MobiLogLeak: A Study on Data Leakage Caused by Poor Logging Practices

Rui Zhou

A Thesis

in the Department of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Applied Science (Electrical and Computer Engineering)
at
Concordia University
Montreal, Quebec, Canada

July 2020

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Rui Zhou

Entitled: MobiLogLeak: A Study on Data Leakage Caused by Poor Logging Practices

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Yan Liu

_____ Examiner
Dr. Amr Yousef

_____ Examiner
Dr. Yan Liu

_____ Supervisor
Dr. Abdelwahab Hamou-Lhadj

Approved by: _____
Dr. Michelle Nokken, Graduate Program Director

_____, 2020

_____
Dr. Mourad Debbabi, Interim Dean,,
Gina Cody School of Engineering & Computer Science

Abstract

MobiLogLeak: A Study on Data Leakage

Caused by Poor Logging Practices

Rui Zhou

Logging is an essential software practice that is used by developers to debug, diagnose and audit software systems. Despite the advantages of logging, poor logging practices can potentially leak sensitive data. The problem of data leakage is more severe in applications that run on mobile devices, since these devices carry sensitive identification information ranging from physical device identifiers (e.g., IMEI MAC address) to communications network identifiers (e.g., SIM, IP, Bluetooth ID), and application-specific identifiers related to the location and accounts of users.

This study explores the impact of logging practices on data leakage of such sensitive information. Particularly, we want to investigate whether logs inserted into an application code could lead to data leakage. While studying logging practices in mobile applications is an active research area, to our knowledge, this is the first study that explores the interplay between logging and security in the context of mobile applications for Android. We propose an approach called MobiLogLeak that identifies log statements in deployed apps that leak sensitive data. MobiLogLeak relies on taint flow analysis. Among 5,000 Android apps that we studied, we found that 200 apps leak sensitive data through logging.

Acknowledgments

During the period of my master, I had the opportunity to learn from many excellent experts. They gave me a lot of advice and help with my thesis.

From the bottom of my heart, I would like to thank my supervisor, Dr. Abdelwahab Hamou-Lhadj, for the support and advice he gave me throughout this whole research. All the time, he shows confidence in me, listens to my thoughts, keeps guiding me with weekly meetings, helps and corrects me no matter when I meet him. I am very appreciated for his advising and extensively reviewing my documents. He has influenced me a lot in the field of research.

Many thanks to Dr. Haipeng Cai from the Department of Electrical Engineering and Computer Science at Washington State University for his assistants and instructions in the process of experiment. He also provides me with clear ideas on my topic.

Then, I would like to thank Kobra Khanmohammadi, the PhD student in our lab, for sharing her knowledge and tool to access the data-set in my experiment. In addition, I would like to thank Steven Arzt, a researcher at the Fraunhofer Institute for Secure Information Technology (SIT) in Darmstadt for helping me to extend the functions of the detection tool.

Also, I would like to thank all the lab mates for their help. They gave me lots of suggestions about researching and provides many useful tips to help me to complete the tasks.

Finally, I would like to thank my family. Although they lived far away from me, they gave me many supports and confidence during my master studies. With their encouragement, I become more and more stronger to face the difficulties and challenges. Thank you all!

List of Contents

List of Figures

List of Listings

List of Tables

Chapter 1 - Introduction

1.1 Objective

Software logging is an important software development practice used by developers to gain insight into the behavior of software systems are run-time [Miranskky 16, Zhu 15]. Log messages printed during the execution of a program are often the only data source available for developers to diagnose program failures [Khatuya 18], detect vulnerabilities and malware infections [Yen 13], detection of system anomalies [Islam 18], etc. Developers of mobile applications (apps), the focus of this thesis, also use logging to keep track of important events that can help them debug problems later. Despite the importance of log data, the practice of logging is still largely ad hoc and somewhat arbitrary [Chen 17b].

There exist studies that examine the practice of logging in mobile app development with a focus on examining the pervasiveness of logging in traditional and mobile applications, the evolution of logging, logging anti-patterns and bed smells, etc. For examples, Yuan et al. [Yuan 2012a] conducted an empirical study on the logging practices in four open source C++ software projects and obtained ten interesting findings on the logging practices. A replication study was conducted by Chen et al. [Chen 17a] focusing on Java systems. The authors examined the logging practices in 21 Java projects from the Apache Software Foundation [ASF 16].

In this thesis, we argue that poor logging practices may lead to more serious problems than quality issues such as exposing user private and personal information and other sensitive data. While studying logging practices in mobile apps is an active research area, to our knowledge, this is the first study that explores the interplay between logging and security (more precisely data privacy) in the context of mobile applications for Android.

1.2 Thesis Outline

The rest of the thesis is structured as follows:

Chapter 2 – Background

This chapter introduces the technologies we used in our thesis. In the first section, we introduce the fundamentals of Android framework. We explain the four main Android components. Then, we review the concept of logging in Android apps. In the second section, we summarize previous work related to the practice of logging in software development. The chapter continues with a detailed literature review, followed by a general discussion.

Chapter 3 - MobiLogLeak Approach

In this chapter, we present our approach, called MobiLogLeak, for detecting logs in Android apps that can be a source of data leakage. We start the chapter with an overview of the approach and continue with describing each component of our methodology.

Chapter 4 - Evaluation

In this chapter, we show the effectiveness of MobiLogLeak when applied to more than 5,000 apps. We discuss the main results and conclude with lessons learned.

Chapter 5 – Conclusion

In this chapter, we revisit the main contributions of this thesis. We conclude with comments about our project and some opportunities for future research**.**

1.3 Related publications

- Rui Zhou, Mohammad Hamdaqa, Haipeng Cai, and Abdelwahab Hamou-Lhadj, "MobiLogLeak: A preliminary study on data leakage caused by poor logging practices," In: *Proceedings of the 27th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2020, pp. 577-58. [Zhou 20]

Chapter 2 - Background

2.1 Android Architecture

Figure 2.1 shows the typical Android development process. Android development is mainly in Java, Kotlin, and C++. After implementing the functionalities of the app, the code is compiled by the Android SDK (Software Development Kit). The result is an archive file with apk extension, which can then be deployed on an Android device.

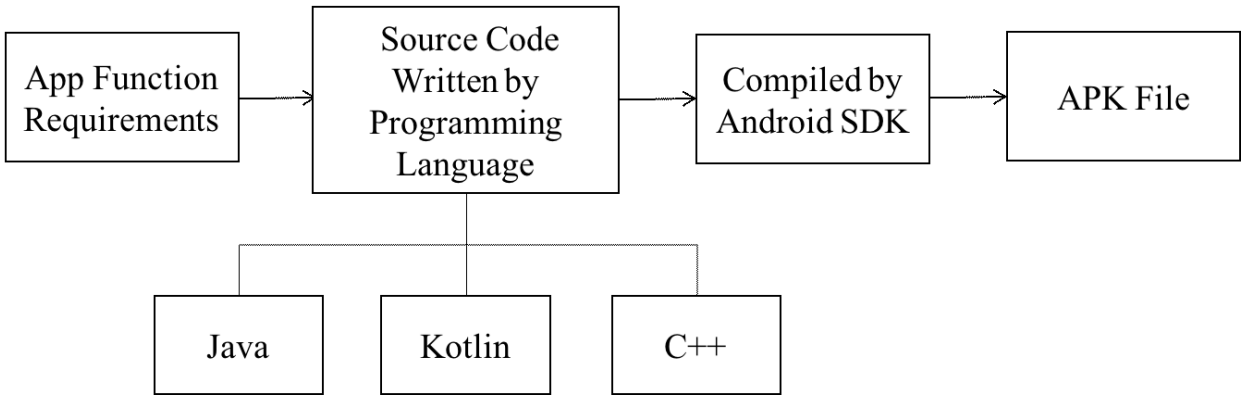


Figure 2.1 Android Apps Development Process

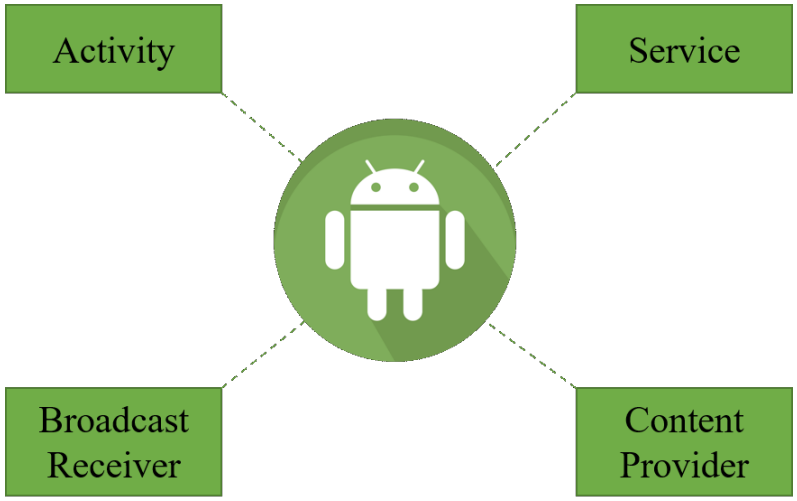


Figure 2.2 Android Components

Android applications have different categories of components that may depend on each other. As shown in Figure 2.2, an Android app can have Activities, Services, Broadcast Receivers, and Content Providers [Android 17]. Each component has its own lifecycle, which defines the functions to create, start, pause, restart, and end the component.

Activity: The Activity component is used to represent the app user interface. Each screen or page in the app is associated to one activity. Android supports various layouts that enable developers to create powerful user interface capabilities.

Service: The Service component is used to implement services that run in the background such downloading a large file, playing background music, etc. Services do not require user interaction and hence do not necessitate a user interface.

Broadcast Receivers: The Broadcast Receiver component is used when the app needs to receive events from other apps. In some cases, even though the app is not running in the background, the

system could still send the broadcasts from other apps to it. For instance, some Android apps could set notifications to users in order to remind them of important events.

Content Provider: The content Provider is used to access the data in the file system. This data may be located in a database, XML file, or stored on remote storage devices.

2.2 Logging in Android Development

A logging statement is typically composed of an object, a verbosity level, a static text, and/or a dynamic content. An example of a logging statement is shown in Listing 1, “Log” is the object. “info” is the verbosity level, which means information, and “log statement” is the static text. A dynamic content could refer to variables (not shown in this example).

```
Log.info("log statement");

Staticinvoke <android.util.Log: int info(java.lang.String,java.lang.String)> ("log statement");
```

Listing 1. Log Message Example

The object is provided by the class android.util.Log. It is the default logging library for Android. Similar to the print statement (e.g. System.out.print), log messages will appear in the backend terminal when the program executes the statement. However, the default logging library does not support any formatting capabilities, making it difficult for developer to process the resulting log files. It just shows the value of its parameters. This is why many developers prefer to import other

advanced logging libraries such as Timber[1], ZLog[2], Logger[3], which provide better formatting features than the standard logging library, hence facilitating post-mortem analysis of log messages. For example, these libraries use different coloring schemes to distinguish among various verbosity levels.

Even though there exist many logging libraries in Android, most of them support five verbosity level: Log.v(), Log.d(), Log.i(), Log.w() and Log.e(), which stand for VERBOSE, DEBUG, INFO, WARN, ERROR. Among these levels, the Verbose and Debug verbosity levels should only be used during development to help developers debug their programs before releasing them. On the other hand, the Info, Warn and Error log messages are part of the program after deployment.

The static text and dynamic content are used by developers to output information about the program. In most cases, this information is used for debugging, which helps check errors and their causes. This explains why the majority of logging statements are usually found in try/catch blocks, for printing exceptions and error messages. In addition, we found that in Android development, developers tend to insert logging statements before the start of services to help monitor the execution of services. In addition, logs help the developers to understand the code and check the performance of as system [Zeng 19]. For example, in Listing 2, the log statement in Line 22 outputs variable $r3, which refers to variable $r1 (shown in Line 20) of type is TimeUnit. Also, $r3 includes the static text, “Closing connections idle longer than”. From this text, we can see that

1https://timber.io

2https://github.com/HardySimpson/zlog

3https://github.com/orhanobut/logger

variable $r1 is sued to measure the time of closing connections; this information can be used to analyze the performance of the program.

Class: org.apache.http.impl.conn.PoolingHttpClientConnectionManager

```
1.  public void closeIdleConnections(long, java.util.concurrent.TimeUnit)
2.  {
3.      org.apache.http.impl.conn.PoolingHttpClientConnectionManager $r0;
4.      long $l0;
5.      java.util.concurrent.TimeUnit $r1;
6.      boolean $z0;
7.      java.lang.StringBuilder $r2;
8.      java.lang.String $r3;
9.      org.apache.http.impl.conn.CPool $r4;
10.     $r0 := @this: org.apache.http.impl.conn.PoolingHttpClientConnectionManager;
11.     $l0 := @parameter0: long;
12.     $r1 := @parameter1: java.util.concurrent.TimeUnit;
13.     $z0 = staticinvoke <android.util.Log: boolean isLoggable(java.lang.String,int)>("HttpClient", 3);
14.     if $z0 == 0 goto label1;
15.     $r2 = new java.lang.StringBuilder;
16.     specialinvoke $r2.<java.lang.StringBuilder: void <init>()>();
17.     $r2 = virtualinvoke $r2.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>("Closing c
    onnections idle longer than ");
18.     $r2 = virtualinvoke $r2.<java.lang.StringBuilder: java.lang.StringBuilder append(long)>($l0);
19.     $r2 = virtualinvoke $r2.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>(" ");
20.     $r2 = virtualinvoke $r2.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.Object)>($r1);
21.     $r3 = virtualinvoke $r2.<java.lang.StringBuilder: java.lang.String toString()>();
22.     staticinvoke <android.util.Log: int d(java.lang.String,java.lang.String)>("HttpClient", $r3);
23. label1:
24.     ...
```

```
25.     return;
26. }
```

Listing 2: Performance Example

App developers use different local directories to store log information including "/data/local/tmp/", "/data/tmp/", etc. Normally, the app developers use Logcat or ADB (Android Debug Bridge) to retrieved and process logs. These tools can get the logs by checking the libraries "/dev/log/main".

