `da248d1132` with a commit log *"release wakelock if already hold when network status changes"*.

High level of concurrency. Android apps adopt a single thread model. All app components that run in the same process are instantiated in the app's main thread (a.k.a. UI thread), which is created by the Android OS when the app is launched. System calls to each component are dispatched from the main thread (Android Processes and Threads 2018). Hence, Android apps usually leverage various concurrent programming constructs such as `android.os.AsyncTask` and `java.lang.Thread` to perform intensive work like network communications and database queries in worker threads in order not to block the main thread (Lin et al. 2014), which would lead to poor runtime performance or even app not responding (ANR) errors (Android ANR Errors 2018). Such a high level of concurrency can easily cause resource leaks,

```
//K-9 Mail revision f1232a119a
1.  public void checkMail(…) {
2.    wakelock.acquire();
3.    put("checkMail", new Runnable() {
4.      public void run(){
5.        ...//check mail and sync to client
6.        putBackground("finalize sync", new Runnable() {
7.          public void run() {
8. +          if(wakelock != null) {
9. +            wakelock.release();
10. +         }
11.           ...//other listener update work
12.         }
13.       });
14.     }
15.   });
16. }
```

Listing 5: A resource leak involving multiple threads

especially when the resource acquiring and releasing operations are not in the same thread. Listing 5 gives an example. The app K-9 Mail, a widely-used email client, acquires a wake lock for keeping its device awake to check emails (Line 2). For synchronizing emails to the local folders, it creates a worker thread to communicate with the server (Lines 3–15). When the synchronization is finished, the worker thread further creates another thread to notify listeners (Lines 6–13). The acquired wake lock should be released after mail syncing. However, developers forgot this as there are multiple threads involved and the wake lock acquiring operation, which runs in the app's main thread, is quite far away (in Listing 5, the wake lock operations are close to each other because the code was simplified to ease understanding). They later figured out the mistake and released the wake lock properly (Lines 8–10) in revision `f1232a119a`.

From the above examples, we can see that detecting resource leaks in Android apps is a non-trivial task. *For dynamic analyses, how to effectively generate user interactions and simulate environment conditions to trigger resource leaking scenarios is a difficult task. For static analyses, how to handle the implicit control flows among various callback methods for inferring possible execution paths is a major challenge. Besides, static analyses also need to precisely model concurrency and perform points-to analysis, since resources may not be acquired and released in the same method or thread.*

Answer to RQ1: DroidLeaks *features a diverse set of resource leak bugs covering 33 different resource classes. Most bugs leak Android-specific system resources and can waste memory and cause system slowdown. Other consequences include battery drain and app crashes. Detecting resource leaks in Android apps can be difficult when the bugs involve complex app lifecycle, environment interplay, or concurrency.*

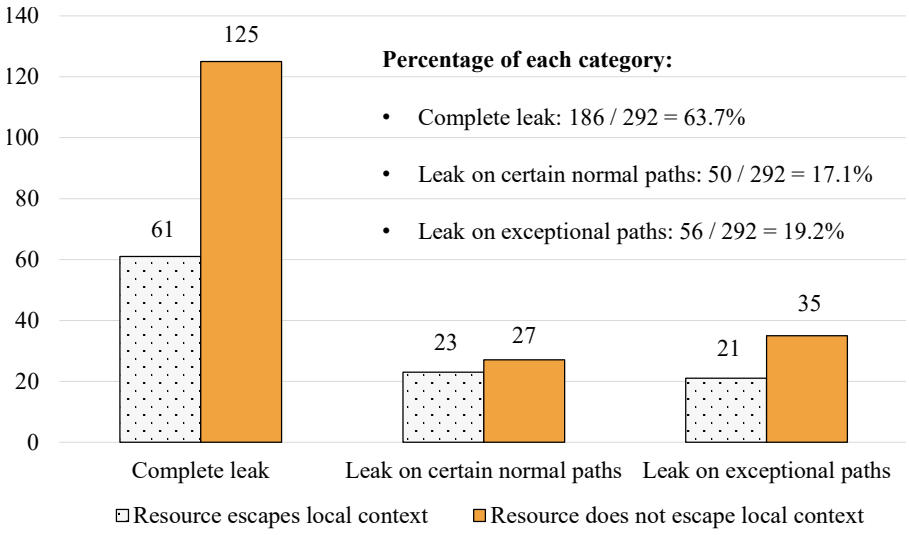

Fig. 2: Resource leak extent

4.2 RQ2: Resource Leak Extent

RQ2 aims to study whether developers forgot to release the concerned system resources on all program execution paths or only on certain execution paths. We also analyze whether the concerned resource escapes local context or not. In this subsection, we first present our analysis methodology and then discuss the main findings.

Resource leaks are essentially code omission faults, where developers forget to release used system resources on certain or even all possible program execution paths. Depending on what execution paths the resources are leaked on, we categorize the bugs in DroidLeaks into three categories:

- *Complete leak:* developers completely forget to release the system resources after use.
- *Leak on exceptional paths:* the system resources are properly released on normal execution paths, but fail to be released in case exceptions occur.
- *Leak on certain normal paths:* the system resources are released on some program paths during normal executions (i.e., without the occurrence of exceptions), but fail to be released on others. Note that this category includes resource leaks that occur under app-specific erroneous conditions, where no exceptions are thrown or handled but the concerned app enters a wrong internal state.[8]

Classification methodology. The classification was manually performed by studying each bug and its patch, which contains the call to the API that

[8]We observed only four such cases in DroidLeaks and therefore do not specifically discuss them in this paper. Interested readers can refer to our project website for details.

releases the concerned resource. If before adding the API call, the concerned resource is not released, we classify the bug as a "complete leak". If before adding the API call, the resource is released on some normal execution paths and the added call is to be invoked on the other normal execution paths, we classify the bug as a "leak on certain normal paths". If the added call is within a `catch` or `finally` block, we classify the bug as a "leak on exceptional path". We also analyzed whether a resource is local or escapes local context (used in other methods or threads). We considered the following cases. If the concerned resource variable is declared as a local variable and never used as a method call argument or by another thread, we consider the resource local. If the resource variable is declared as a local variable and used as an argument in a resource wrapping method call, we also consider the resource local (e.g., the constructor of `FilterInputStream` can take a `FileInputStream` object as an argument). If the resource variable is declared as an instance/class variable, we consider that the resource may escape local context. If the resource variable is declared as a local variable but used as a method call argument or by another thread, we consider that the resource escapes local context. The process involved four people. Firstly, two authors of this paper, who are experienced Java and Android developers, performed independent classifications for all bugs. Since the criteria are clear, there were disagreements only for five bugs after the first round of checking. The two authors then made further discussions and reached consensus. After that, the other two authors of the paper further checked the results for consistency. Fig. 2 on page 18 presents the result. By analyzing the data, we made two major observations.

First, it is surprising that the majority (63.7% = 186/292) of bugs in DroidLeaks caused system resources to be leaked on all program execution paths. The remaining 36.3% bugs caused system resources to be leaked on certain normal or exceptional paths. According to existing studies (Torlak and Chandra 2010), it is understandable that Java developers, even experienced ones, can easily fail to release all system resources along all possible exceptional paths. However, in DroidLeaks, we observed that resource leaks on exceptional paths only account for a minority (19.2% = 56/292). 44 resource leaks occurred due to improper handling of checked exceptions (e.g., forgetting to put resource releasing statements in a `finally` block that handles a `java.io.Exception`). 12 resource leaks occurred due to runtime exceptions such as `java.lang.IllegalStateException`. In comparison, leaks during normal executions are the majority (80.8% = (186 + 50)/292), which is unexpected.

Second, for 188 (64.4%) of the 292 bugs, the resource variable does not escape the local context. For example, as shown in Table 4 on page 11, for the majority (116) of the 143 leaks of `Cursor`, the resource variable does not escape the local context. For the remaining 104 (35.6%) bugs, the resource variable escapes the local context. Example resource classes include `WakeLock`, `MediaPlayer`, `WifiLock`, `LocationListener`, and `Camera`. For all bugs concerning these Android-specific resource classes, the resource variables escape the local context. Such acquired resources are typically used by multiple meth-

ods or threads. For instance, the `MediaPlayer` object in Listing 3 (page 15) is used by multiple methods and the `WakeLock` object in Listing 5 (page 17) is used by multiple threads. For general Java platform resources, a large percentage of `InputStream` (26/32), `OutputStream` (5/6), and `Semaphore` (3/3) variables also escape the local context. I/O stream variables are passed as arguments to various methods that perform tasks such as parsing, decoding, and encryption, while `Semaphore` variables are shared and used by multiple threads. *For these resources that escape the local context, detecting their leaks via static analysis is non-trivial, which requires inter-procedural analysis that handles concurrency.*

Answer to RQ2: *63.7% bugs in* DroidLeaks *completely leak system resources on all program execution paths. Only 19.2% resource leak bugs leak system resources on exceptional paths. For 64.4% of our studied resource leaks, the concerned resource variable does not the escape local context. However, we found that for all bugs that are related to several Android-specific resources including* `WakeLock`*,* `MediaPlayer`*,* `WifiLock`*,* `LocationListener`*,* `Camera`*, the resource variables escape the local context and could be used by multiple methods and threads. For general Java platform resources, a large percentage (over 81.3%) of* `InputStream`*,* `OutputStream`*, and* `Semaphore` *variables also escape the local context.*

4.3 RQ3: Common Fault Patterns

RQ3 studies the common resource management mistakes made by Android app developers. To answer RQ3, we analyzed all bugs in DroidLeaks and tried to understand the mistakes made by Android developers. From the discussions in Section 4.2, we can observe that in most cases, developers simply forgot to release system resources after use (complete leaks account for 63.7% cases). Nonetheless, we still observed three patterns of faults that recur across our studied apps. We discuss them with examples in the following.