2.3 Literature Review of Existing Log Analysis Studies

2.3.1 Logging practices

Yuan et al. [Yuan 12a] surveyed the logging practices in four open-source projects. They deep analyzed which parts of logging code cost developers' most of time to fix. To achieve this objective, they build a prototype with some recurring error patterns to evaluate the feasibility of anticipating the introduction of problematic logging code. In addition, they provide a simple checker which could detect unknown problematic logging statements and confirmed the feasibility of leveraging historical data to improve logging code.

Chen et al. [Chen 17a] conducted a study focusing on Java systems. The authors examined the logging practices in 21 Java projects from the Apache Software Foundation. They addressed five questions related to logging pervasiveness, bug reports, log modification, characteristics of consistent updates and after-thought updates. They set several criteria to filter the log statements and calculated the log density, logging insertion, log deletion, log move, and log update. Their research shows that despite the pervasiveness of logging, the logging practice remains ad hoc and arbitrary.

Zeng et al. [Zeng 19] conducted a research study on the characteristics of logging practices in 1,444 open source mobile apps from the F-droid repository [F-Droid 17]. They checked the logging density, rational behind logging, and performance impact. The authors compared the results to two previous studies [Yuan 12a] [Chen 17a] focusing on C++ and Java applications. They found that logs are less in mobile apps than in server and desktop traditional software projects. Then, they identified several reasons why developers put the log statements in the code: Debug, Anomaly detection, assisting in development, bookkeeping, performance, change for consistency, customized logging library and from third-party library.

Shang et al. [Shang 14] proposed an approach where they used the development knowledge to help understand the meaning and impact of each specific lines. The approach starts by examining the email threats from Hadoop, Cassandra and Zookeeper. They spend 100 hours on manually detection, and they found 15 email inquires and 73 inquires from web research. Then, they summarized five types of development knowledge which are applied a lot in industry, including meaning, cause, context, impact and solution, to help understand log lines in code commits, issue reports and other development repositories. In conclusion, they argued it's feasible to use the derived development knowledge to resolve the real-life log inquires.

In another study, Pecchia et al. [Pecchia 15] surveyed the event logging practices in the area of critical industrial development. They corporated with Selex ES, a company focus on electronic and information solutions for critical system. They want to answer: how do developers work? Why do developers work? How do industry practices impact event logging? They did the experiment by accessing the source code of software, inspecting about 2.3 million log entries and getting the feedback directly from the development team. Their research contains programming practices,

logging objectives and issue impacting log analysis. They get 7 findings from their analysis and their work contribute a lot to improve the event logging reengineering tasks at Selex ES.

Fu et al. [Fu 14] run a study with Microsoft developers to investigate where software developers put log statement in industrial systems. The study focuses on three research questions: What categories of code snippets are logged? What factors are considered for logging? Is it possible to automatically determine where to log? In their experiment, they accessed the source code of 2 Microsoft systems. Also, they used a questionnaire survey with 54 experienced Microsoft developers. They proposed six patterns of where the logging statements are located, and their prediction accuracy is up to 90% F-Score. Their results suggest that it is feasible to predict the logging statement location using a classification model.

Another study was conducted by Li et al. [Li 17a], where the authors proposed suggestions to developers for better classification of new log statements with a focus on log levels. Normally, log levels include fatal, debug, info, warn and error. However, in their studies, software developers often have problems deciding which level they should give to the new statements. Therefore, they investigated the development history of Hadoop, Directory Server, Hama and Qpid to survey the log level practices and build a classifier that predict the adequate log level based on historical data.

2.3.2 Quality of Logs

Cinque et al. [Cinque 10] proposed an approach to prove the effectiveness of log statement when software fault. They used the G-SWFIT technique [Durase 06], which could inject faults to applications, to cause the failure of tested software. Then, they check the log files which keep track of this problems and verify whether the log files could help detect the faults and help resolve them.

They tested Apache server, TAO open data Distribution Service and MySQL. In their result, 60% failures are caused without leaving log trace. They hope their work will contribute to the improvement of logging mechanisms. Cinque et al. [Cinque 12] proposed a rule-based approach after their first study, which aims at improving the logs effectiveness of analyze software failures. This approach works during the development time and give several criteria for the placement of logging statements in source code. They tested this approach and it detect about 12,500 software fault injection in real-world systems.

Chen et al. [Chen 17a] concentrated on the quality of logging and identified logging anti-patterns. In their work, they identified six types of misuse of logs, including null-able objects, explicit cast, wrong verbosity level, logging code smells and malformed output. In the logging code smells, they detected two kind of duplication. They manually examined 352 pairs of independently changed logging code snippets from ActiveMQ, Hadoop, and Maven. They provided a tool called LCAnalyzer, which helps detect these anti-patterns.

One of logging anti-pattern from Chen et al. [Chen 17a], duplication, is deeply studied by Li et al. [Li 19]. They did the research in 4 open-source system (Hadoop, CloudStack, ElasticSearch and Cassandra) and they manually studied more than 3,000 duplicate logging statements. Furthermore, they summarized 5 patterns of duplication logging code smells. Then, they verified their results by contacting the developers and get their feedback. After research, they provide the tool named DLFinder, which could conduct automatically log static analysis in duplication. Next, they tested their tool by applying it in another 2 systems, Camel and Wicket. In their results, DLFinder reported 82 duplicated code smell instances and in the end, all of them are fixed by the developers.

Another tool named “LOGADVISOR” was proposed by Zhu et al. [Zhu 15] by applying Machine Learning. The tool is used to guide software developers on to place the new log statements. To evaluate the feasible of their tool, LOGADVISOR, they tested it on two systems from Microsoft and two open-source projects which are maintained in GitHub. Their results show that they can make provide good recommendations to developers on how to write log statement.

Khanmohammadi et al. [Khanmohammadi 19] conducted research on Android repackaged apps, which are considered as one the top 10 risks in mobile security [OWASP 16]. They tested more than 15,000 apps from AndroZoo [Li 17b] and studied the motivation of developers and users of repacked apps. Also, they detected the factors which determine the apps to be repackaged and the ways how these apps are repackaged. Their insights can be of a great help to security experts. In addition, a novel app indexing scheme was proposed to minimize the number of comparisons needed to detect repackaged apps in app stores.

2.3.3 Android Vulnerability Analysis

Software systems are known to have vulnerabilities that can be exploited by attackers [Murtaza 16]. In the context of Android apps, there is a large body of research on app vulnerabilities. In this thesis, we present some studies related to this thesis. Cai et al. [Cai 17] created a software toolkit, called DroidFax, that is designed to assist developers in program comprehension of Android applications. This tool shows the app characterization in multiple dimensions and views. Particularly, DroidFax provides all sorts of statistics about an Android app for quality assessment. The authors applied the tool to the analysis of 125 apps selected randomly from Google Play and concentrate on their dynamic characteristics. Also, they used it on 610 sample apps of malware.

Their results provide new insights about Android app behavior. It also helps provide a comprehensive understanding of the code structure of Android apps.

Lu et al. [Lu 12] proposed an approach to automatically detect component hijacking vulnerability in Android apps. They used a static analysis framework called CHEX (Component Hijacking Examiner), which operates on the system dependency graph, extracted from Android app bytecode. The graph is analyzed the detect possible hijack-enabling flow paths. The authors evaluated their approach on 5,486 Android apps and found 254 apps with potential component hijacking vulnerabilities. In their result, CHEX spent 37.02 seconds on each Android app in average, concluding that the approach can scale up to larger datasets.

Huang et al. [Huang 14b] proposed a novel approach, AsDroid, to check stealthy behaviours in Android apps that may be exploited by attackers to insert malware. Their approach concentrates on the analysis of top-level functions of an Android app, which are the functions that are executed the most when users interact with the app. The authors also used the text from the user interface component. They analyzed the top-level functions and the extracted text to check whether they match or not. In their work, AsDroid reported that 113 apps have stealthy behaviours, including 28 false positives and 11 false negatives.

Arzt et al. [Artz 14] developed a tool, called FlowDroid, which performs static taint analysis of Android apps. The tool does not assume the presence of the app source code. Instead, it operates on Jimple code. It could provide the location of source and sink in the taint flow path. The authors applied this tool to Google Play apps, and succeed to detect vulnerabilities in 500 apps and around 1,000 malware apps used in the VirusShare project. Similarly, Huang et al. [Huang 14a] conducted a study on applying taint analysis to Java-based web applications. They presented SFlow and

SFlowIner to make the dataflow and point-to-based taint flow analysis. In addition, their work focuses on handling reflection, library and framework.

Huang et al. [Huang 16] conducted a study to detect sensitive data revelation in Android apps. They proposed a novel static analysis technique, called BIDTEXT, that concentrates on the variable-related text labels to check the potential disclosure of sensitive. They experimented with a dataset of 10,000 Android apps downloaded from the Google Play store. The results show that 4,406 Android apps incur data disclosure through logging and HTTP requests.

Rasthofer et al. [Rasthofer 14] proposed SUSI, a novel machine-learning guided approach for identifying sources and sinks directly from the code of any Android API. Based on Android apps, they categorized the sources and sink using different labels. For the source, it has a unique identifier, account, Bluetooth, etc. and for the sink, it has network, files, etc. Their approach was evaluated on 11.000 malware samples and in their results achieve 92%.

Feng et al. [Feng 17] conducted a study on selecting the critical data flows based on the differences between benign apps and malware apps. They presented a tool, Scoflow, to automatically detect these critical data flows. They used these flows to help distinguish the malware abnormal sensitive data usage. In their results, they compared Scoflow and Mudflow. They found their tool has high rate of malware detection by 5.73%~9.07% on different dataset. They also argued that their tool takes less memory space than Mudflow.

Rahul et al. [Rahul 13] conducted research on system permission of mobile apps by using the analysis of nature language. The authors examined the app's description and checked whether the permissions requested by the app were justified. For example, a typical question is: why does this app need this specific permission? They presented a framework based on natural language

processing called WHYPER. In the results, the framework achieved an average precision of 82.8%, and an average recall of 81.5% for address book, calendar, and record audio permissions.

2.4 Summary

Logging is an important practice in mobile software development that is used by developers to debug and analyze apps. There exist studies that focus on understanding the practice of logging in general-stream software development with a focus on the quality of logs as well as where and how to log. Logs, however, can be misused yielding security problems. To our knowledge, there are no studies that examine the interplay between logging and security and data privacy, which is the main objective of this thesis.

Chapter 3 - Mobilelike Approach

3.1 Overview

This thesis focuses on studying potential data leakage due to poor logging practices in released Android mobile applications. Figure 3.1 shows an overview of our approach, called MobiLogLeak, for detecting log statements in Android apps that may potentially leak private data. Our approach consists of five steps. First, we convert the Android APK into an intermediary representation using Jimple code [Bartel 12] in order to be able to analyze its content (Section 3.1). We do this because we do not assume the presence of the course code. Then, we apply taint analysis to the resulting Jimple code to identify the list of taint flow paths (Section 3.2). In our work, we focus only on apps with taint flows since these are the potential sources for data leakage. Moreover, since this study focuses on data leakage as a result of poor logging practices, we prune paths that are not log-related. To do that, we use source-sink paths generated through taint flow analysis, and search the sinks for possible log related statements (see Section 3.4 .

The results will be log-specific taint flow paths. After this, to obtain a better understanding of these generated log related flows, we perform context analysis to analyze the code structure in order to uncover the Android components that have the most log related flows (Section 3.4). Finally, we manually inspect each of the taint flow paths to understand the type of the possible data leakage cases (Section 3.6).

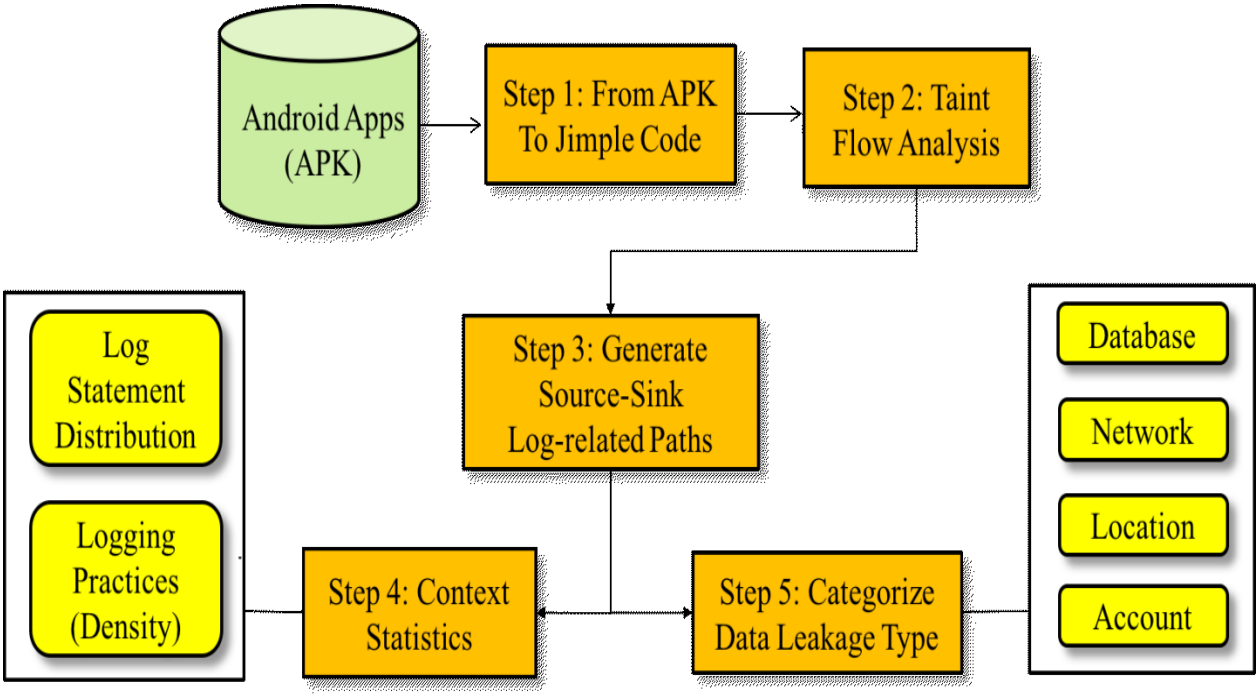


Figure 3.1 Overview of Our Approach

3.2 Converting App APK to Jimple code

Since we are interested in analyzing log statements in deployed apps, we cannot assume the presence of the source code. We therefore need to reverse engineer the app APK to an intermediate representation that we can analyze. To this end, we turn to the Soot framework [Bartel 12], which is used for analyzing and transforming Java and Android apps to Jimple, a code format between source code and Byte code that is commonly used in program analysis.

Although it is longer than the normal source code, Jimple code maintains the required program constructs needed for program analysis, such as the function names, classes and code statements.

Hence, it can be used to identify the logging statements in an Android app. Figure 3.2 shows an example of a Jimple code snippet of a logging statement that is generated from an APK using Soot.

Java	Jimple

1 static int factorial (int x){
2 int result = 1;
3 int i = 2;
4 while (i <= x) {
5 result *= i;
6 i++;
7 }
8 return result;
9 }
	1 static int factorial (int) {
	2 int x, result, i, temp$0, temp$1, temp$2, temp$3;
	3 x := @parameter0: int; /*1*/
	4 result = 1; /*2*/
	5 i = 2; /*3*/
	6
	7 label0:
	8 nop; /*3*/
	9 if i <= x goto label1; /*4*/
	10 goto label2;/*4*/
	11
	12 label1:
	13 nop; /*4*/
	14 temp$0 = result; /*4*/
	15 temp$1 = temp$0 * i; /*4*/
	16 result = temp$1; /*5*/
	17 temp$2 = i; /*5*/
	18 temp$3 = temp$2 + 1; /*6*/
	19 i = temp$3; /*6*/
	20 goto label0; /*4*/
	21
	22 label2:
	23 nop; /*4*/
	24 return result; /*8*/
	25 }

Figure 3.2 Source code and its corresponding Jimple code

The Java code contains one static function, which returns an integer. Inside the function, there is one while-loop to modify the variable resulting from multiplying it by 1 to x. Compared to Java code, Jimple code also shows the function type, return value and parament. Rather than showing the while-loop, Jimple uses “goto” to control the data flow in the process. The data is from label

0 to label 1 and then comes from the label 1 to label 0. Jimple code uses another way to achieve the while-loop.

3.3 Taint Flow Analysis

The main aspect of our approach is to identify sources of data leaks that are related to log statements. One way to identify data leaks is through taint flow analysis [Klieber 14]. The goal of taint flow analysis is to check whether sensitive data remains within an expected application's boundaries [Klieber 14]. Taint analysis can be applied statically or dynamically.

In this thesis, we apply static taint flow analysis on the generated Jimple code from the previous step. We achieve this using FlowDroid [Arzt 14], which is a static analysis tool built on Soot that allows us to retrieve the data flow between sources and sinks, hence uncovering all the paths that are related to data leaks.

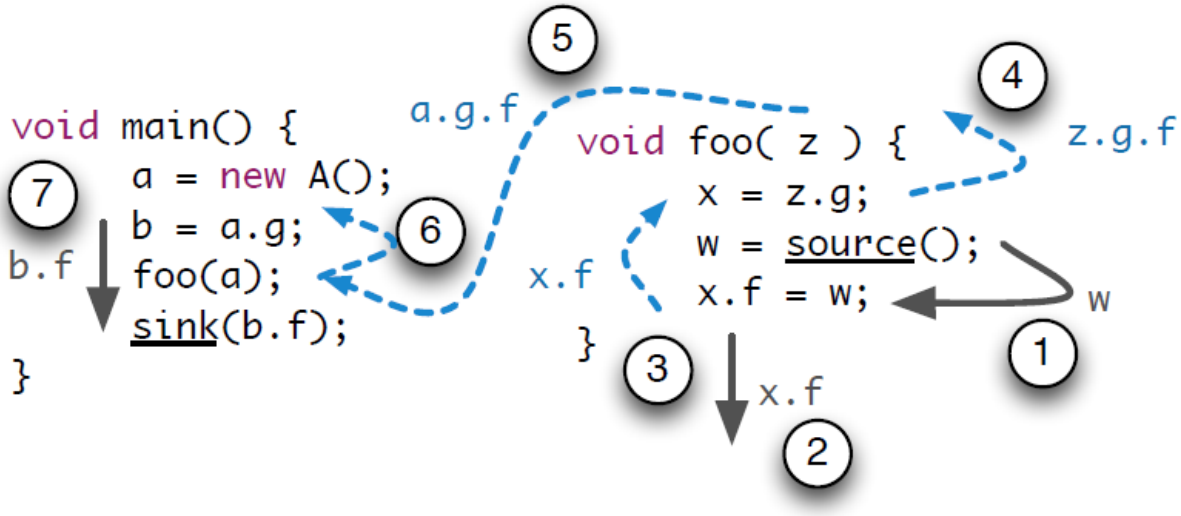


Figure 3.3 Taint flow analysis example

Arzt et al. [Arzt 14] proposed a tool, called FlowDroid, to help find the taint flows in Android apps. The tool is based on the Soot framework and operates on Jimple code. FlowDroid takes the

apk file as input and builds a control flow graph, which shows the source and sink of taint flows. Taint flow analysis can be described using the example of Figure 3.3. In Label 1, the variable w gets the value from source(). Then, the source is transformed from x.f to a.g.f (Label 2 to Label 5). Next, the variable b gets the variable a.g and in the end, the sink gets the b.f in Label 7, which is the source in Label 1.

3.4 Generating the Source-Sink Log-Related Paths

The paths that are generated in Step 2 include all sources of data leaks (shown in Figure 3.4). In this step, we refine the list of paths, by focusing only on paths that are related to log statements. In other words, the goal is to retrieve only paths whose sinks contain a logging related statement.

In previous studies (e.g., [Chen 17a]), researchers filter the log statements using regular expressions and keywords such as the ones shown in Figure 3.5. In other words, they look at statements that contain the word “Log”. The problem is that this approach contains many false positives such as expressions that contains words like “dialog” and “login”, which are not log statements.

In Jimple, shown in Listing 1, log statements are accompanied with their library classes (e.g., android.util.Log), making it easy to filter the generated taint flow analysis paths by keeping only those that contain log statements as their sink. A simple string match search looking for the log related libraries in the sink for the taint flow paths suffices. The result of this step are all the taint flow paths that are related to log statements. In the rest of this thesis, we refer to these paths as ALRS (App Logging Related Sinks).

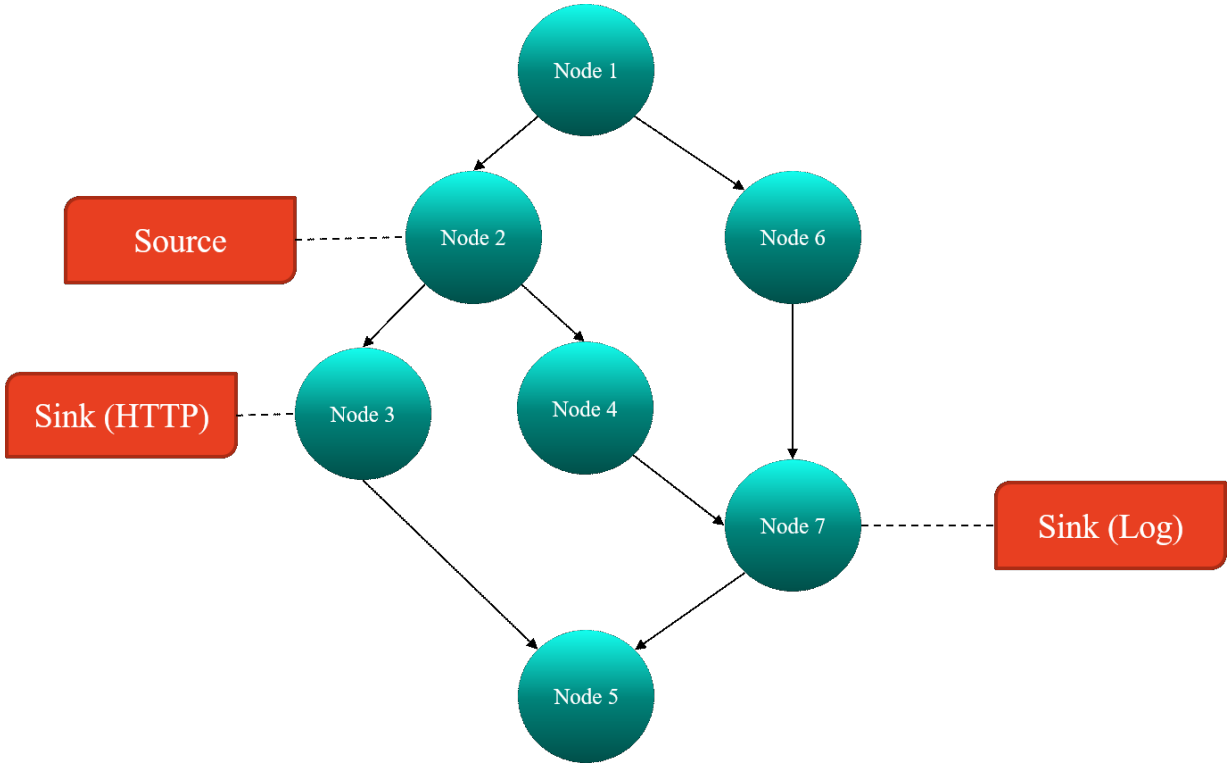


Figure 3.4 Example of a taint flow graph

uses regular expressions to match the source code. The regular expressions used in this paper are *".*?(pointcut|aspect|log|info|debug|error |fatal|warn |trace|(system\.out)|(system\.err)).*?(.*?);"*:

- "(system\.out)|(system\.err))" is included to flag source code that uses standard output (System.out) and standard error (System.err).
- Keywords like "log" and "trace" are included, as the logging code, which uses logging libraries like log4j, often uses logging objects like "log" or "logger" and verbosity levels like "trace" or "debug".
- Keywords like "pointcut" and "aspect" are also include to flag logging code that uses the AspectJ (The AspectJ project 2015).

After the initial regular expression matching, the resulting dataset is further filtered to removed code snippets that contain wrongly matched words like "login", "dialog", etc. We manually sampled 377 pieces of logging code, which corresponds to a 95 % of confidence level with a 5 % confidence interval. The accuracy of our technique is 95 %, which is comparable to the original study (94 % accuracy).

Figure 3.5 Log Filter in Previous Research

3.5 Context Analysis of Log-Related Paths

In this step, we measure different aspects related to 'bad' log statements (logs that appear on the taint flow analysis paths). These aspects include the component, the class, the block of this log statement. In the component level, we check these logs are in Activity, Service, Broadcast Receiver or Content Provider. In the class level, we print their java class names. In block level, we verify whether they are in exception block or not. If yes, we go deep and find they are in the part of catch or final.

For this, we use DroidFax [Cai 17], a software toolkit that is designed to assist developers in program comprehension of Android applications. Particularly, DroidFax provides all sorts of statistics about an Android app for quality assessment. Similar to FlowDroid [Arzt 14], it also analyzes the application using Jimple code. We used FlowFax to check the log density, identify the components that are logged the most, identify parts of the code that contain logs such as exceptions and reflective functions. For example, in Listing 3, there is one log statement in Line 14. From this code snippet, we could generate Figure 3.6 for its details.

```
1.    Class c.a.a.b.a
2.    public void onPreviewFrame(byte[], android.hardware.Camera)
3.    {
4.        ...
5.        java.lang.String $r17;
6.        java.lang.NullPointerException $r18;
7.        java.lang.ArrayIndexOutOfBoundsException $r19;
8.        java.lang.Throwable $r20, $r21;
9.        ...
10.   label29:
11.       $r16 := @caughtexception;
```

```
12.         $r17 = virtualinvoke $r16.<java.lang.RuntimeException: java.lang.String toString()>();
13.         $r20 = (java.lang.Throwable) $r16;
14.         staticinvoke <android.util.Log: int e(java.lang.String,java.lang.String,java.lang.Throwable)>("ZXingScanner
View", $r17, $r20);
15.         return;
16.         ...
17.     }
```

Listing 3: Log Snippet Example

As shown in Figure 3.6, we can directly find the context of log statement, including its method name (onPreviewFrame) and class name (c.a.a.b.a). Furthermore, when we check the app downloaded information, we can retrieve the app hash key (which is 00CFCA10 in our case) and the package name (net.intricare.gobrowserkiosklockdown), highlighted in blue color in Figure 3.6.