API misuses. The Android platform provides over ten thousand public APIs (Felt et al. 2011) to developers to ease app development. In practice, it is generally impossible for developers to get familiar with the specification of each Android API before developing apps. Therefore, they can easily make mistakes when using unfamiliar APIs and resource leaks can arise in such cases. In DroidLeaks, 35 resource leak bugs, which affect 12 different apps, occurred due to API misuses. In particular, we observed three widely-used database APIs that Android developers often misuse.

– The first one is the `moveToFirst()` API defined in the `Cursor` class. Calling it will move the concerned database cursor to the first row if the cursor is not empty, or return `false` otherwise. Developers may think that only non-empty database cursors need to be closed (i.e., when this API returns `true`). Listing 6(a) (page 21) gives an example bug in IRCCloud. The app's

(a) IRCCloud revision 827b5e1b2b

```
1.    Cursor c = activity.getContentResolver().query(…);
2.    if(c != null && c.moveToFirst()) {
3.      ...
4.  -   c.close();
5.    } else { ... }
6.  + if(c != null) {
7.  +   c.close();
8.  + }
```

(b) CSipSimple revision 920c6c95d9

```
1.    Cursor c = getContentResolver().query(…);
2.    if(c != null && c.getCount() > 0) {
3.      ...
4.      c.close();
5.  + } else if(c != null) {
6.  +   c.close();
7.    }
```

(c) WordPress revision 57c0808aa4

```
1.    public class NotesAdapter extends CursorAdapter {
2.      public void reloadNotes() {
3.  -     swapCursor(mQuery.execute());
4.  +     changeCursor(mQuery.execute());
5.      }
6.    }
```

Listing 6: Leaks of database cursors due to API misuses

buggy version only closes the database cursor when `moveToFirst()` returns `true` (Lines 2–4) and would not close the cursor when it is empty. Later, developers realized this mistake and closed the database cursor properly (Lines 6–8) in revision `827b5e1b2b`.

– The second API `getCount()` is also defined in the `Cursor` class and it returns the number of rows in a cursor. Developers can use `getCount()` to check whether a database query returns an empty cursor (i.e., when `getCount()` returns 0). Similar to `moveToFirst()`, developers may think that only when there is at least one row returned by the query, the cursor needs to be closed. Listing 6(b) gives an example bug in CSipSimple. Originally, the buggy version only closes the database cursor when `getCount()` returns a positive number (Line 2). This would cause the leak of the cursor when no result is returned after querying the database. The developers later realized the problem and closed the database cursor correctly (Lines 5–6) in revision `920c6c95d9`.
– The third typical example is the `swapCursor()` API defined in the class `android.widget.CursorAdapter`, which can be used to adapt cursor data to a list view widget (a view that shows items in a vertically scrolling list).

```
//Owncloud revision dd35ee031b, in PreviewImageFragment class
1.   Bitmap result = null;
2. + InputStream is = null;
3.   try {
4.     File picture = new File(storagePath);
5.     if(picture != null) {
6. -     result = BitmapFactory.decodeStream(
7. -       new BufferedInputStream(new FileInputStream(picture)));
8. +       is = new BufferedInputStream(new FileInputStream(picture));
9. +       result = BitmapFactory.decodeStream(is);
10.    }
11. + } finally {
12. +   if(is != null) {
13. +     try {
14. +       is.close();
15. +     } catch(IOException e) {
16. +       Log.e("Unexpected exception...");
17. +     }
18. +   }
19. + }
```

Listing 7: A resource leak due to the lack of resource object reference

To replace the underlying database cursor associated with a `CursorAdapter` with a new one (e.g., after a new query), developers have two APIs to use: `swapCursor(Cursor newCursor)` or `changeCursor(Cursor newCursor)`. The only difference between the two APIs is that the former does not close the old cursor, but returns it, while the latter closes the old cursor. We found that developers may mistakenly thought that `swapCursor()` would also close the old cursor. For example, the developers of WordPress, a famous app for creating websites and blogs, made such a mistake. Listing 6(c) gives the concerned code. They originally used the `swapCursor()` API when reloading notes (Line 3). This would cause the leak of the old database cursor that is replaced. Later, they found the mistake and revised their code (Line 4) in revision `57c0808aa4` and left this commit log: "*Use changeCursor instead of swapCursor, so the old cursor is closed*".

Lacking references to resource objects is the second common pattern of faults made by developers. In Android apps, resource operations are performed by invoking certain APIs on resource objects. However, in DroidLeaks, we observe that developers often forget to create resource object reference variables and simply put resource operations in nested method calls. Listing 7 gives an example in OwnCloud, a private file sync and share app, where the input stream opening API call is nested in the stream decoding API call (Lines 6–7), and there is no variable holding the reference to the underlying input stream object. In such cases, developers can easily forget to release the acquired resources and Lint (Android Lint 2018a), the built-in static analyzer in Android Studio, would not report any resource leaks to warn them (see Section 5.4 for more detailed discussions why Lint would miss such

```
//AnkiDroid revision d095337329
1.    Cursor cur = null;
2.    try{
3.      cur = getDatabase().rawQuery(…);
4.      ...//some computation
5.  +   if(cur != null && !cur.isClosed()) {
6.  +     cur.close();
7.  +   }
8.      //a new query
9.      cur = getDatabase().rawQuery(…);
10.     ...//more computation
11.   } finally {
12.     if(cur != null && !cur.isClosed()) {
13.       cur.close();
14.     }
15.   }
```

Listing 8: A resource leak due to losing resource object reference

bugs). Such faults affected seven apps and caused 12 resource leak bugs in DroidLeaks.

Losing references to resource objects is the third common pattern of faults, which affected five apps and caused eight resource leak bugs. Listing 8 on page 23 gives a typical example in the app AnkiDroid, a popular flashcard app for education. As we can see from the code snippet, the app performs two queries consecutively to retrieve data from its database (Lines 3 and 9). The developers were aware that database cursors need to be closed after use and put the cursor closing code in a `finally` block (Lines 11–15). However, since there are two queries, two cursor objects are constructed, but the local variable `cur` only holds the reference to the second cursor object. The reference to the first cursor object is lost after requery (Line 9). The consequence is that the first database cursor is left unclosed, resulting in the leak of its associated resources, which would not be automatically recycled by the garbage collector (Torlak and Chandra 2010). Developers later fixed their mistake by releasing the leaked cursor (Lines 5–7) in revision `d095337329`. Besides database cursors, we also observed similar faults, where developers mistakenly override the variables that hold references to I/O streams.

Answer to RQ3: *We observed three common patterns of faults made by Android developers in* DroidLeaks*: (1) API misuses, (2) lacking references to resource objects, and (3) losing references to resource objects.*

5 Performance of Existing Resource Leak Detectors

As we discussed earlier, DroidLeaks can provide a common and reliable basis for evaluating and comparing existing resource leak detectors for Android apps.

To show such usefulness, in this section, we perform large-scale experiments to evaluate and compare the following eight static resource leak detectors, which are freely available to Android developers. The first six are general-purpose detectors, while the last two were specifically designed for finding the leaks of wake locks.

- **Android Lint** (Google 2018a) is a static code analysis tool for Android apps. It scans the source files (e.g., Java code files, resource and configuration files) of an Android app to identify micro-optimization opportunities (Linares-Vásquez et al. 2017) for improving the apps' correctness, security, performance, usability, accessibility, and internationalization.[9] It supports detecting various types of resource leaks as we will show shortly. Lint is built-in in Android Studio, the official IDE for Android app development.
- **Code Inspection** (JetBrains 2018) is a robust, fast, and flexible static source code analysis tool provided by the IntelliJ IDEA, a popular Java IDE with a large user base. It supports detecting various kinds of compiler errors, runtime errors, and code inefficiencies such as resource leaks. It also suggests corrections and improvements for developers to enhance the quality of their Java and Android apps. Since Android Studio is built on IntelliJ IDEA, Code Inspection is also freely accessible to Android developers.
- **FindBugs** (Hovemeyer and Pugh 2004) is a popular open-source static analysis tool for detecting bugs in Java programs. It operates on Java bytecode and performs efficient analysis to look for potential problems by matching the bytecode against a list of bug patterns,[10] some of which are related to resource leaks. Since Android apps are typically written in Java and first compiled to Java bytecode before being translated to the Dalvik bytecode, FindBugs can also help detect quality threats in Android apps' code.
- **PMD** (PMD 2018), similar to FindBugs, is another open-source and static analysis tool for Java programs. It uses rule-sets to define when a piece of source code is erroneous. PMD by default includes a set of built-in rules,[11] some of which describe resouce leaks, and analyzes Java source files to find common programming flaws. As Android apps are mostly Java programs, PMD is also popularly used by Android developers for app quality assurance.
- **Relda2** (Wu et al. 2016) is a light-weight static analysis tool for detecting resource leaks in Android apps. It directly analyzes the `.apk` file of an Android app for bug detection and focuses on Android-specific resources such as camera, media player, and sensors. It provides a general framework to support the analysis of various kinds of resources in a conservative way that identifies the resource releasing points as suggested by the Android API Guides. Relda2 can be configured to perform flow-sensitive or flow-

[9] http://tools.android.com/tips/lint-checks
[10] http://findbugs.sourceforge.net/bugDescriptions.html
[11] https://pmd.sourceforge.io/pmd-4.3.0/rules/index.html

insensitive analyses to adapt to different analysis precision and efficiency requirements.

- **Infer** (Facebook 2018) is a popular open-source static analysis tool provided by Facebook to detect potential bugs in Java and C/C++/Objective-C code. It can help developers intercept critical bugs before they ship their products to users and help prevent program crashes and poor performance. It takes information from the compilation process of the programs under analysis and translates the source files to its own intermediate language for detecting different patterns of bugs. Currently, for Android apps and Java programs, Infer detects and reports resource leaks and null pointer exceptions.
- **Elite** (Liu et al. 2016b) is a static analysis technique designed by us to detect the misuses of wake locks in Android apps. It takes an app's `.apk` file as input and systematically explores different executions of each Android app component that uses wake locks to locate the problematic program points, where wake locks are not needed but acquired, by performing an interprocedural data flow analysis. It also suggests the earliest program points to release wake locks by analyzing whether the uses of wake locks at different application states can bring users peceptible benefits.
- **Verifier** (Vekris et al. 2012) is a static verification technique proposed by Vekris et al. It analyzes an Android app's `.apk` file to verify the absence of the leaks of wake locks with respect to a set of resource management policies derived by studying the lifecycle of Android app components. The policies specify that at key program exit points, where an app component has finished computation, the component must be in a low energy state with all acquired wake locks released. Similar to Elite, Verifier also leverages data flow analysis for the policy checking.

5.1 Research Questions

Our experiments aim to answer two research questions:

- **RQ4 (Resource Class Coverage):** *What types of system resources (in terms of Java classes) does each resource leak detector support?*
- **RQ5 (Bug Detection Effectiveness):** *How does each resource leak detector perform in terms of bug detection rate and false alarm rate?* Here, *bug detection rate* evaluates to what extent each resource leak detector can successfully detect the bugs in DroidLeaks, whose resource classes are supported by it. *False alarm rate*, on the other hand, evaluates to what extent each detector reports false warnings. We will define the two evaluation metrics shortly in Section 5.2.

5.2 Experimental Setup

This subsection explains our experimental setup in detail. We first present our bug selection process. We then explain how we compiled the app subjects and ran the existing tools to evaluate their performance.

5.2.1 Bug Selection

DroidLeaks features a large set of resource leak bugs in Android apps. For our experiments, we selected a subset of bugs in DroidLeaks. We did not evaluate the above-mentioned eight detectors on all bugs due to two major reasons. First, as demonstrated by our findings in Section 4.3, some bugs in DroidLeaks were caused by the same patterns of faults, and hence there is no need to use all such bugs to evalute the eight detectors to understand their strengths and limitations. Second, compiling open-source Android apps is a labor-intensive process, especially for the versions that rely on specific libraries and do not have well-prepared build scripts or instructions. To select a comprehensive subset of bugs, we followed several criteria: (1) the subset should contain leaks of each type of system resource (see Section 4.1), (2) the subset should contain bugs from each of the 32 app subjects with resource leak bugs (see Table 1 on page 6), (3) the subset should contain bugs with all three different extents of resource leaks (see Section 4.2), and (4) the subset should contain resource leak bugs of each fault pattern (see Section 4.3). With these criteria, we selected 116 resource leak bugs from the whole set of 292 bugs in DroidLeaks. Table 6 (page 27) lists the number of bugs selected for each type of system resource. These bugs will be used to evaluate the eight detectors.

5.2.2 App Compilation

For each of the 116 bugs, we compiled the corresponding buggy version and bug-fixing version (i.e., patched version) of the concerned app into both Java bytecode and `.apk` files (different tools take different types of inputs). The compilation was done in Android Studio. If the source code of the apps contain Gradle build scripts, we simply used the scripts to compile the apps. For the other cases, we prepared the Gradle build scripts by ourselves and compiled the apps for the Android versions specified in the apps' configuration file, i.e., the `AndroidManifest.xml` file.

5.2.3 Tool Running & Result Analysis

Lint and Code Inspection are built-in in Android Studio. FindBugs and PMD provide Android Studio plugins. So, in the experiments, we ran these four detectors, namely Lint (built-in in Android Studio version 3.1.3)[12], Code In-

[12] https://developer.android.com/studio/releases/

Table 6: Resource leak bugs selected for our experiments

Resource class	# Bugs	Related Projects
`Cursor`	38	AnkiDroid, AnySoftKeyboard, APG, BankDroid, ChatSecure, CSipSimple, Google Authenticator, IRCCloud, Osmand, OSMTracker, Owncloud, SMSDroid, TransDroid, WordPress
`SQLiteDatabase`	3	AnySoftKeyboard, ConnectBot, FBReader
`WakeLock`	8	CallMeter, ConnectBot, CSipSimple, K-9 Mail, Open GPS Tracker, VLC-Android
`MediaPlayer`	3	IRCCloud, SureSpot
`WifiLock`	1	IRCCloud
`LocationListener`	2	OsmDroid, Ushahidi
`AndroidHttpClient`	2	Barcode Scanner
`MotionEvent`	1	Xabber
`ParcelFileDescriptor`	1	K-9 Mail
`Parcel`	1	c:geo
`Camera`	1	SipDroid
`InputStream`	9	K-9 Mail, SureSpot, Terminal Emulator
`FileInputStream`	1	CycleStreets
`FileOutputStream`	2	Quran for Android, Xabber
`BufferedReader`	1	SureSpot
`FilterOutputStream`	7	ChatSecure
`OutputStream`	4	K-9 Mail, SureSpot, Terminal Emulator
`FilterInputStream`	5	ChatSecure
`DefaultHttpClient`	4	BankDroid
`BufferedOutputStream`	1	Quran for Android
`Semaphore`	3	BitCoin Wallet, K-9 Mail, Ushahidi
`BufferedWriter`	1	VLC-Android
`ByteArrayOutputStream`	2	SureSpot
`OutputStreamWriter`	1	VLC-Android
`Socket`	2	IRCCloud, K-9 Mail
`Scanner`	2	c:geo
`ObjectInputStream`	2	Hacker News Reader
`ObjectOutputStream`	2	Hacker News Reader
`PipedOutputStream`	2	K-9 Mail
`DataOutputStream`	1	APG
`InputStreamReader`	1	FBReader
`Formatter`	1	BitCoin Wallet
`FileHandler`	1	Osmand
Total	116	32

Note: This table only provides the simple names of resource classes. Please refer to Table 4 on page 11 for the fully qualified names.

```
1. public class MainActivity extends AppCompatActivity {
2.  @Override
3.  protected void onCreate(Bundle savedInstanceState) {
4.    super.onCreate(savedInstanceState);
5.    PowerManager pm = (PowerManager) getSystemService(POWER_SERVICE);
6.    WakeLock wl = pm.newWakeLock(PARTIAL_WAKE_LOCK, "wakelock");
7.    wl.acquire();
8.  }
9. }
```

Listing 9: Example code in the test app

spection (built-in in Android Studio version 3.1.3), FindBugs (version 1.0.1)[13], PMD (version 6.0.1, integrated into Android Studio Plugin QAPlug-PMD version 1.4.0)[14], directly within Android Studio on a Macbook Pro with Intel Core i7 CPU @3.1 GHz and 16 GB RAM. The other four detectors are stand-alone ones with only command-line interfaces. Infer (version 0.15.0)[15] was also run on the Macbook Pro. Relda2, Elite and Verifier were run on a Linux server with 16 cores of Intel Xeon CPU @2.10GHz and 192GB RAM, running CentOS 7.3. These three tools are research prototypes without specific version numbers and we used their latest accessible copies for the experiments.