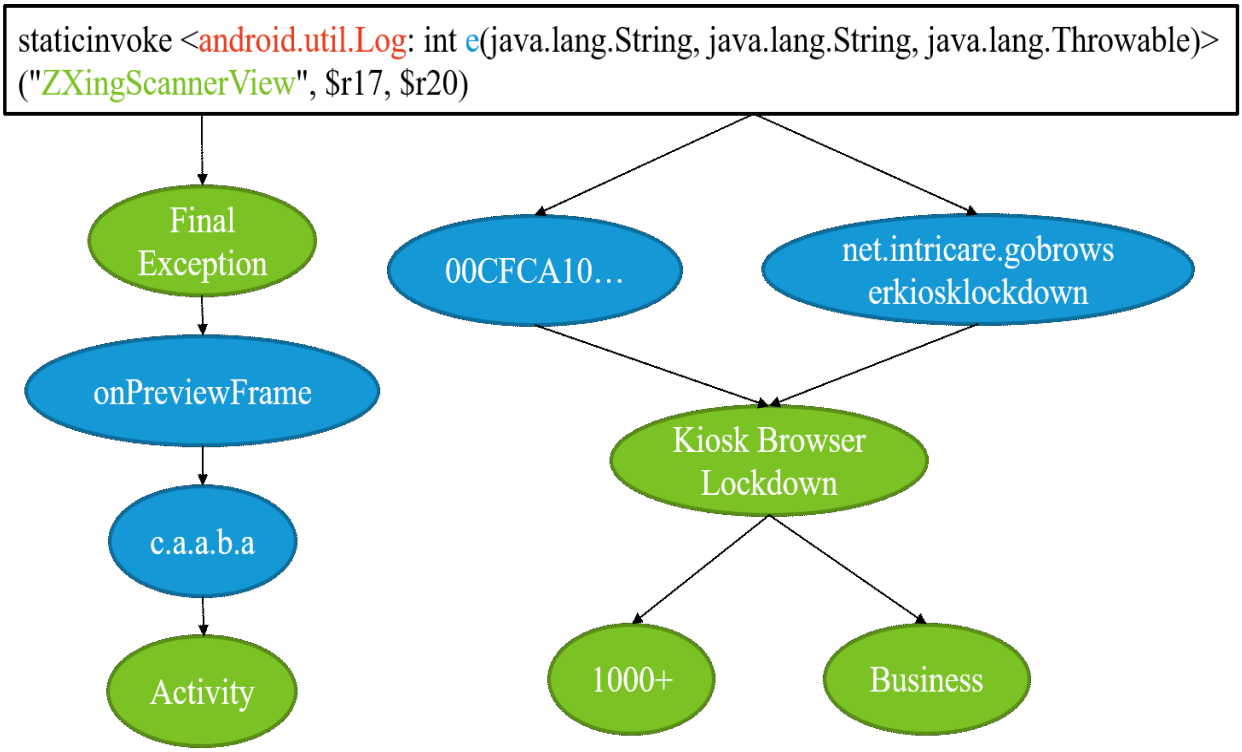


Figure 3.6 Overview of Context Process

In addition, we can further use Soot to find more information such as the Activity component, the code blocks, etc. (shown in green in Figure 3.6).

DroidFax provides the API to obtain the component type. The name is getComponentType and it is shown in the Listing 4. It requires the soot class and return the type.

```
1.  public static String getComponentType(SootClass cls) {
2.    try {
3.      if (fhar==null) {
4.        fhar = Scene.v().getOrMakeFastHierarchy();
5.      }
6.      if (fhar.isSubclass(cls, iccAPICom.COMPONENT_TYPE_ACTIVITY))
7.        return "Activity";
8.      if (fhar.isSubclass(cls, iccAPICom.COMPONENT_TYPE_SERVICE) ||
```

```
9.              fhar.isSubclass(cls, iccAPICom.COMPONENT_TYPE_GCMBASEINTENTSER          VICECLASS) ||
    fhar.isSubclass(cls, iccAPICom.COMPONENT_TYPE_GCMLISTENERSERVICECLASS))
10.             return "Service";
11.         if (fhar.isSubclass(cls, iccAPICom.COMPONENT_TYPE_RECEIVER))
12.             return "BroadcastReceiver";
13.         if (fhar.isSubclass(cls, iccAPICom.COMPONENT_TYPE_PROVIDER))
14.             return "ContentProvider";
15.         if (fhar.isSubclass(cls, iccAPICom.COMPONENT_TYPE_APPLICATION))
16.             return "Application";
17.         return "Unknown";
18.     }
19.     catch (Exception e) {
20.         e.printStackTrace();
21.         return "Unknown";
22.     }
23. }
```

Listing 4: Get Component Type

When we use this function, we get the class before checking the log statement. It means we look at the class at first and get the type. Then, all the log statements will be marked by this label. Different from it, exception block detection is done at the level of blocks. It uses another object named Body that helps check the exception features in the method.

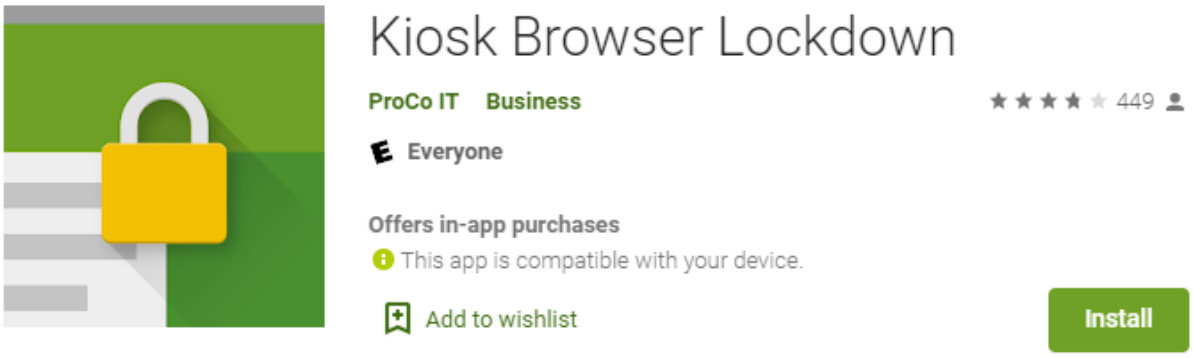


Figure 3.7 App Detail in Google Play

In addition, we can cross reference the Hash Key and package name with information from Android app stores (such as Google Play) such as the app name (Kiosk Browser Lockdown), the number of downloads (more than 1,000) and the app category (Business) (Shown in Figure 3.7).

We compare the number of log statements in ALRS and those where no taint flow paths were discovered. First, we compare the logging density between applications with taint flows (ALRS) and those without. Second, we compare the context, in which the log statements appear, in other words, the distribution of log statements in the different Android components: Activities, Services, Content Providers, and Broadcast Receivers (shown in Figure 3.8).

For each app, we first identify the component that has the largest number of logs. Then, we calculate how many apps with this component that have most logs in our dataset (AWLC).

To do it, we use the equation as below:

$$Component\ Percentage = \frac{AWLC}{\text{Total Apps in This Year}}$$

To compare the component percentage of Good apps and ALRS., we separately computing it and list the result in Table 4-3.

To measure the density of logging, we use the following equation:

$$Density = \frac{LLOC}{\text{SLOC}}$$

where LLOC refers to the total number of log statements of Jimple code and SLOC refers to the total number of code source lines of Jimple Code.

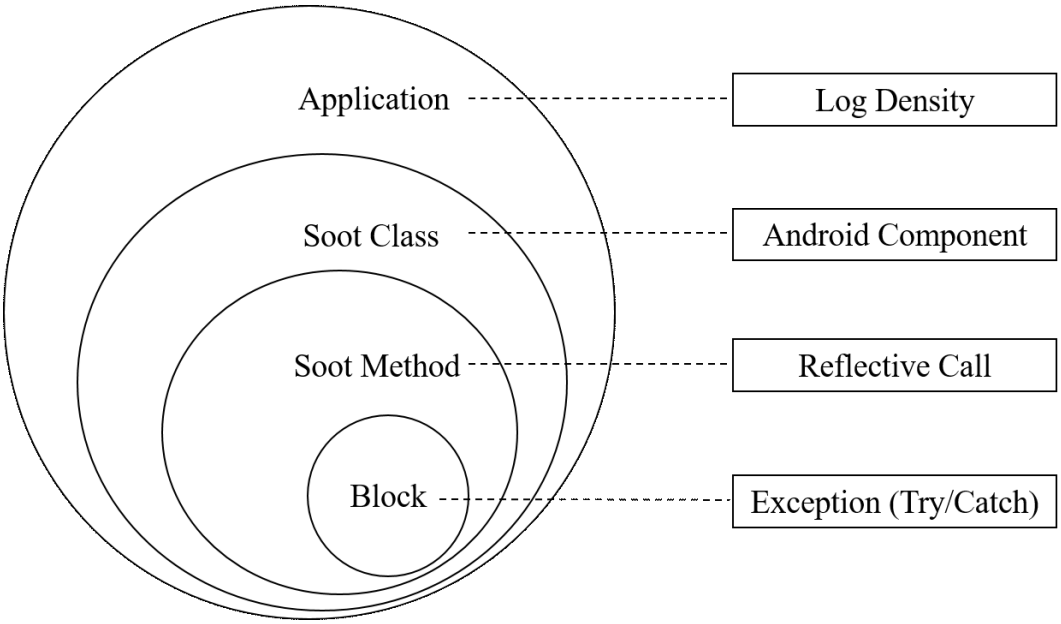


Figure 3.8 Context Statistics Level

Meanwhile, we calculate the percentage of exception block and reflective method. Exception is used often during Android development, and it often uses log to show the errors. To clarify how many logs in this part, we used the following equation:

$$Exception\ Percentage = \frac{LLE}{\text{LLOC}}$$

Inside, the LLE is Line of Log statements in Exception block. Same to it, we calculated the reflective calls by the equation:

$$Reflective\ Percentage = \frac{LLR}{\text{LLOC}}$$

LLR is Line of Log statements in Reflective method.

3.6 Types of Log-Related Data Leakage

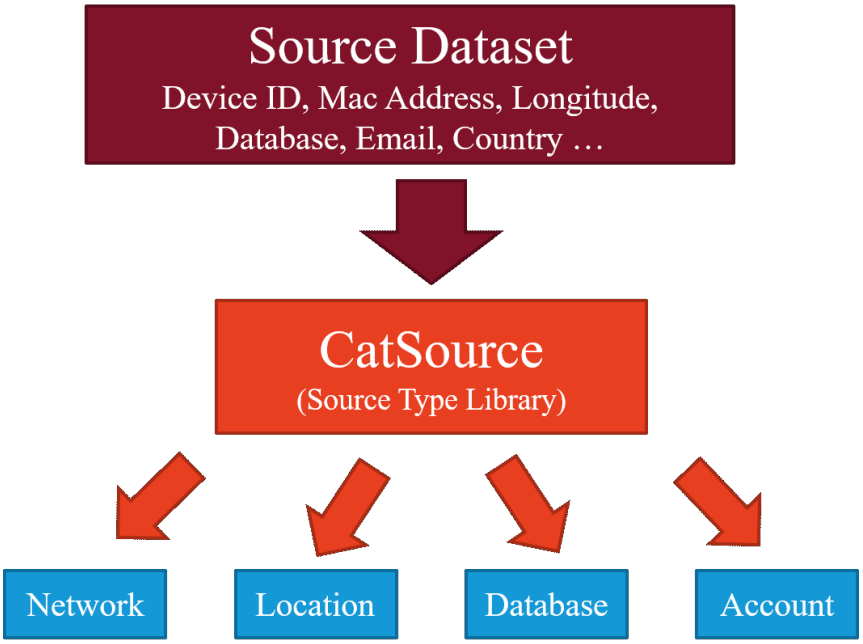


Figure 3.9 Source Categorization

The aim of this step is to categorize logging related data leakage cases. To this end, we manually inspect each of the taint flow paths generated in Step 3 and categorize them into four identified data leak types, which are database related, network related, location related, and account related. As shown in Figure 3.9, we start from the source of taint flow path. In each source, the type is different. From DroidFax, we could categorize it bash on its features by CatSource. CatSource is one file which list all possible data source in taint flow analysis. We list four categories in Listing 5 and give five examples for each category.