To answer RQ4, we manually constructed a test app by simply requesting system resources without releasing them. For example, in the code snippet in Listing 9, we make the test app acquire a wake lock in its main activity's `onCreate()` callback (Line 7), which will be invoked when the app is launched, without releasing it. In our test app, the main activity acquires all kinds of resources in Table 4 (page 11) but does not release any of them. In this way, we can test whether a tool supports detecting a certain type of resource leak or not. We ran the above-mentioned tools except the three ones from academia, whose supported resource classes are clearly reported in the corresponding research papers, to analyze this test app. If the tool detects a certain type of resource leak in our simple test app, we will further evaluate its bug detection effectiveness with real-world resource leaks of the type indexed by DroidLeaks. Otherwise, we conclude that the tool does not support detecting this type of resource leak. This is because if the tool cannot detect the simple case in our test app, it is unlikely to be able to detect other complex cases. Our experimental results are reported in Section 5.3.

After knowing the resource classes supported by each detector, we further selected the applicable bugs from our earlier selected 116 bugs to evaluate them. As we mentioned earlier, we aim to evaluate the *bug detection rate* and *false alarm rate* of the eight detectors. We define the two metrics in Equations (1) and (2), respectively. To evaluate the bug detection rate of a detector t, denoted $BDR(t)$, we leveraged the buggy app versions. Specifically, for

[13] https://plugins.jetbrains.com/plugin/3847-findbugs-idea/
[14] https://plugins.jetbrains.com/plugin/4596-qaplug--pmd/
[15] https://github.com/facebook/infer/releases/tag/v0.15.0

each selected bug that is applicable to evaluate t, we performed the following checking: when analyzing the corresponding buggy app version, if the tool t reports a warning that describes the leak of the concerned resource at the corresponding bug location, we consider this warning as a true one, i.e., the tool t successfully detects the bug. Otherwise, we consider that the tool misses the bug. After evaluating t with all applicable bugs, t's bug detection rate can be calculated by Equation (1). To evaluate the false alarm rate of each tool t, denoted $FAR(t)$, we leveraged the patched app versions. Specifically, for each selected bug that is applicable to evaluate t, we performed the following checking: when analyzing the corresponding patched app version, if the tool t reports a warning that describes the concerned resource leak bug (it should not since the bug is fixed), we consider this warning as a false alarm. Similar to calculating bug detection rate, after evaluating t with all applicable bugs, t's false alarm rate can be calculated by Equation (2). One can observe that during the experiments, we would ignore the warnings that are not related to the experimented bugs. This is due to the lack of ground truth to judge whether the warnings are true ones. Preparing such ground truth requires tremendous manual effort, which individual researchers like us cannot afford, as each tool could report hundreds of warnings when analyzing each of our app subjects.

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

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

5.3 RQ4: Resource Class Coverage

Table 7 (page 31) reports the supported types of system resources for the eight evaluated resource leak detectors. We mark a table cell with the "✓" symbol if the concerned resource leak detector supports the corresponding type of system resource. From the results, we can observe several interesting findings.

Finding 1: None of the existing detectors supports all 33 types of systems resources indexed by DroidLeaks. As we can see from Table 7, the six detectors have very different resource leak detection capabilities. Code Inspection supports the most (22 out of 33) types of system resources. FindBugs and Infer support 14 and 11 types, respectively. In comparison, Relda2 and Android Lint only support a few types. Unexpectedly, PMD does not support any type of system resource covered by DroidLeaks. We further checked the rule set of PMD (see Footnote [11] on page 24) and found out the reason. PMD indeed supports detecting the leaks of system resources in Java programs, including the `Connection`, `Statement`, and `ResultSet` classes in the `java.sql` package. However, since Android apps typically use the SQLite database APIs in the `android.database.sqlite` package, no resource leak

bugs indexed by DroidLeaks are related to the types of resources supported by PMD. Therefore, we believe that PMD is less useful in detecting resource leaks for Android apps comparing to the other detectors and we will not further evaluate it.

Finding 2: While we found that 29 of the 33 types of system resources are supported by at least one existing resource leak detector, the remaining four types, which are highlighted in Table 7, are not supported by any existing detector. Although there are only 10 (3.4% of 292) bugs related to the four types of system resources in DroidLeaks, the leaks of such resources can cause serious consequences. For example, as we discussed earlier in Section 4.1, the leak of the exclusive system resource `java.util.concurrent.Semaphore` can affect app functionality and lead to thread blocking, which could cause app crashes (Android ANR Errors 2018) and significantly affect user experience. It would be helpful if the widely-used detectors can support detecting such resource leaks.

Finding 3: While the majority (18 out of 22) of the general Java platform resources are supported by multiple resource leak detectors, only two (out of 11) types of Android platform resources are supported by multiple detectors. How to detect the leak of general Java platform resources has long been studied by researchers (Weimer and Necula 2004). This may explain that most general Java platform resources have multiple detectors. In comparison, we observe that there is still the lack of research and tool support for detecting Android-specific resource leaks. This is likely because Android, as a mobile computing platform, has much shorter history than the traditional computing platforms and the resources on the Android platform that are newly introduced have their special characteristics (e.g., the SQLite database cursor object is not reference-counted, but the wake lock object is reference-counted by default). We advocate that more efforts should be devoted into designing and developing useful tools for detecting the leaks of Android platform resources. We believe that the large number of real-world bugs in DroidLeaks can guide the design of such tools and help reliably evaluate them in the future.

Answer to RQ4: *Existing resource leak detectors, especially those designed for Android apps, only support detecting the leaks of limited types of system resources. We advocate that more efforts can be spent to make the existing detectors support detecting the leaks of more types of system resources.*

5.4 RQ5: Bug Detection Effectiveness

All our evaluated detectors finished analyzing each subject quickly. Android Lint, Code Inspection, and FindBugs could finish analyzing an app in a couple of seconds, while Facebook Infer, Relda2, Elite, and Verifier could finish in a

Table 7: Resource tables of existing resource leak detectors

Concerned Java class	Android Lint	Code Inspection	FindBugs	PMD	Relda2	Infer	Verifier	Elite
Android platform resources								
android.database.Cursor	✓	✓				✓		
android.database.sqlite.SQLiteDatabase		✓						
android.os.PowerManager.WakeLock					✓		✓	✓
android.media.MediaPlayer					✓			
android.net.wifi.WifiManager.WifiLock					✓			
android.location.LocationListener					✓			
android.net.http.AndroidHttpClient								
android.view.MotionEvent	✓							
android.os.ParcelFileDescriptor		✓						
android.os.Parcel	✓							
android.hardware.Camera					✓			
General Java platform resources								
java.io.InputStream		✓	✓			✓		
java.io.FileInputStream		✓	✓			✓		
java.io.FileOutputStream		✓	✓			✓		
java.io.BufferedReader		✓	✓					
java.io.FilterOutputStream		✓	✓			✓		
java.io.OutputStream		✓	✓			✓		
java.io.FilterInputStream		✓	✓			✓		
org.apache.http.impl.client.DefaultHttpClient								
java.io.BufferedOutputStream		✓	✓					
java.util.concurrent.Semaphore								
java.io.BufferedWriter		✓	✓					
java.io.ByteArrayOutputStream		✓						
java.io.OutputStreamWriter		✓	✓					
java.net.Socket		✓				✓		
java.util.Scanner		✓				✓		
java.io.ObjectInputStream		✓	✓					
java.io.ObjectOutputStream		✓	✓					
java.io.PipedOutputStream		✓				✓		
java.io.DataOutputStream		✓	✓					
java.io.InputStreamReader		✓	✓					
java.util.Formatter		✓				✓		
java.util.logging.FileHandler								

[1] Resource classes in shaded cells are not supported by any of our studied tools.

Table 8: Resource leak detector performance

Detector	# experimented bugs	# detected bugs (Bug detection rate)	# false alarms (False alarm rate)
Code Inspection	89	61 (68.5%)	47 (52.8%)
Infer	72	40 (55.6%)	16 (22.2%)
Lint	40	12 (30.0%)	0 (0.0%)
FindBugs	38	6 (15.8%)	0 (0.0%)
Relda2-FS [1]	15	13 (86.7%)	10 (66.7%)
Relda2-FI [1]	15	9 (60.0%)	4 (26.7%)
Elite	8	7 (87.5%)	5 (62.5%)
Verifier	8	4 (50.0%)	3 (37.5%)

[1] Relda2 supports two analysis modes: flow-sensitive and flow-insensitive. Relda2-FS and Relda2-FI represents the two modes, respectively.

few minutes. Table 8 reports the bug detection rate and false alarm rate of these detectors. As we can see from the results, the performance of these detectors vary a lot and is generally not satisfactory. In the following, we analyze the results of the detectors in details.

Finding 4: The static analysis of Lint, Code Inspection, and FindBugs is not inter-procedural. Lint, Code Inspection, and FindBugs can analyze an Android app in a matter of seconds and provide nearly real-time feedback to developers. However, despite such efficiency, their analyses are not inter-procedural, meaning that the analyses do not take into account the way that information flows among method calls. Due to this limitation, Lint, Code Inspection, and FindBugs missed 13, 20, and 10 of the experimented resource leak bugs, respectively. Take a resource leak bug in the Xabber app for example. Listing 10 (page 33) gives the simplified code snippet with the patch that fixed the resource leak. As we can see from the code, after the creation of the file and the output stream `out`, the method `rotateImageIfNeeded()` further invokes the bitmap compressing API `compress()` with `out` as an argument. Shortly after the API call returns, `rotateImageIfNeeded()` also returns without properly closing the output stream `out`. Such resource leak bugs would not be detected by Lint, Code Inspection, or FindBugs because when analyzing `rotateImageIfNeeded()`, the three detectors would not further analyze the `compress()` method call, but conservatively assume that the output stream `out` would be closed by the method call `compress()`. Unfortunately, the API call would not close the stream and hence the leak occurs. In comparison, the other two general resource leak detectors Infer and Relda2 do not have such a limitation. Their analyses are inter-procedural. However, for practical concerns, there is still a length limit on the call chain in inter-procedural analyses, meaning that such analyses cannot handle cases where the call chains are exceptionally long. Although rare, we did observe one case, where Infer failed to detect the leak of database cursors (see Owncloud revision `0c8dfb6e8d`).

Finding 5: Lint, Code Inspection, FindBugs, and Infer are not lifecycle-aware and cannot detect resource leak bugs where the re-

```
//Xabber revision 749fc810b6
 1.  public class AccountInfoEditorFragment {
 2.    private Uri rotateImageIfNeeded(Uri srcUri) {
 3.      FileOutputStream out = null;
 4.      try {
 5.        Bitmap oriented = Bitmap.createBitmap(…);
 6.        final File rotateImageFile = createImageFile(…);
 7.        out = new FileOutputStream(rotateImageFile);
 8.        oriented.compress(…, out);
 9.        oriented.recycle();
10.        return Uri.fromFile(rotateImageFile);
11.      } catch (Exception e) {
12.        e.printStackTrace();
13.        return srcUri;
14. +    } finally {
15. +      try {
16. +        if (out != null) { out.flush(); out.close(); }
17. +      } catch (IOException e) {
18. +        e.printStackTrace();
19. +      }
20.      }
21.    }
22.  }
```

Listing 10: A resource leak in Xabber. The resource variable escapes local context. The leak requires inter-procedural analysis to detect.

```
//ConnectBot revision ef8ab06c34
 1.  public class PubkeyListActivity extends ListActivity {
 2.    protected PubkeyDatabase pubkeydb;
 3.    @Override
 4.    public void onCreate(…) {
 5.      //set up the SQLiteDatabase pubkeydb...
 6.    }
 7.    //other methods will query the database
 8.    protected void updateCursor() {
 9.     ...
10.    }
11.    @Override
12.    public void onStop() {
13.      ...
14. +    if(this.pubkeydb != null) {
15. +      this.pubkeydb.close();
16. +    }
17.    }
18.  }
```

Listing 11: A resource leak in ConnectBot. The resource variable is an instance variable.

source variables are defined at class level. In Java programs, there are three kinds of variables: local variables, instance variables, and class/static variables. When evaluating the general resource leak detectors, we observed that Lint, Code Inspection, FindBugs, and Infer can only detect resource leak bugs, where the resource variables are local ones defined in a method under analysis. Take a bug in ConnectBot as an example. Listing 11 (page 33) gives the simplified code snippet with the bug-fixing patch. As we can see, the activity component `PubkeyListActivity` has an instance variable `pubkeydb` of the type `PubkeyDatabase`, which is a subclass of `SQLiteDatabase`. The database is connected when the activity starts (i.e., when `onCreate()` is invoked). It would remain connected while the activity interacts with users. However, when the activity component is terminated, `pubkeydb` is forgotten to be closed, causing resource leaks. Due to the event-driven computing paradigm, such component lifecycle related bugs are common in Android apps (i.e., event handlers often need to share system resources). We observed 14 such bugs in DroidLeaks. However, the four resource leak detectors could not detect these bugs because they analyze methods one after another, but the scope of the resource variables in such cases are beyond a single method (). In comparison, the other general resource leak detector Relda2 does not have this limitation. By modeling the lifecycle of Android app components, it is capable of performing analysis on a sequence of method calls and recovering implicit inter-callback control flows, and thus would be able to detect resource leaks, regardless of the resource variables' scope.

Finding 6: Code Inspection, FindBugs, and Infer do not handle exceptional paths properly in their analyses. As we reported earlier, 56 (19.2%) resource leak bugs in DroidLeaks occurred on exceptional paths. A resource leak detector therefore needs to pay attention to such cases. However, properly handling exceptional paths is known to be a challenge for static analyses (Torlak and Chandra 2010). On one hand, not considering exceptions would miss real resource leak bugs like the one in VLC-Android, whose relevant code snippet and patch are given in Listing 12 (page 35). In our experiments, we observed that FindBugs missed nine such bugs and Infer missed five such bugs. On the other hand, considering exceptional paths that would not be exercised in real executions would generate many false alarms. For example, Code Inspection suggests developers to open I/O resources in a `try` block and close them in the corresponding `finally` block or use the **try-with-resources** statement, which is introduced in Java 7, to ensure that the used resources are properly closed. While such suggestions represent good practices, they are not the only way to guarantee correct resource management. In fact, we observed that real-world developers rarely adopt these practices in our study. For instance, for majority ($71.9\% = 210/292$) of the cases in our data set, we found that developers do not enclose resource operations with the **try-catch-finally** blocks, possibly because the Java exception handling mechanism, which is arguably limited if not flawed, could seriously impair their productivity (Cabral and Marques 2007). Therefore, tools like Code Inspection and Infer generated many uninteresting resource leak warnings when we applied them to analyze

```
//VLC-Android revision 8eb52cba71
 1.  public class Logcat {
 2.    public static void writeLogcat(String filename)… {
 3.      ...
 4.      InputStreamReader in = new InputStreamReader(…);
 5.      OutputStreamWriter out = new OutputStreamWriter(…);
 6.      BufferedReader br = new BufferedReader(input);
 7.      BufferedWriter bw = new BufferedWriter(output);
 8.      String line;
 9. -    while ((line = br.readLine()) != null) {
10. -      bw.write(line); bw.newLine();
11. -    }
12. +    try {
13. +      while ((line = br.readLine()) != null) {
14. +        bw.write(line); bw.newLine();
15. +      }
16. +    } catch (Exception e) {...
17. +    } finally {
18. +      bw.close(); out.close();
19. +      br.close(); in.close();
20. +    }
21. -    bw.close(); out.close();
22. -    br.close(); in.close();
23.    }
24.  }
```

Listing 12: A resource leak in VLC-Android due to I/O exceptions

the patched versions of our app subjects. The high false alarm rate (52.8%) of Code Inspection is due to this reason. 12 of the 16 false alarms generated by Infer is also due to this reason.

Finding 7: Lack of full path sensitivity is a common limitation of all evaluated detectors. Path-sensitive analyses treat different execution paths separately in order to yield precise analysis results. However, the precision comes at the price of analysis overhead. Path-sensitive analyses, even only performed intra-procedurally, are known to be expensive for large code bases and therefore are not widely adopted. Besides, it is hard for static analyses to achieve full path sensitivity due to the approximations they make when handling complex constructs such as loops (static analyses usually unfold loops a finite number of times). During our experiments, we observed that the analyses of Lint, Code Inspection, FindBugs, and Relda2-FI (i.e., Relda2's flow-insensitive mode) are path-insensitive and all failed to detect those bugs that only leak system resources on certain program paths (see Listing 6(a) on page 21 for an example). These detectors simply conclude that there is no resource leak as long as the concerned resource is released on any execution path. Elite, Verifier and Relda2-FS (i.e., Relda2's flow-sensitive mode) could detect resource leaks on certain program paths, but their analyses are still path-insensitive. Relda2-FS performs model checking on a program's value flow graph, which is a concise program representation model that preserves only resource-related information, and would report resource leaks unless the

concerned resources are released on all graph paths after use. Although this model checking approach increased the bug detection rate of Relda2-FS, it also significantly increased the tool's false alarm rate as many paths on the statically constructed value flow graphs are infeasible in reality. Unfortunately, Relda2-FS does not analyze path feasibility. Elite and Verifier merge data flow facts at control flow confluence points and therefore lose the path sensitivity. This is the main reason for them to miss bugs and report false alarms. Comparatively, Infer performs inter-procedurally path-sensitive analysis (on average, Infer takes several minutes to analyze each of our app subjects) and checks path feasibility. However, when the predicates of conditional branch instructions involve return values from library API calls, Infer could still generate false alarms possibly because it does not analyze the internals of library methods. Take the bug in Listing 8 (page 23) for example. The resource leak was fixed by adding the cursor closing statements (Lines 5–7), but Infer would still report a resource leak warning when analyzing the patched version. This is because it cannot properly analyze the library method `isClosed()`, especially when the methods involves native calls, and hence does not know the logic relation between `cur.isClosed()` and `cur.close()` and assumes that `cur` also needs to be closed when both `cur != null` and `cur.isClosed()` evaluate to `true`. We reported this limitation of Infer to its developers.[16]