LOCATION_INFORMATION:

<android.location.Country: java.lang.String getCountryIso()> (LOCATION_INFORMATION)

<com.android.server.location.PassiveProvider: long getStatusUpdateTime()> (LOCATION_INFORMATION)

<android.telephony.TelephonyManager: java.lang.String getNetworkTypeName()> (LOCATION_INFORMATION)

<android.location.Address: int getMaxAddressLineIndex()> (LOCATION_INFORMATION)

<android.location.LocationRequest: long getInterval()> (LOCATION_INFORMATION)

NETWORK_INFORMATION:

<com.android.server.pm.PackageManagerService$ServiceIntentResolver: java.util.List queryIntentForPackage(android.content.Intent,java.lang.String,int,java.util.ArrayList,int)> (NETWORK_INFORMATION)

<android.net.wifi.p2p.WifiP2pWfdInfo: int getControlPort()> (NETWORK_INFORMATION)

<com.android.server.AppWidgetServiceImpl: java.util.List getInstalledProviders()> (NETWORK_INFORMATION)

<android.support.v4.view.ViewPager$2: float getInterpolation(float)> (NETWORK_INFORMATION)

<com.android.internal.telephony.IccIoResult: java.lang.String toString()> (NETWORK_INFORMATION)

ACCOUNT_INFORMATION:

<android.accounts.AccountManagerService$Session: android.accounts.IAccountManagerResponse getResponseAndClose()> (ACCOUNT_INFORMATION)

<android.accounts.AccountManager: android.accounts.AccountManagerFuture confirmCredentials(android.accounts.Account,android.os.Bundle,android.app.Activity,android.accounts.AccountManagerCallback,android.os.Handler)> (ACCOUNT_INFORMATION)

<android.accounts.AccountManagerService: android.accounts.Account[] getAccountsAsUser(java.lang.String,int)> (ACCOUNT_INFORMATION)

<android.test.IsolatedContext$MockAccountManager: android.accounts.AccountManagerFuture getAccountsByTypeAndFeatures(java.lang.String,java.lang.String[],android.accounts.AccountManagerCallback,android.os.Handler)> (ACCOUNT_INFORMATION)

<android.accounts.IAccountManager$Stub$Proxy: java.lang.String getUserData(android.accounts.Account,java.lang.String)> (ACCOUNT_INFORMATION)

```
21. SMS_MMS:
22. <com.google.android.mms.pdu.DeliveryInd: byte[] getMessageId()> android.permission.STOP_APP_SW
    ITCHES (SMS_MMS)
23. <com.google.android.mms.pdu.AcknowledgeInd: byte[] getTransactionId()> (SMS_MMS)
24. <com.google.android.mms.pdu.NotifyRespInd: byte[] getTransactionId()> (SMS_MMS)
25. <com.google.android.mms.ContentType: java.util.ArrayList getVideoTypes()> (SMS_MMS)
26. <com.google.android.mms.pdu.PduBody: com.google.android.mms.pdu.PduPart getPartByContentId(jav
    a.lang.String)> (SMS_MMS)
```

Listing 5: CatSource: Examples of Different Categories

For example, if we find the source is from the Android default function, getPartByContentId, we will find the function in the CatSource file. Then, we could see the function in Line 27 of Listing 5. Therefore, we mark this source as SMS_MMS.

3.7 Summary

In this chapter, we presented our approach for detecting log statements that potentially leak sensitive data using taint flow analysis that is generated by FlowDroid.

Our method uses static analysis of Jimple code generated from app APKs. The taint flow analysis automatically narrows the range of apps which are potential to leak data from apps. In addition, we study the context of the sinks and sources to understand where risky logs appear in the app. For each log statement, we search the function that contains the statement and retrieve the corresponding app component, class, function and code block. The manual analysis goes one step further to uncover the leaked data.

Chapter 4 - Evaluation

4.1 Dataset Description

We applied MobiLogLeak to a dataset of a randomly collected sample of 5,000 Android applications from AndroZoo [Li 17b] (2,500 apps were published in year 2017 and the other 2,500 are from 2018). We converted their APKs to Jimple code and used Flowdroid to apply taint flow analysis. As shown in Figure 4.1, we found taint flow paths in 276 apps. We pruned paths that are not log-related by searching the sink paths for log related statements. This resulted in 200 applications (shown in Table A1 of Appendix A) with taint flow paths that have log-related sinks.

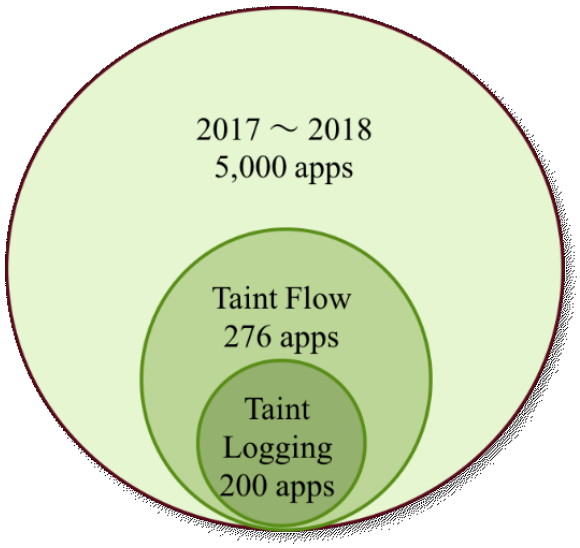


Figure 4.1 Distribution of apps based in the dataset

The resulting 200 apps include apps from various app categories including music, finance, education, fitness, etc. (as shown in Table 4-1). We found that about half of the dataset of apps with log-related taint paths (102 apps) belong to the Personalization category. These apps collect user profile information including all sort of people's personal data. They manipulate more private information than apps in other categories. For example, the app Pink Love Heart Keyboard Theme, a Personalization app, uses logs to display network information through the Broadcast component. We found that when this app receives data from other apps, this data is output through logging statement. The complete list of log-related statements of the analyzed apps is shown in Table A1 of Appendix A.

Table 4-1 Categorization of apps in the dataset with log-related sinks

Category	**Number**	**Category**	**Number**
Personalization	102 (51%)	Music & Video	4 (2%)
Entertainment	13 (7.5%)	Finance	4 (2%)
Business	8 (4%)	News & Magazines	4 (2%)
Tools	8 (4%)	Casino	2 (1%)
Communication	7 (3.5%)	Travel & Local	2 (1%)
Lifestyle	6 (3%)	Others	40 (20%)

Apps in other categories are also affected. For example, most entertainment apps require from users to register with third-party social media accounts (i.e., Facebook, Twitter, Instagram), which may lead to leakage of social media related information. The same applies to Business apps, which manipulate information related to organizations.

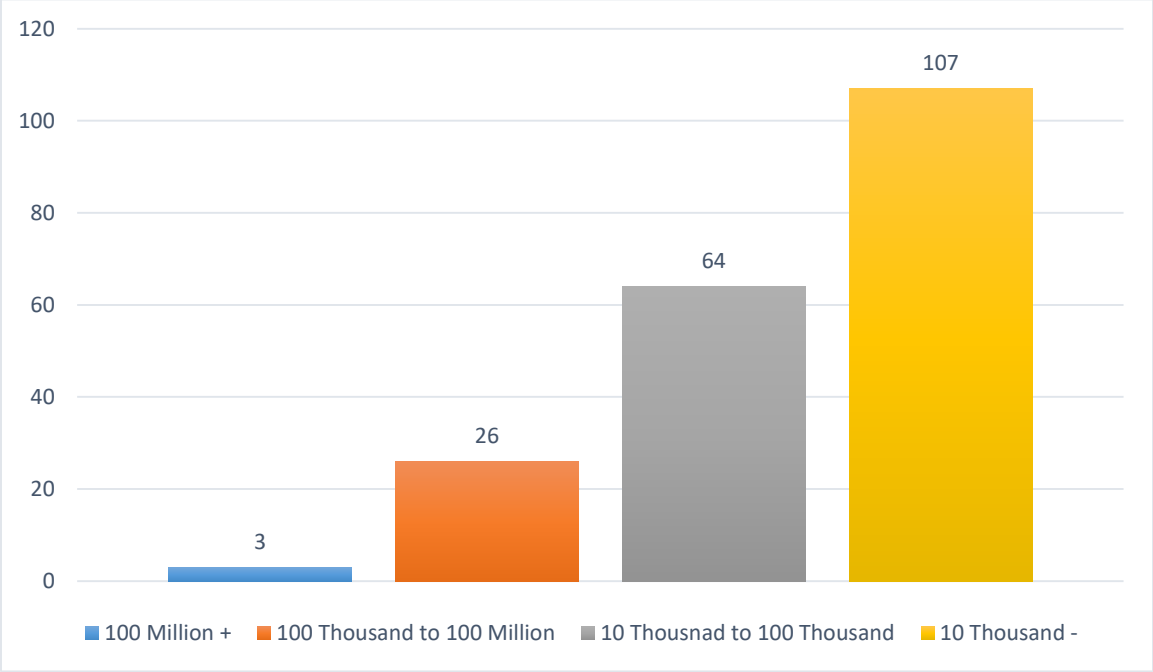


Figure 4.2 Distribution of ALRS based on the number of installations

Figure 4.2 shows the distribution of ALRS based on the number of installations. We can see that many of these applications are highly popular. In particular, three of these apps were installed more than 100 million times, meaning that even though these apps count for only 1.5% of the total ALRS, the number of impacted users can be considerably high.

In addition, about half of ALRS are installed less than 10 thousand times, suggesting that for these less popular apps, developers do not pay the needed attention to logging issues. It appears that for these apps, developers use logging for monitoring purposes, ignoring the potential threats that poor logging practices may cause.

4.2 Context Analysis Results

Table 4-2 shows that good apps have higher log density than ALRS. At first glace, this looks as a surprising result. To further explore this, we separated the data into two datasets based on the year of publication of the apps, i.e., 2017 and 2018, and recalculate the log density. The number of apps in each year is 2,500. The results were consistent with our previous finding. One possible explanation for this is that apps that are heavily logged are those written by experienced developers, who also seem to apply good logging practices, such as removing log statements that could reveal sensitive data before releasing the apps, but also use more logs for exception handling and for explaining errors.

Table 4-2 Log context analysis

Type	**SLOC**	**LLOC**	**Density**	**Construct**	
				Exception	**Reflective**
Good Apps	436,167,099	1,205,557	1/362	323,295 (26.82%)	72,867 (6.04%)
ALRS	34,686,096	64,296	1/539	15,948 (24.80%)	2,272 (3.53%)
2,500 apps in 2017	246,732,779	690,173	1/357	175,361 (25.41%)	41,994 (6.08%)
2,500 apps in 2018	224,120,416	579,680	1/387	163,882 (28,27%)	33,145 (5.71%)
Total	470,853,195	1,269,853	1/371	339,243 (26.71%)	75,139 (5.92%)

Table 4-3 shows the distributions of logging statements through Android app components. The results show that most logging occur in the Activity components, with almost no logging in the Content provider component (zero in our case). We can see that the Activity component accounts for 98.5% of all the logs in ALRS, suggesting that any further study for detecting logging statements that may leak data should focus on the Activity component. The reason is that all the interactions with users take place in Activity component such collecting the input stream, sending text to background, and showing the results.

Table 4-3 Components for each app in different types

Component	**4,800 Good Apps**	**200 ALRS**	**Total 5000 Apps**
Activity	4620 (96.25%)	197 (98.5%)	4817 (96.34%)
Service	15 (0.3125%)	0	15 (0.3%)
Broadcast Receiver	6 (0.125%)	1 (0.5%)	7 (0.14%)
Content Provider	0	0	0
Application	0	0	0

Unknown	159 (3.3125%)	2 (1%)	161 (3.22%)

As part of this analysis, we also studied if log-related sinks appear in reflective methods and exception blocks as those two constructs normally contain log statements. We found that 25% of the total log statements are in the exception blocks and 5% in the reflection methods. No poor logging (i.e., sink related log statements) was reported in those two constructs.

4.3 Taint Flow Analysis Results

In this section, we show by example, how we manually inspect each of the taint flow paths generated in Chapter 3.6 in order to categorize the log-related leakage cases based on the types of leakage.

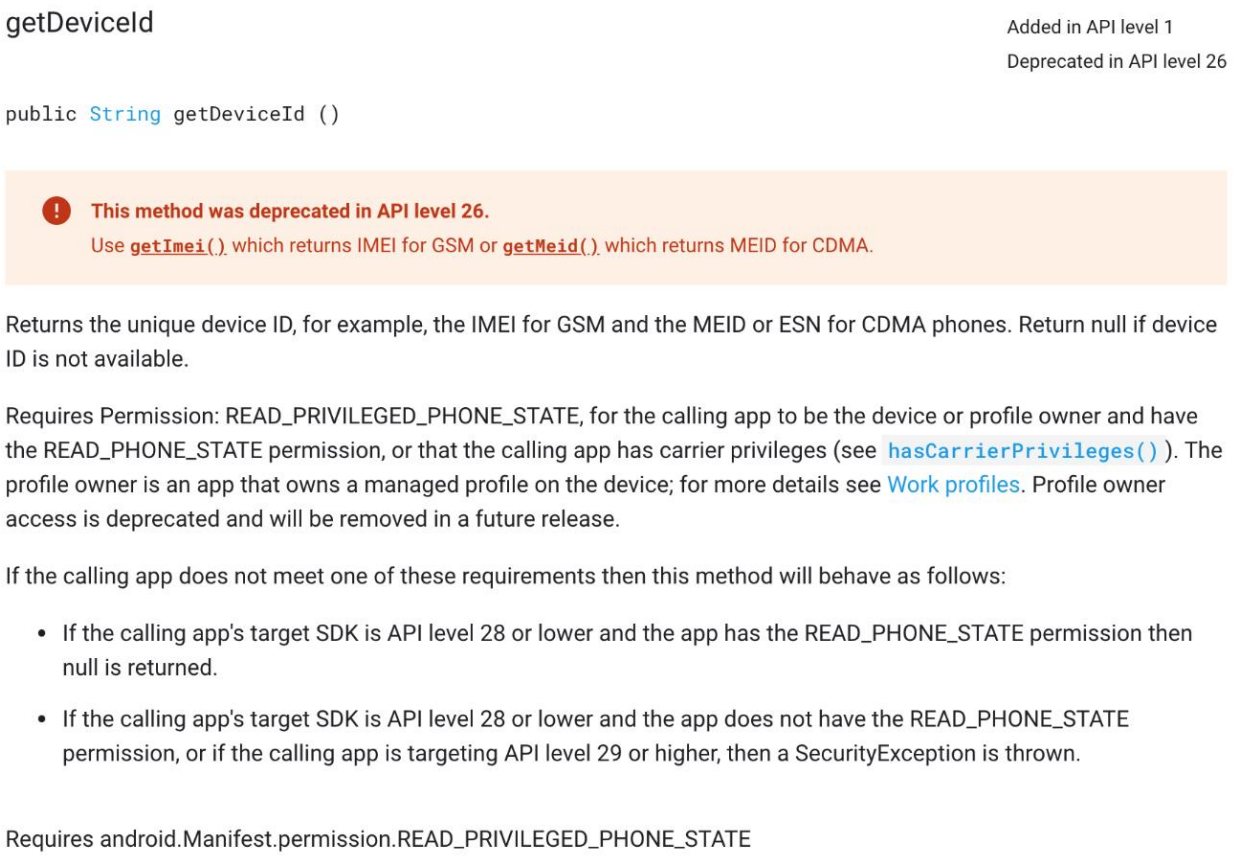
Recall that each taint flow path consists of a sink and a source. In our manual analysis, we start from the source and then we find the corresponding API that is related to that source. Using the description of the API, we can identify the actual data leaked in the corresponding taint flow path.

```
01.  public static java.lang.String i(android.content.Context)
02.      { ...
03.          label05:
04.              $r3 = virtualinvoke $r2.<android.telephony.TelephonyManager: java.lang.String getDeviceId()>();
05.          label06:
06.              $r4 = $r3;
07.              $r5 = new java.lang.StringBuilder;
08.      label07:
09.              specialinvoke $r5.<java.lang.StringBuilder: void <init>()>();
10.      $r5 = virtualinvoke $r5.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>($r3);
11.       $r5 = virtualinvoke $r5.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>("device ID");
```

```
12.     $r6 = virtualinvoke $r5.<java.lang.StringBuilder: java.lang.String toString()>();
13.     staticinvoke <android.util.Log: int d(java.lang.String,java.lang.String)>("deviceIMEI", $r6);
14.      ...
15.   }
```

Listing 6: Code Snippet for Case Network Device ID

For example, in Listing 6 (the complete code is in Listing B1), the method getDeviceId() in Line 4 is a source that is related to the log-related sink that leaks the device IMEI in Line 13. To obtain the API that corresponds to the source, we can look up the method "getDeviceId" in the list of APIs provided by the Android software development kit (SDK).



getDeviceId

Added in API level 1
Deprecated in API level 26

```
public String getDeviceId ()
```

This method was deprecated in API level 26.
Use `getImei()` which returns IMEI for GSM or `getMeid()` which returns MEID for CDMA.

Returns the unique device ID, for example, the IMEI for GSM and the MEID or ESN for CDMA phones. Return null if device ID is not available.

Requires Permission: READ_PRIVILEGED_PHONE_STATE, for the calling app to be the device or profile owner and have the READ_PHONE_STATE permission, or that the calling app has carrier privileges (see `hasCarrierPrivileges()`). The profile owner is an app that owns a managed profile on the device; for more details see Work profiles. Profile owner access is deprecated and will be removed in a future release.

If the calling app does not meet one of these requirements then this method will behave as follows:

- If the calling app's target SDK is API level 28 or lower and the app has the READ_PHONE_STATE permission then null is returned.
- If the calling app's target SDK is API level 28 or lower and the app does not have the READ_PHONE_STATE permission, or if the calling app is targeting API level 29 or higher, then a SecurityException is thrown.

Requires android.Manifest.permission.READ_PRIVILEGED_PHONE_STATE

Figure 4.3 Android API – Network: getDeviceId

Figure 4.3 shows the description of the API and the actual data retrieved by calling getDeviceID, which will be then leaked through the log statement at the sink. In this case, it is the IMEI for GSM. Based on the API description, we can categorize the leakage into one of the following four categories.

- Network, including Mac address, Device Id Sim Serial Number, country and package manager.
- Account, including name, token, password and type of the account owner.
- Location, including latitude, longitude, and last-known location.
- Database, including ID, password, subdomain, website link name, etc.

For the previous example, this data leakage is related to Network.

```
01. public static java.lang.String f(android.content.Context)
02.   {
03.       ...
04.       $r4 = $r5.<android.accounts.Account: java.lang.String name>;
05.       $r6 = $r5.<android.accounts.Account: java.lang.String type>;
06.       ...
07.       $r3 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>("Emails: ";
08.       $r3 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>($r4);
09.       $r6 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.String toString()>();
10.   staticinvoke <android.util.Log: int e(java.lang.String,java.lang.String)>("PIKLOG", $r6);
11.       ...
12.   }
```

Listing 7: Code Snippet for Case Account Name

Account is another default object of Android. It represents an Account in the AccountManager. This object is parcelable and also overrides equals(Object) and hashCode(), making it suitable for use as the key of a Map. There is one example shown in the following Listing 7 (the complete code is in Listing B2). As shown in Line 04, $r4 gets the account list and takes the value, name of account. From Android API shown in Figure 4.4, we check the class “android.accounts.Account”.

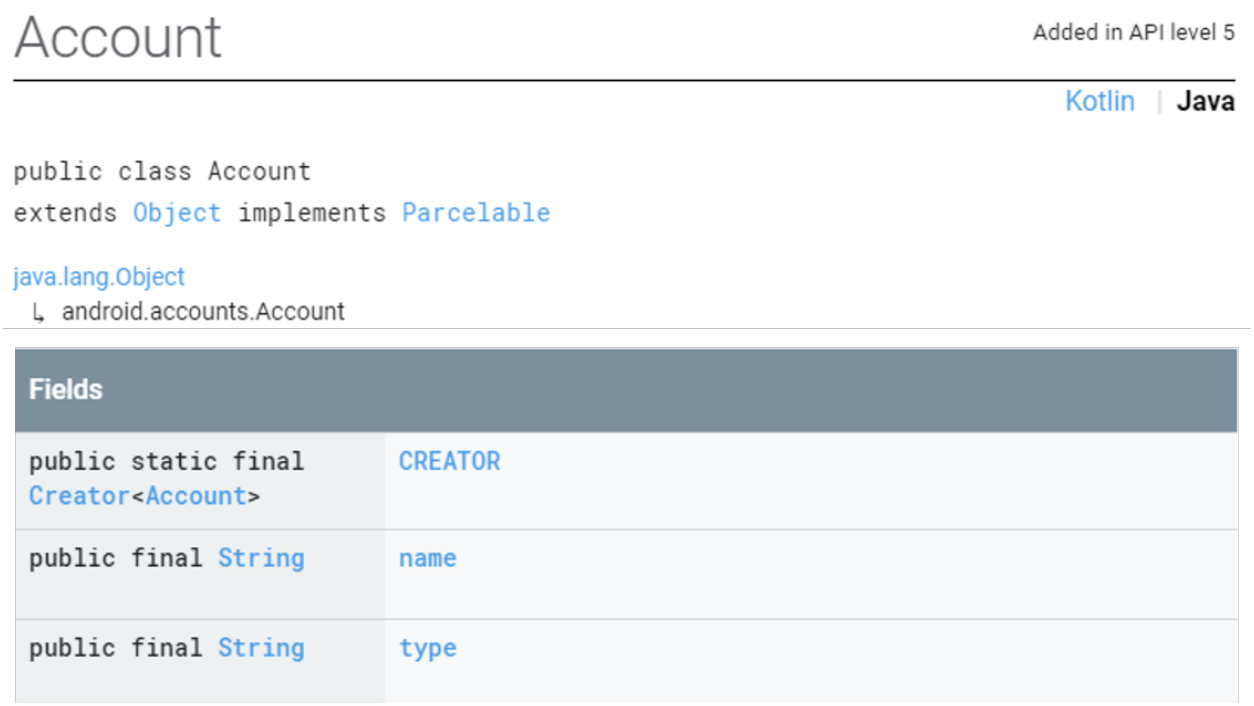


Figure 4.4 Android API - Account

Then, in Line 08 of Listing 7, it appends this data into the String $r3 and in line 10, this statement uses error function of logging to show it. According to the static text in Line 07, the name is the “Emails”. Then, we can confirm the apps store the “email” as name and this code releases the email here.

Location is a data class representing a geographic location. A location can consist of a latitude, longitude, timestamp, and other information such as bearing, altitude and velocity. All locations

generated by the LocationManager are guaranteed to have a valid latitude, longitude, and timestamp (both UTC time and elapsed real-time since boot), all other parameters are optional.

```
1.  public android.location.Location a()
2.      {
3.       ...
4.      label12:
5.        $r0.<net.intricare.gobrowserkiosklockdown.services.f: android.location.Location d> = $r5;
6.        $r5 = $r0.<net.intricare.gobrowserkiosklockdown.services.f: android.location.Location d>;
7.        if $r5 == null goto label18;
8.      label13:
9.        $r5 = $r0.<net.intricare.gobrowserkiosklockdown.services.f: android.location.Location d>;
10.       $d0 = virtualinvoke $r5.<android.location.Location: double getLatitude()>();
11.     label14:
12.       $r0.<net.intricare.gobrowserkiosklockdown.services.f: double e> = $d0;
13.     label15:
14.       $r5 = $r0.<net.intricare.gobrowserkiosklockdown.services.f: android.location.Location d>;
15.       $d0 = virtualinvoke $r5.<android.location.Location: double getLongitude()>();
16.     label16:
17.       $r0.<net.intricare.gobrowserkiosklockdown.services.f: double f> = $d0;
18.     label17:
19.       $r6 = new java.lang.StringBuilder;
20.       specialinvoke $r6.<java.lang.StringBuilder: void <init>()>();
21.       $r6 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>("latit
    ude [");
22.       $d0 = $r0.<net.intricare.gobrowserkiosklockdown.services.f: double e>;
23.       $r6 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.StringBuilder append(double)>($d0);
24.       $r6 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>(" ]");
25.       $r7 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.String toString()>();
26.       staticinvoke <android.util.Log: int d(java.lang.String,java.lang.String)>("Lac", $r7);
```

```
27.        $r6 = new java.lang.StringBuilder;
28.        specialinvoke $r6.<java.lang.StringBuilder: void <init>()>();
29.        $r6 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>("lon
    gitude [");
30.        $d0 = $r0.<net.intricare.gobrowserkiosklockdown.services.f: double f>;
31.        $r6 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.StringBuilder append(double)>($d0);
32.        $r6 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.StringBuilder append(java.lang.String)>(" ]");


33.        $r7 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.String toString()>();
34.        staticinvoke <android.util.Log: int d(java.lang.String,java.lang.String)>("Lac", $r7);
35.      ...
36.  }
```

Listing 8: Code Snippet for Case Location latitude and longitude

We have one example shown in the Listing 8 (the complete code is in Listing B3). Similar to the other categories, in Line 15, it gets the longitude data and assigns it to $d0. Then, it appends this data to String $r6 in Line 31, and converts it into String $r7 in Line 33. Finally, the program uses debug function of logging to show it in Line 34.

getLatitude

Added in API level 1

```
public double getLatitude ()
```

Get the latitude, in degrees.

All locations generated by the `LocationManager` will have a valid latitude.

Returns
double

Figure 4.5 Android API – Location getLatitude

Figure 4.6 Android API – Location getLongitude

Figure 4.5 and Figure 4.6 indicate the Android API for "getLatitude" and "getLongtitude". These two functions work for the "LocationManager".

When we install an Android app, the system asks the user for permission to access location data. For personal security or other reasons, we can deny the app from getting this permission. However, even if we forbid an app to access the location, the apps could access the log file of other apps to get our location information, which circumvents Android permission mechanism.

The process described so far can help in retrieving the corresponding API of a source (hence the actual data) where the leakage is related to network, account, or location type. Unfortunately, this process may not work in complex situations, such as in the case of the database leakage type. In such scenarios, the path from the source to the sink is normally long and can take several alternatives. Moreover, the automatically identified source (by Flowdroid) does not provide us with enough information about the real source of the data that is leaked.

```
01. private boolean a(java.lang.String)
02.     {
03.         ...
04.         $r3 = specialinvoke $r0.<net.intricare.gobrowserkiosklockdown.firebasenotify.MyFirebaseMessagingServic
          e: java.lang.String a()>();
05.         ...
06.         staticinvoke <android.util.Log: int i(java.lang.String,java.lang.String)>("MyFirebaseMsgService", $r3);
07.         staticinvoke <android.util.Log: int i(java.lang.String,java.lang.String)>("MyFirebaseMsgService", "oldpassw
          ord and newpassword matches. save new password");
08.         ...
09.     }
10. ----------------------------------------------------------------------------------------
11. private java.lang.String a()
12.     {
13.         ...
14.         $r1 = virtualinvoke $r2.<net.intricare.gobrowserkiosklockdown.b.b: java.lang.String e()>();
15.         return $r1;
16.     }
17. ----------------------------------------------------------------------------------------
18. public java.lang.String e()
19.     {
20.     ...
21.         $i0 = interfaceinvoke $r3.<android.database.Cursor: int getColumnIndex(java.lang.String)>("password");
22.         $r4 = interfaceinvoke $r3.<android.database.Cursor: java.lang.String getString(int)>($i0);
23.         return $r4;
24.         ...
25.     }
```

Listing 9: Code Snippet for Case Database Password

Hence, the analysis normally includes tracking the real source. For example, in Listing 9 (the complete code is in Listing B4), starting from the source in Line 22, Flowdroid pointed to a source for a database leakage. In order to identify the actual leaked data, a thorough analysis is required starting from the log statement at the sink to the actual source. Once we identify the real source, based on the parameters, we can find which query the source used and what are the fields in that query (shown in Figure 4.7)