Besides the above common limitations, the evaluated detectors also have their unique limitations as discussed below:

- Lint does not distinguish resource objects of the same type when analyzing each method. It would not be able to detect resource leaks such as the one illustrated in Listing 8 (page 23) since it does not know that the variable `cur` points to different objects at line 3 and line 9. Lint would also fail to detect resource leaks when the resource object references are not assigned to local variables (see Listing 7 on page 22 for an example, where the anonymous resource objects are passed to other method calls as arguments).
- FindBugs does not analyze class hierarchy and would miss resource leak bugs when the resource classes are self-defined and extend the standard resource classes. For example, the class `PubKeyDatabase` in Listing 11 (page 33) extends the `SQLiteDatabase` class and is therefore also a resource class. However, FindBugs would not consider it as a resource class and thus would not detect resource leaks like the one illustrated in the figure.
- Relda2 cannot capture control flows among threads and therefore would miss bugs or report false alarms when resource acquiring and releasing operations reside in two different threads (see Listing 5 on page 17 for example).
- Verifier does not systematically locate program callbacks defined in each app component and capture the implicit control flows among them. Instead, it only handles a set of pre-defined callback methods. Due to this limitation, it missed several resource leaks.

[16] https://github.com/facebook/infer/issues/679

Answer to RQ5: *The performance of existing resource leak detectors is generally unsatisfactory. The detectors suffer from several common limitations, including improper handling of exceptional paths, not performing inter-procedural analysis, the lack of path sensitivity in code analysis, as well as their own unique limitations.*

6 Discussions

6.1 Threats & Limitations

The validity of our study results may be subject to several threats. We discuss them in the following.

- The first threat is the representativeness of our selected Android app subjects. In our work, we randomly selected 34 popular open-source Android apps from F-Droid. These apps are diverse and cover 13 different app categories. We did not study more open-source Android apps because the study requires careful manual validation of code commits to decide whether they are fixing resource leaks or not. The process is labor-intensive as it requires code comprehension. Manually checking the 1,811 code commit candidates after keyword search took four co-authors several months to finish. After the checking, we obtained a collection of 292 fixed resource leak bugs that cover a diverse set of resource classes. This is already sufficient for carrying out our current study. In the future, we plan to further investigate more open-source apps (e.g., smaller-scale and less popular ones, whose resource management mistakes might be of interest to novice Android developers) and commercial apps to include more resource leak bugs into DroidLeaks.
- The second threat is the potential misses of real resource leak bugs in our keyword search process (i.e., false negatives in bug collection). To reduce the threat, we leveraged the resource operations identified by the state-of-the-art work (Wu et al. 2016) to formulate keywords. We also used general keywords such as "leak", "release", and "close", aiming to maximally cover resource leak bugs that could affect our app subjects. The strategy indeed helped us find a large number of fixed resource leak bugs in 32 out of our 34 app subjects. To understand whether the bugs in DroidLeaks are comprehensive and representative, we further performed two studies as follows. First, we studied the bugs that were used to evaluate the existing resource leak detectors for Android apps in the literature (Pathak et al. 2012, Vekris et al. 2012, Guo et al. 2013, Liu et al. 2014; 2016b, Wu et al. 2016, Wu et al. 2018). We found that the bug patterns discussed in these studies are also included in DroidLeaks. For example, Wu et al. (2016) discussed resource leaks due to complex app component lifecycles. Vekris et al. (2012) pointed out that the high level of asynchronous computing in Android apps (e.g., multi-threading) also often leads to resource leaks. As we can see from Section 4.1, DroidLeaks provides similar instances of such bug patterns and

include many other types of bugs. Second, we searched the issue tracking systems of all our studied 34 Android apps to locate documented resource leak bugs. Specifically, we looked for closed issues whose reports and discussions contain the word "leak".[17] In total, we found 162 candidate issue reports and 75 of them contain links to buggy code or patches, which can help us validate whether a documented bug is indeed a resource leak or not. We then studied these 75 issues and found 16 resource leak bugs (most of the remaining issues are related to memory leaks). We examined the 16 bugs and found that our approach missed two of them. We missed OsmDroid issue #832 because it involves a custom resource class `MapView`, which wraps the standard Android and Java resource classes. Since the resource acquiring and releasing operations are not standard APIs and the developers did not mention the fixing of resource leaks in the commit log, our approach would not be able to find it. We also missed K-9 Mail issue #618 because its developers simply removed the code that does not close resources after file writing. From these observations, we can see that although DroidLeaks does not cover all resource leaks in the 34 projects, its currently indexed bugs are already quite comprehensive. An alternative approach to compiling a comprehensive set of real-world resource leak bugs is to analyze all classes in API specifications and extract all resource acquiring and releasing APIs. One can then analyze the code repository of open-source projects to find the call sites of these resource acquiring APIs and check whether there are potential resource leaks. Future work with a similar goal of bug collection may also follow this alternative approach.

- The third threat is that we assume the developers have fully fixed the bugs in our dataset. Based on this assumption, if a tool reports a resource leak when analyzing a patched version, we would classify the report as a false alarm. However, it is possible that developers may not have completely fixed the bugs in DroidLeaks (it is generally hard for developers to guarantee program correctness). To reduce the threat, we randomly sampled 60 of our 292 bugs and reviewed their patched versions. We observed that for all sampled bugs, the relevant patching code are still in the latest commit in the code base. This indicates that it is highly likely the patches indeed fixed the resource leaks.
- The last threat is the errors in our manual investigation of bugs and manual analyses of experimental results. To minimize the threat, four authors carefully cross-validated the results. We also release our bug dataset for public access (https://zenodo.org/record/2589909).

One limitation of the current version of DroidLeaks is that it does not provide the test cases that can trigger the resource leak bugs. This is mainly because the majority of our studied open-source Android apps do not have

[17] We cannot collect resource leak bug reports by checking issue report labels. None of the 34 projects have labels for resource leak bugs in their issue tracking systems. In fact, the majority of the projects do not even have a clear labeling of bug types and only five of them have labels for general performance bugs.

associated test suites and manually constructing test cases is very expensive. We leave this as our future work. Another limitation of our work is that we did not study the concrete impact of our identified resource leaks. Resource leak is known to be a major type of defects in conventional software (Torlak and Chandra 2010) as well as mobile software (Wu et al. 2016). As a type of performance bugs, resource leaks may not have immediate fail-stop consequences such as app crashes, but they may gradually slow down an app and waste limited computational resources, causing negative user experiences especially when the app runs on low-end mobile devices. We observed that for 103 of the 292 resource leak bugs in DroidLeaks, there are associated bug reports or pull requests, which were merged into the code bases, showing that the resource leaks are of concern to developers. In the future, we plan to construct test cases to trigger different types of resource leak bugs in DroidLeaks and quatitatively study the impact of the bugs. We will also study whether resource leak bugs are relevant to Android users via analyzing app reviews and conducting large-scale user surveys.

6.2 Usefulness of DroidLeaks

DroidLeaks has many potential applications. We discuss several usage scenarios in the following.

- *Programming guidance.* DroidLeaks contains a large number of real-world resource leak bugs, covering diverse types of resources. Its various bugs and patch examples can be used for training or educational purposes, providing programming guidance to Android developers, especially novices.
- *Evaluating bug detection and fixing techniques.* The large number of diverse bugs in DroidLeaks can also be used to evaluate existing resource leak detection and fixing techniques for Android apps to understand their strengths and limitations and to guide the future development of similar techniques. While we have leveraged DroidLeaks to evaluate several existing static analysis based resource leak detection techniques, other researchers can further leverage DroidLeaks to assess other techniques such as resource leak testing (e.g., Yan et al. 2013, Wu et al. 2018) and fixing (e.g., Liu et al. 2016a, Banerjee et al. 2018).
- *Enabling resource leak patching research.* Automatically patching program defects can significantly improve the productivity of software developers. Since resource management policies are well-defined, it is possible to automatically fix potential resource leaks in programs to improve their performance and reliability. Towards this end, DroidLeaks provides diverse resource leak bugs and human-written patches to facilitate the research in automated resource leak patching for Android apps.
- *Supporting pattern-based bug detection.* Our empirical study revealed common patterns of faults made by Android developers. Such patterns can be leveraged to design static checkers (e.g., plug-ins to Android Lint) for real

time detection of resource leaks in Android apps. For example, we implemented a static analyzer on the Soot program analysis framework (Soot 2018) to detect the misuse of the `moveToFirst()` API (see Listing 6(a) on page 21 for example). The analysis simply checks whether the database cursor closing statement is control dependent on the `if` statements that check the return value of the `moveToFirst()` API call. When we applied the analyzer to the latest version of our 34 app subjects, it located 17 database cursor leaks in eight apps. Table 9 (page 41) reports the names of these apps and their versions, in which our checker detected bugs. We reported our findings to the developers of these apps. The last column of Table 9 gives the IDs of our bug reports. So far, all 17 bugs reported in eight bug reports have been confirmed by developers. 16 bugs have been quickly fixed by developers themselves or by merging our pull requests (the bug reports are marked with "+"). The only bug that was not fixed by developers is the one in WordPress (see the last row in Table 9). The reason is that the developers were going to abandon the concerned Java file completely because it was too buggy and hence there was no need to fix our reported bug. This is not our main contribution. We do not further discuss it here.

6.3 Implications on Future Resource Leak Detection Techniques

With our quantitative analysis of existing resource leak detectors, we summarize the characteristics that a resource leak detection technique should have in order to achieve a high precision and recall in practice.

- First of all, its analysis should be inter-procedural. As we mentioned in Section 4.2, for 104 of the 292 bugs in DroidLeaks, the resource variables escape the local context, many of which are passed as arguments to other method calls. We found that IDE tools Android Lint, Code Inspection and FindBugs do not perform inter-procedural analysis and this is a major reason of their false negatives in resource leak bug detection.
- Second, it should handle exceptional paths selectively. While ignoring exceptional paths would lead to false negatives in resource leak detection (e.g., FindBugs missed nine out of 38 bugs due to this reason), reporting resource leaks due to all possible runtime errors would overwhelm developers with a large number of spurious reports (e.g., all 47 false warnings generated by Code Inspection is due to this reason). An effective analysis should process exceptional paths that are likely to be exercised in practice. Existing work (Torlak and Chandra 2010) proposed a belief-based heuristic to enable selective exceptional path analysis.
- Third, its analysis should be lifecycle-aware. Android apps are event-driven programs whose components follow prescribed lifecycles. In our empirical study, we found that Android-specific resources, including `SQLiteDatabase`, `WakeLock`, `MediaPlayer`, `LocationListener`, are often acquired and released in different callback methods. Without considering implicit control

Table 9: Detected Leaks of Database Cursors

App name	Version	# found bugs	Bug report ID
IRCCloud	a96eda0860	2	147+
SureSpot	76b6f931b0	3	142+
OwnCloud	b7577d8d86	1	1818+
SMSDroid	20e9fb149b	1	31+
Osmand	0970ad6496	1	3135+
OSMTracker	d80dea16e4	2	74+
OI File Manager	03aa8903e2	1	82+
WordPress	4a90526c41	6	4526, 4591+

flows among various callback methods, a resource leak detector could miss many leaks of such resources. In our experiments, we found that all tools from the research community (i.e., Relda2, Elite, Verifier) are lifecycle-aware and consider control flows among callbacks. However, the tools from industry, including Android Lint, Code Inspection, FindBugs, and Infer do not handle such implicit control flows and missed many real resource leaks.

– Fourth, its analysis should be able to recognize custom resources. It is often that developers of an app would extend a standard resource class to create a custom resource (see the `PubkeyDatabase` example in Listing 11 on page 33). An analysis tool that cannot recognize such custom resources would not be able to detect their leaks. For example, in our experiments, 11 of the 32 bugs FindBugs missed is due to this reason.
– Lastly, its analysis should be path-sensitive. In our empirical study, we found that 50 out of the 292 resource leaks occur along certain normal paths (see Section 4.2). A path-insensitive analysis would not be able to detect such leaks. For example, our experiments revealed that the analyses of IDE tools Android Lint, Code Inspection and FindBugs are path-insensitive and therefore missed quite a lot of real leaks, which is understandable since these tools are light-weight and expected to provide instant feedback to developers. However, for tools that run off-line analyses, path-sensitivity should be considered to avoid false negatives and false alarms.

7 Related Work

7.1 Resource Management

System resources are finite. Developers are required to release resources used by their apps in a timely fashion when the resources are no longer needed. However, tasks for realizing this requirement are often error-prone. Empirical evidence shows that resource leaks are common in practice (Weimer and Necula 2004). To prevent resource leaks, researchers proposed various language-level mechanisms and automated management techniques (Dillig et al. 2008).

Various tools were also developed to detect resource leaks (Arnold et al. 2011, Liu et al. 2014, Torlak and Chandra 2010, Vekris et al. 2012, Wu et al. 2016). For example, QVM (Arnold et al. 2011) is a specialized runtime that tracks the execution of Java programs and checks for the violations of resource management policies. Tracker (Torlak and Chandra 2010) is an industrial-strength tool for finding resource leaks in Java programs. These techniques are applicable to Android apps, which are typically Java programs, but they do not deal with the specialties in Android apps (e.g., implicit control flows). Therefore, in recent years, researchers also tailored resource leak detection techniques for Android apps. Examples are no-sleep energy bug detector (Pathak et al. 2012), Relda (Guo et al. 2013), Relda2 (Wu et al. 2016), LeakDroid (Yan et al. 2013, Zhang et al. 2016), SENTINEL (Wu et al. 2018) and our earlier work GreenDroid (Liu et al. 2014). Besides the efforts from research communities, there are also industrial tools for resource leak detection for Android apps, such as Facebook Infer, the built-in checker Lint in Android Studio. Despite the existence of so many techniques, there does not exist a common set of real-world resource leak bugs in Android apps to facilitate the evaluation and comparison of these techniques. Our work makes an initial attempt to fill the gap.

7.2 Memory Usage Analysis

Programs written in the Java programming language enjoy the benefits of garbage collection, which frees the developers from the responsibility of memory management. Although developers do not need to care about explicitly recycling the created objects, memory leak may still happen when the programs maintain references to objects that prevent garbage collection or constantly create objects that have poor utility. To help diagnose such memory usage problems, many techniques have been developed. For example, researchers proposed to use object staleness (Bond and McKinley 2006, Hauswirth and Chilimbi 2004), growing instances of types (Jump and McKinley 2007, Mitchell and Sevitsky 2003), and cost-benefit analysis (Xu et al. 2010) to identify suspicious and low-utility data structures that may cause memory leaks. Similar to resource leaks, besides tools originating from research communities, there are also industrial tools for memory usage analysis. For example, Android Profiler[18] in Android Studio and MAT[19] in Eclipse are both powerful and fast tools to help Android developers analyze heap usage for finding memory leaks and reducing memory consumption. As we discussed earlier, many bugs in DROIDLEAKS cause memory wastes and they can also be used to evaluate these techniques.

[18] https://developer.android.com/studio/profile/android-profiler

[19] http://www.eclipse.org/mat/

7.3 Bug Benchmarking

Bug databases/benchmarks enable controlled experimentation and reproducible research. In early years, researchers constructed the widely-used benchmark Siemens (Hutchins et al. 1994), which provides a set of small to median sized C programs with manually seeded faults to facilitate the evaluation of data flow and control flow based testing techniques. Similarly, to facilitate the evaluation of tainting-based data flow analyses for Android, researchers constructed DroidBench (Arzt et al. 2014), whose latest version consists of 120 handcrafted Android apps with malicious data flows (e.g., those leaking users' private data). SIR (Do et al. 2005) is the first benchmark that contains real bugs in Java, C, C++, and C# programs, but still the majority of the bugs indexed by SIR were manually seeded or obtained by mutation and the program sizes are small. In recent years, researchers started to construct benchmarks of real bugs from large-scale software as many complex systems have been open-sourced (Amann et al. 2016, Dallmeier and Zimmermann 2007, Jalbert et al. 2011, Just et al. 2014, Lu et al. 2005). One typical example is the Defects4J bug database (Just et al. 2014). It provides 357 bugs from five large Java programs with exposing test cases. Compared to such bug benchmarks/databases, our DroidLeaks has its unique features. First, to the best of our knowledge, it is the largest collection of real bugs in open-source Android apps. Second, DroidLeaks focuses on resource leaks and covers a wide range of different resource classes. Third, due to its focus, DroidLeaks features resource leaks that occurred due to various root causes and common patterns of coding mistakes, which can support Android programming education and future research.

8 Conclusion

This paper presented DroidLeaks, a database of 292 resource leak bugs in real-world Android apps. We constructed DroidLeaks by analyzing 124,215 code revisions of 34 popular open-source app subjects. To understand the characteristics of these bugs, we conducted an empirical study and discovered common resource management mistakes made by developers. To show the usefulness of our study, we evaluated eight existing resource leak detectors for Android apps using DroidLeaks. The evaluation led to a detailed analysis of the limitations and strengths of the detectors.