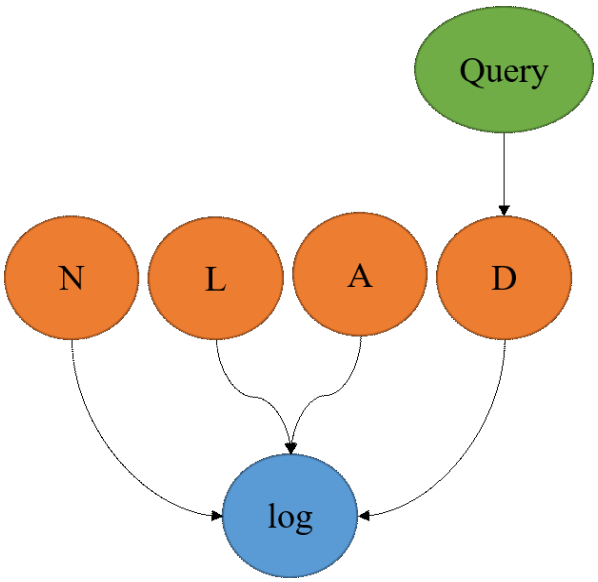
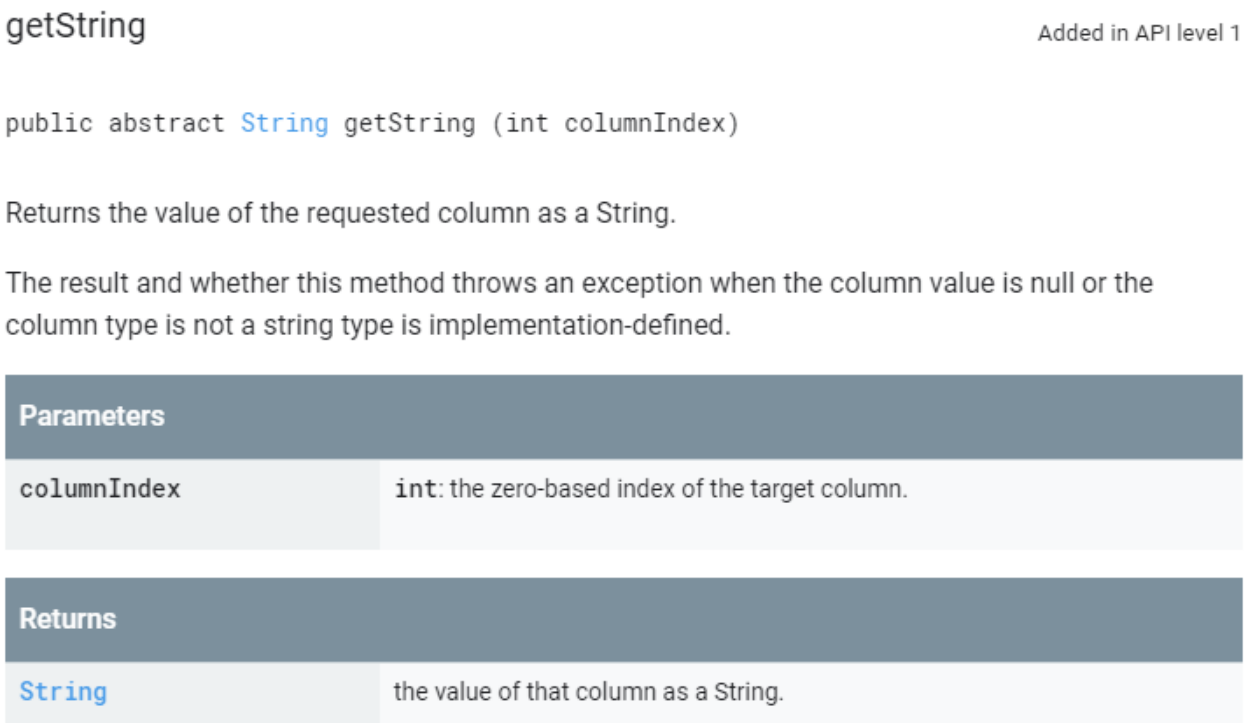


Figure 4.7 ALRS Apps Source Categories

For example, in Listing 9, to track the data and confirm the real path, we start from the sink in Line 06 in Listing 9. It is the log statement that contains the variable $r3. We go back and look for the value of $r3 and in Line 04, we find it is from the function, private Boolean a(java.lang.String). Then, we go to this function and check it. We know the final return value is $r1 in Line 15. It comes from the function 'public java.lang.String e()' in Line 14. Then we trace this function shown in Line 18. In the function, public java.lang.String e(), we could find the final return value is $r4.

Based on FlowDroid, the source is in Line 22. From Figure 4.8, we can relate this statement to the function “getString”, which returns the value of that column.



getString

Added in API level 1

```
public abstract String getString (int columnIndex)
```

Returns the value of the requested column as a String.

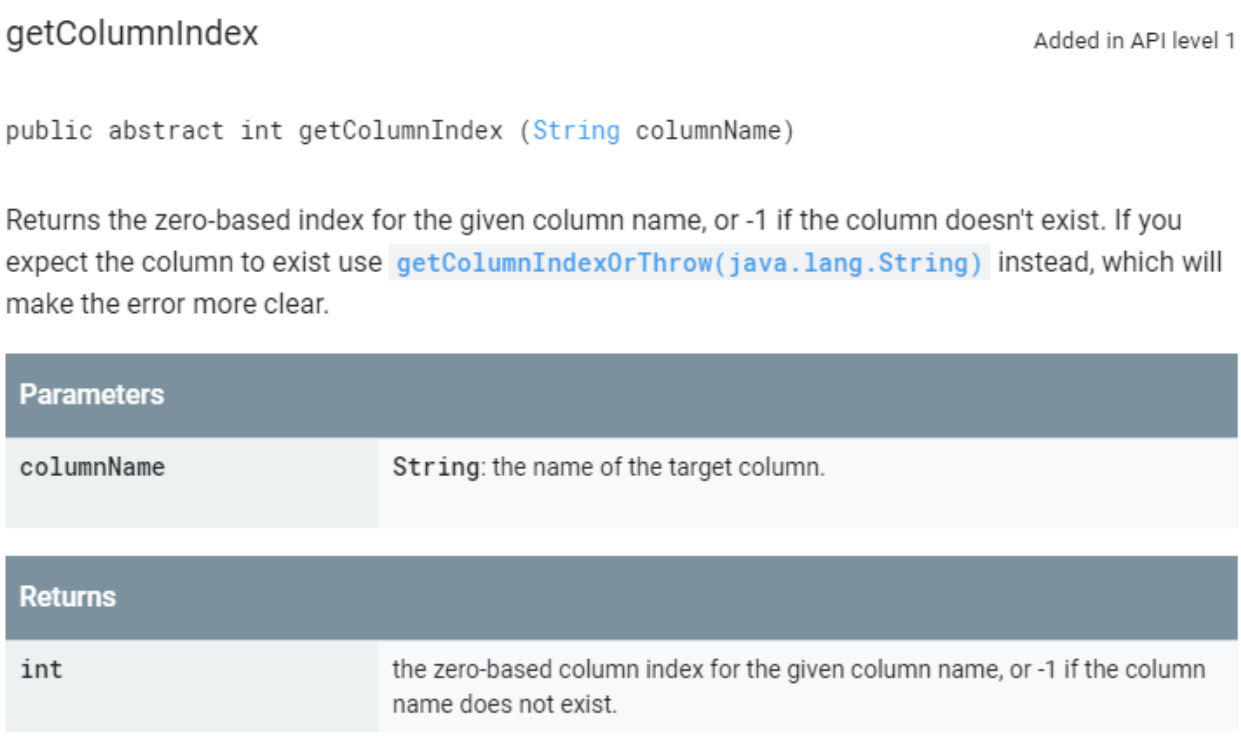
The result and whether this method throws an exception when the column value is null or the column type is not a string type is implementation-defined.

Parameters	
columnIndex	int: the zero-based index of the target column.

Returns	
String	the value of that column as a String.

Figure 4.8 Android API - Database getString

However, if we stop here, we can only know the data from the database. We need to go further to identify the real source and the value from the database that is leaked. This information cannot be inferred from getString(int)($i0). To solve this, we read this function and check it again. Luckily, we find the $r4 is from the query operation, getColumnIndex(java.lang.String)>("password") in Line 21. As shown in Figure 4.9, the function, getColumnIndex returns the index of the given variable. In the code, the given variable is password, so this statement returns the index of password and uses the index to get its value in Line 22 of Listing 9.



getColumnIndex

Added in API level 1

```
public abstract int getColumnIndex (String columnName)
```

Returns the zero-based index for the given column name, or -1 if the column doesn't exist. If you expect the column to exist use `getColumnIndexOrThrow(java.lang.String)` instead, which will make the error more clear.

Parameters	
`columnName`	`String`: the name of the target column.

Returns	
`int`	the zero-based column index for the given column name, or -1 if the column name does not exist.

Figure 4.9 Android API - Database getColumnIndex

In this case, the source is transformed into two other functions, and we can check the static text of logging in Line 07. It is “oldpassword and newpassword matches. save new password”. It means this logging happens during the change of passwords. As we mentioned before, most taint flows are likely caused by developers who use log for debugging during development but forget to delete this type of log statement before deployment.

We applied this manual analysis to the 200 apps with log-related sinks (i.e., ALRS). In total, there were 380 sources with elements of sensitive data, and 293 sinks with log statements related to the data elements in the source. Out of the 380 sources, 186 (49%) leaked network sensitive data, 170 (45%) leaked database sensitive data, 22 (5%) leaked location sensitive data and 2 (0.5%) sources leaked user account data.

These results, although preliminary, clearly demonstrate that logging, when not used carefully, may lead to the leakage of sensitive information. There is a need to raise awareness around this topic and start developing logging guidelines to prevent these situations. We suspect that the cases we found are caused by developers who may have needed these logs during development, but omitted to remove them before releasing the apps. We need to dig further to understand (1) the scale of this problem, and (2) the causes.

4.4 Threats to validity

Internal Validity: We manually analyzed all the taint flow analysis paths that contain log statements as sinks. Three authors checked the results. Because this is done manually, errors may have occurred, which we recognize as a threat to internal validity. Another threat is related to the selection of the 5,000 apps. We selected these apps randomly, but a different set may lead to different results.

External Validity: Software engineering studies suffer from the variability of the real world, and the generalization problem cannot be solved completely. Although we have used 5,000 apps in this study (to reduce the risk of insufficient generalization), our evaluation remains preliminary and should be qualified as an early research, and may not be generalizable to other apps.

4.5 Limitations

In our dataset, some of them cannot be analyzed by soot, the exceptions of building graph failed. This is attributed to the limitations of static analysis. Flowdroid is known to be unable to analyze some apps due to various reasons (e.g., corrupted DEX code, missing app assets, obfuscated code, use of special features that are not accounted for by the analysis, etc.)

Flowdroid and droidfax are based on Soot. During the experiment, we have three kinds of apps. The first group works well with Soot. They are built the process graph and return the amount for the method categories. Second group just throws the exception like java.lang.AssertionError but it also gives me the result. For this group, we ignore the exception. Obviously, the exception appears after the result comes out. For the third group, it just gives me one exception like NonePointException. This group will influence our accuracy, so when we calculate the percentage, we will ignore this group.

Also, for the taint flows, some apps use query instead of getcolumn to show the data. For these applications, we cannot know what kind of information is leaked. For example,

```
$r9 = interfaceinvoke $r15.<android.database.Cursor: java.lang.String getString(int)>(1)
```

It checks the index of the column so we cannot know the column name. It depends on the structure of the database and each record returned by the query operation.

Chapter 5 - Conclusion and Future Work

5.1 Research Contributions

In this study, we investigated the impact of logging practices on data leakage. Particularly, we explored how common are poor logging practices in mobile applications and the effect of not removing logs related to sensitive information before releasing the application. Our preliminary results show that log statements are common in the released mobile applications. There is one log statement in each 362 lines of code. Not all these logs are a result of bad logging practices. For example, in our study on 5,000 mobile applications, none of the log statements in the exception blocks leaked sensitive data. On the other hand, poor logging practices are also common in mobile applications. Out of 276 apps with taint flows, 200or around 72% leaked sensitive data due to poor logging practices. We categorized the data leakages that are related to logging practices into four types. The results demonstrate that in the 200 apps that we manually analyzed, there were 380sources of data leakage, 186 of these sources leaked network sensitive data, 170 leaked database sensitive data, 22 leaked location sensitive data and 2 sources leaked user account data.

5.2 Opportunities for Further Research

Our preliminary results suggest a correlation between the context of logging practices and whether these practices are good or poor, further investigation is required as part of future work. Another interesting future direction is to automatically classify the leaked data in the categories network, account, location, and database.

For the callback functions, we need to improve our method for analyzing them because they are always used when systems want to manage the app lifecycle. Also, we should continue examining log statements that appear in exception blocks, used by developers to uncover the root causes of defects, to prevent situations where logged variables may adversely leak sensitive data.

Moreover, we should continue experimenting with more apps, including commercial apps, from various categories to understand the scope of this problem and to design techniques for preventing it. Priority should be given to popular apps since they can affect a large number of users.

In addition, we should develop automated techniques to detect potential data leaks in existing log data. However, the large size of typical log data may render this task very difficult. Tools for achieving this should be equipped with some sort of log abstraction techniques (such as the ones surveyed by El-Masri et al. [El-Masri 2020]) in order to reduce the large size of log data while keeping the main information. There exist also several abstraction techniques used in tracing such as the ones proposed by Hamou-Lhadj et al. [Hamou-Lhadj 2002, Hamou-Lhadj 2006] that can be readily applied to the abstraction of large streams of log data.