In the future, we expect DroidLeaks to further grow and contain more diverse bug instances. We plan to construct test cases to trigger the bugs indexed by DroidLeaks and quantify their impacts. We also plan to evaluate the existing bug patching techniques, especially those specifically designed for resource leaks (Liu et al. 2016a, Banerjee et al. 2018), using DroidLeaks and quantitatively compare their strengths and weaknesses to see whether we can observe new challenges that need to be addressed for effective resource leak repairing. With our efforts, we hope to shed light on future research and

facilitate the development of effective automated techniques to ensure correct resource management in Android apps.

Acknowledgements We would like to thank the reviewers for their valuable comments and improvement suggestions. This work is supported by the National Natural Science Foundation of China (Grant Nos. 61802164 and 61690204), the Hong Kong RGC/GRF (Grant No. 16202917), the Science and Technology Innovation Committee Foundation of Shenzhen (Grant No. ZDSYS201703031748284) and the Program for University Key Laboratory of Guangdong Province (Grant No. 2017KSYS008). The authors would also like to thank the support from the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.

References

S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini. Mubench: A benchmark for api-misuse detectors. In *2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR)*, pages 464–467, May 2016.

Android ANR Errors, 2018. https://developer.android.com/training/articles/perf-anr.html.

Android API Guides, 2018. https://developer.android.com/guide/.

Android Processes and Threads, 2018. https://developer.android.com/guide/components/processes-and-threads.html.

Matthew Arnold, Martin Vechev, and Eran Yahav. Qvm: An efficient runtime for detecting defects in deployed systems. *ACM Trans. Softw. Eng. Methodol.*, 21(1):2:1–2:35, December 2011.

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, 2014.

A. Banerjee, L. K. Chong, C. Ballabriga, and A. Roychoudhury. Energypatch: Repairing resource leaks to improve energy-efficiency of android apps. *IEEE Transactions on Software Engineering*, 44(5):470–490, May 2018.

Michael D. Bond and Kathryn S. McKinley. Bell: Bit-encoding online memory leak detection. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 61–72, 2006.

B. Cabral and P. Marques. Exception handling: A field study in java and .net. In *Proceedings of the 21st European Conference on Object-Oriented Programming*, pages 151–175, 2007.

Valentin Dallmeier and Thomas Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 433–436, 2007.

Isil Dillig, Thomas Dillig, Eran Yahav, and Satish Chandra. The closer: Automating resource management in java. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, pages 1–10, 2008.

Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, October 2005.

F-Droid, 2018. A Catalogue of Open-Source Android Apps. https://github.com/geometer/FBReaderJ.

Facebook, 2018. Infer: A tool to detect bugs in Java and C/C++/Objective-C code. http://fbinfer.com/.

Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, 2011.

Google, 2018a. Android Lint: A Code Scanning Tool for Android Apps. https://developer.android.com/studio/write/lint.html.

Google, 2018b. Android Studio. https://developer.android.com/studio/index.html.

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

Matthias Hauswirth and Trishul M. Chilimbi. Low-overhead memory leak detection using adaptive statistical profiling. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 156–164, 2004.

David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004. ISSN 0362-1340.

M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering*, pages 191–200, May 1994.

Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. Radbench: A concurrency bug benchmark suite. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, pages 2–2, 2011.

Java API Specifications, 2018. https://docs.oracle.com/javase/7/docs/api/.

JetBrains, 2018. Code Inspection in IntelliJ IDEA. https://www.jetbrains.com/help/idea/2016.3/code-inspection.html.

Maria Jump and Kathryn S. McKinley. Cork: Dynamic memory leak detection for garbage-collected languages. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 31–38, 2007.

René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In

Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, pages 437–440, 2014.

Yu Lin, Cosmin Radoi, and Danny Dig. Retrofitting concurrency for android applications through refactoring. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 341–352, 2014.

Mario Linares-Vásquez, Christopher Vendome, Michele Tufano, and Denys Poshyvanyk. How developers micro-optimize android apps. *J. Syst. Softw.*, 130(C):1–23, August 2017.

J. Liu, T. Wu, J. Yan, and J. Zhang. Fixing resource leaks in android apps with light-weight static analysis and low-overhead instrumentation. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 342–352, Oct 2016a.

Y. Liu, C. Xu, S. C. Cheung, and J. Lü. Greendroid: Automated diagnosis of energy inefficiency for smartphone applications. *IEEE Transactions on Software Engineering*, 40(9):911–940, Sept 2014.

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

J. Lovins. Development of a stemming algorithm. *Mechanical Translation and Computational Linguistics*, 11(1 and 2):22–31, 1968.

Shan Lu, Zhenmin Li, Feng Qin, Lin Tan, Pin Zhou, and Yuanyuan Zhou. Bugbench: Benchmarks for evaluating bug detection tools. In *In Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

N. Mitchell and G. Sevitsky. Leakbot: An automated and lightweight tool for diagnosing memory leaks in large java applications. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, pages 351–377, 2003.

Abhinav Pathak, Abhilash Jindal, Y. Charlie Hu, and Samuel P. Midkiff. What is keeping my phone awake?: Characterizing and detecting no-sleep energy bugs in smartphone apps. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 267–280, 2012.

PMD, 2018. A Java Source Code Analyzer. http://pmd.sourceforge.net/.

Soot, 2018. A Framework for Analyzing and Transforming Java and Android Apps. http://sable.github.io/soot/.

Emina Torlak and Satish Chandra. Effective interprocedural resource leak detection. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 535–544, 2010. ISBN 978-1-60558-719-6.

Panagiotis Vekris, Ranjit Jhala, Sorin Lerner, and Yuvraj Agarwal. Towards verifying android apps for the absence of no-sleep energy bugs. In *Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems*, HotPower'12, pages 3–3, 2012.

Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, pages 419–431, 2004.

Haowei Wu, Yan Wang, and Atanas Rountev. Sentinel: Generating gui tests for android sensor leaks. In *Proceedings of the 13th International Workshop on Automation of Software Test*, AST '18, pages 27–33, 2018.

T. Wu, J. Liu, Z. Xu, C. Guo, Y. Zhang, J. Yan, and J. Zhang. Lightweight, inter-procedural and callback-aware resource leak detection for android apps. *IEEE Transactions on Software Engineering*, 42(11):1054–1076, Nov 2016.

Guoqing Xu, Nick Mitchell, Matthew Arnold, Atanas Rountev, Edith Schonberg, and Gary Sevitsky. Finding low-utility data structures. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 174–186, 2010.

D. Yan, S. Yang, and A. Rountev. Systematic testing for resource leaks in android applications. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 411–420, Nov 2013.

Hailong Zhang, Haowei Wu, and Atanas Rountev. Automated test generation for detection of leaks in android applications. In *Proceedings of the 11th International Workshop on Automation of Software Test*, AST '16, pages 64–70, 2016.

Appendix A Open Source Projects Referenced in The Paper

Table 10: The URLs of the code repositories of the open-source projects

App name	Code Repository
AnkiDroid	https://github.com/ankidroid/Anki-Android/
AnySoftKeyboard	https://github.com/AnySoftKeyboard/AnySoftKeyboard
APG	https://github.com/thialfihar/apg
BankDroid	https://github.com/liato/android-bankdroid
Barcode Scanner	https://github.com/zxing/zxing
BitCoin Wallet	https://github.com/bitcoin-wallet/bitcoin-wallet
CallMeter	https://github.com/felixb/callmeter
ChatSecure	https://github.com/guardianproject/ChatSecureAndroid
ConnectBot	https://github.com/connectbot/connectbot/
CSipSimple	https://github.com/r3gis3r/CSipSimple
CycleStreets	https://github.com/cyclestreets/android
c:geo	https://github.com/cgeo/cgeo
FBReader	https://github.com/geometer/FBReaderJ
Google Authenticator	https://github.com/google/google-authenticator
Hacker News Reader	https://github.com/manmal/hn-android
IRCCloud	https://github.com/irccloud/android
K-9 Mail	https://github.com/k9mail/k-9
OI File Manager	https://github.com/openintents/filemanager
Open GPS Tracker	https://github.com/rcgroot/open-gpstracker
Osmand	https://github.com/osmandapp/Osmand
OsmDroid	https://github.com/osmdroid/osmdroid
OSMTracker	https://github.com/nguillaumin/osmtracker-android
ownCloud	https://github.com/owncloud/android
Quran for Android	https://github.com/quran/quran_android
SipDroid	https://github.com/i-p-tel/sipdroid
SMSDroid	https://github.com/felixb/smsdroid
SureSpot	https://github.com/surespot/android
Terminal Emulator	https://github.com/jackpal/Android-Terminal-Emulator
Transdroid	https://github.com/erickok/transdroid
Ushahidi	https://github.com/ushahidi/Ushahidi_Android
VLC-Android	https://github.com/mstorsjo/vlc-android
WebSMS	https://github.com/felixb/websms/
WordPress	https://github.com/wordpress-mobile/WordPress-Android
Xabber	https://github.com/redsolution/xabber-android