Finally, we should study whether the use of logging can be reduced by resorting to other dynamic analysis techniques such as tracing of program flows and program profiling. These approaches do not require as much user input as logging, which may reduce errors related to data leakage.

Appendix A. Results of the experiments

Table A1. 200 Taint Apps Dataset

Name	Installations	Label
CM Launcher 3D - Themes, Wallpapers	100,000,000	Personalization
Power Clean - Antivirus & Phone Cleaner App	100,000,000	Tools
Bit Clean	100,000,000	Tools
CM Browser - Ad Blocker, Fast Download, Privacy 5.22.21.0051	50,000,000	Communication
Opera News	50,000,000	News & Magazines
Emoji Keyboard - Cute Emoji, GIF, Sticker, Emoticon	50,000,000	Tools
Starbucks	10,000,000	Food & Drink
Moto File Manager	10,000,000	Tools
ESPNCricinfo - Live Cricket Scores, News & Videos	10,000,000	Sports
Cool Black Theme	10,000,000	Entertainment
Taichi Panda	10,000,000	Role Playing
Opera Mini browser beta	10,000,000	Communication
Gradeup: Exam Preparation App | Free Mocks | Class	5,000,000	Education
DdcatApp	1,450,000	Entertainment
Anime Wallpaper Master	1,000,000	Personalization
Say caller name	1,000,000	Tools
SignEasy | Sign and Fill PDF and other Documents	1,000,000	Business

Pink Glitter Bow Keyboard Theme	1,000,000	Personalization
Rummy 45 - Remi Etalat	1,000,000	Card
Alive In Shelter	1,000,000	Strategy
Facemoji Keyboard Lite: GIF, Emoji, DIY Theme	1,000,000	Personalization
Pixel Doodle: Color by Number, Pixel Art, Color Game	1,000,000	Video Editor & Video Maker Dev
God christ keyboard	1,000,000	Personalization
Lucky Slots - Free Casino Game	1,000,000	Casino
Messaging+ SMS, MMS Free	1,000,000	Communication
Color shiny rose theme keyboard	1,000,000	Personalization
Pink Love Heart Keyboard Theme	1,000,000	Personalization
Messaging+ SMS, MMS Free	1,000,000	Communication
Summer GO Keyboard Theme	1,000,000	Personalization
Fairy live wallpaper	500,000	Personalization
Silk Gold Icons Theme	500,000	Personalization
Black Cool Wolf King Theme	500,000	Personalization
ForteBank	500,000	Finance
قصص الصغار	500,000	Entertainment
Golden Fidget Spinner Theme	500,000	Personalization
Equalizer Pro - Extra Sound	500,000	Music & Audio
Indian Girls Photo Maker	500,000	Photography
Arab Man Fashion Photo Suit	500,000	Photography

Custom Wallpaper Maker FREE	100,000	Tools
Graffiti Skull Smoke keyboard	100,000	Personalization
Cute Cartoon Owl Bowknot Theme	100,000	Personalization
Neon Butterfly Live Wallpaper	100,000	Personalisation
Gold Luxury Car Theme	100,000	Personalization
Cute Girl Theme	100,000	Lifestyle
Sketched Street View	100,000	Personalization
Theme for Galaxy S9 Plus	100,000	Personalization
Crimson Horrific White Eyes Theme	100,000	Personalization
Gun Man Photo Montage	100,000	Entertainment
The Purple Fantasy Wonderland Theme	100,000	Personalization
Calisteniapp - Calisthenics & Street Workout	100,000	Health & Fitness
Krishna Flute Ringtones	100,000	Music & Audio
Pink Minny Diamond keyboard	100,000	Personalization
BankSA Mobile Banking	100,000	Finance
Black Gold Business Theme	100,000	Personalization
Hijau Gothic Logam Coretan Tengkorak Tema	100,000	Personalization
Tavla	100,000	Board
Pink Rose Black Lace Theme	100,000	Personalization
Heathrow Express	100,000	Travel & Local
Shiny Neon Love Launcher	100,000	Personalization
Citations de La Vie	100,000	Entertainment

Dhoka Shayari	100,000	Social
Glass Water Drop Keyboard	100,000	Personalisation
Panda Sakura Theme	100,000	Personalization
Goal Achiever's Multitool - Goalist	100,000	Productivity
Motobike Sport Thème	100,000	Personalisation
Dard Shayari	100,000	Entertainment
Subway Bus Racer	50,000	Racing
Pink Love Butterfly Keyboard	50,000	Personalization
Crystal Rose Love 3D Theme	50,000	Personalization
com.hld.anzenbokusu	50,000	Tools
Tattoo Rose Romantic Wolf Theme	50,000	Personalization
Space Planet 3D Earth Theme	50,000	Lifestyle
Romantic Red Love Heart Theme	50,000	Personalization
cute skull icon pink bow theme	50,000	Personalization
Typewriter Keyboard	50,000	Personalization
Blue Flames Keyboard Theme	50,000	Personalization
Purple Luxury Golden Butterfly Theme	50,000	Personalization
Pink Cute Flower Rose Red Petals Keyboard Theme	50,000	Personalisation
3D Green Maple Leaf	50,000	Personalization
悟空问答	20,000	Communication
HTC Yellow Pages	10,000	Business
Lavender water drop keyboard theme	10,000	Personalization

Neon Koi Fish Space 3D Theme	10,000	Personalization
Thermal Camera Filter Effect Flashlight	10,000	Entertainment
Blackboard Graffiti Theme	10,000	Personalization
Water Fire Fist Battle Keyboard Theme	10,000	Personalization
Movilizer	10,000	Business
Spoint	10,000	Lifestyle
Butterfly Fairy Nature Theme	10,000	Personalization
Girly Paris keyboard theme	10,000	Personalization
Glitter Blue Dream Theme - glitter wallpaper	10,000	Personalization
Weed Ghost Gun Launcher Theme	10,000	Personalization
Tumbi	10,000	Music & Audio
Blue Glitter Cute Panda Keyboard	10,000	Personalization
Animated Cute Pink Hearts Keyboard	10,000	Beauty
Турецкий для влюблённых (DEMO)	10,000	Travel & Local
Orange Cartoon Cute Lazy Cat Theme	10,000	Personalization
Pink Black Kitty Love Theme	10,000	Lifestyle
Black Business Gold Theme	10,000	Personalization
Judai Shayari	10,000	Social
SuperSMS - Text Messages	10,000	Communication
Fire Lion Theme	10,000	Personalisation
Cute Anime Girl 3D	10,000	Personalisation
Cute Dog Love Theme	5,000	Lifestyle

Citations de Nicolas Michiavel	5,000	Education
Bubble Spinner Deluxe	5,000	Casual
Cute Snow Penguin Baby Theme	5,000	Personalization
Black Skull Keyboard Theme	5,000	Personalization
Cute Panda Girl Theme	5,000	Personalization
Consumer Protection Act	5,000	Books & Reference
Cute Unicorn Whale	5,000	Personalization
Bullet Gun Theme	5,000	Personalization
Cool Speedy Racing car keyboard	5,000	Personalization
My BVF	1,000	House & Home
Electronic Music DJ Mellow Theme	1,000	Entertainment
Diamond pink leopard theme	1,000	Personalisation
GGM VIEW	1,000	Business
Christmas Gift Tree Gravity Theme	1,000	Personalization
Greenback Slots – Big Win	1,000	Casino
Therapeutic Lucky Koi Fish Satisfaction Theme	1,000	Personalization
Omni	1,000	Education
Pink SMS Keyboard Theme Diamond Ribbon	1,000	Beauty
Hot Flame Evil Skull Keyboard Theme	1,000	Entertainment
Gold Sequins Flip Keyboard Theme	1,000	Personalization
3D Galaxy Earth Moon Parallax Theme	1,000	Personalization
Kiosk Browser Lockdown	1,000	Business

Colorful Abstract Simple Theme	1,000	Personalization
Panda Unicorn Star Galaxy Keyboard Theme	1,000	Personalization
Gold Christmas Theme Wallpaper	1,000	Personalization
Glow Neon Wolf Theme	1,000	Personalization
Dark Neon Panther Theme	1,000	Personalization
Multi Level Car Parking-Crazy Driving School	1,000	Simulation
Cute Bear Keyboard Theme	1,000	Personalization
Colorful Water Drop Flower Theme	1,000	Personalization
Green Football Pitch Theme	1,000	Personalization
Angry Flame Tiger Keyboard Theme	1,000	Personalization
Golden Effiel Tower Keyboard Theme	1,000	Personalization
TWICE WALLPAPER 2018	1,000	Personalization
Graffiti Theme Street Art	1,000	Personalisation
Gold Luxury Car Theme	1,000	Personalisation
Twinkle Gemstone Theme	1,000	Personalisation
The Konnected	500	Entertainment
Golden Green Butterfly Theme	500	Personalization
Nellore	500	News & Magazines
Cool Rain Glass Waterdrop Theme	500	Personalization
Tech Skull Red Live Keyboard	500	Personalization
Horror Bloody Bat Theme	500	Personalization
Snake Color Box Keyboard Theme	500	Personalization

Father Chritsmas Elk Theme	500	Personalization
Purple Neon Business Theme	500	Personalization
Modern Trade SFA- PepUpSales	500	Business
New Year Pink Kitty Theme	100	Personalization
Hoshiarpur	100	News & Magazines
Brunet Swan Love keyboard	100	Personalization
Radio Miel de Dios	100	Communication
VP Tandem	100	Business
Mandla	100	News & Magazines
Fantasy Flowers Live Wallpaper	100	Personalization
Sunlit Flowery Day Theme	100	Personalization
Tiny Bowman Hero: Archery Rescue Challenge	100	Comics
Cutie Bear Heart keyboard	100	Personalisation
BIGG B	50	Music & Audio
蒲公英	50	Photography
Fast Mountain Train Driver : Train Simulator 2018	50	game
고창동막골농장	5	Shopping

Appendix B. Code Snippet for Cases

Listing B1. Network

```
1.          Class: net.intricare.gobrowserkiosklockdown.util.aa
2.          public static java.lang.String i(android.content.Context)
3.          {
4.              android.content.Context $r0;
5.              java.lang.Object $r1;
6.              android.telephony.TelephonyManager $r2;
7.              int $i0;
8.              java.lang.String $r3, $r4, $r6;
9.              java.lang.StringBuilder $r5;
10.             java.lang.Throwable $r7, $r8;
11.
12.             $r0 := @parameter0: android.content.Context;
13.
14.          label01:
15.             $r1 = virtualinvoke $r0.<android.content.Context: java.lang.Object
        getSystemService(java.lang.String)>("phone");
16.
17.          label02:
18.             $r2 = (android.telephony.TelephonyManager) $r1;
19.
20.          label03:
21.             $i0 = staticinvoke <android.support.v4.app.a: int
        a(android.content.Context,java.lang.String)>($r0,
        "android.permission.READ_PHONE_STATE");
22.
```

```
23.           label04:
24.              if $i0 == 0 goto label05;
25.
26.              return "";
27.
28.           label05:
29.              $r3 = virtualinvoke $r2.<android.telephony.TelephonyManager: java.lang.String
         getDeviceId()>();
30.
31.           label06:
32.              $r4 = $r3;
33.
34.              $r5 = new java.lang.StringBuilder;
35.
36.           label07:
37.              specialinvoke $r5.<java.lang.StringBuilder: void <init>()>();
38.
39.              $r5 = virtualinvoke $r5.<java.lang.StringBuilder: java.lang.StringBuilder
         append(java.lang.String)>($r3);
40.
41.              $r5 = virtualinvoke $r5.<java.lang.StringBuilder: java.lang.StringBuilder
         append(java.lang.String)>("device ID");
42.
43.              $r6 = virtualinvoke $r5.<java.lang.StringBuilder: java.lang.String
         toString()>();
44.
45.              staticinvoke <android.util.Log: int
         d(java.lang.String,java.lang.String)>("deviceIMEI", $r6);
46.
47.           label08:
```

```
48.              return $r3;
49.
50.           label09:
51.              $r7 := @caughtexception;
52.
53.           label10:
54.              virtualinvoke $r7.<java.lang.Exception: void printStackTrace()>();
55.
56.              return $r4;
57.
58.           label11:
59.              $r8 := @caughtexception;
60.
61.              $r4 = "";
62.
63.              $r7 = $r8;
64.
65.              goto label10;
66.
67.              catch java.lang.Exception from label01 to label02 with label11;
68.              catch java.lang.Exception from label03 to label04 with label11;
69.              catch java.lang.Exception from label05 to label06 with label11;
70.              catch java.lang.Exception from label07 to label08 with label09;
71.          }
```

Listing B2. Account

```
1.      Class: net.intricare.gobrowserkiosklockdown.util.aa
2.          public static java.lang.String f(android.content.Context)
3.          {
4.              android.content.Context $r0;
5.              android.accounts.AccountManager $r1;
6.              android.accounts.Account[] $r2;
7.              java.lang.StringBuilder $r3;
8.              int $i0, $i1;
9.              java.lang.String $r4, $r6;
10.             android.accounts.Account $r5;
11.             boolean $z0;
12.             java.lang.Throwable $r7, $r8;
13.
14.             $r0 := @parameter0: android.content.Context;
15.
16.          label01:
17.             $r1 = staticinvoke <android.accounts.AccountManager:
                android.accounts.AccountManager get(android.content.Context)>($r0);
18.
19.             $r2 = virtualinvoke $r1.<android.accounts.AccountManager:
                android.accounts.Account[] getAccounts()>();
20.
21.          label02:
22.             $r3 = new java.lang.StringBuilder;
23.
24.          label03:
25.             specialinvoke $r3.<java.lang.StringBuilder: void <init>()>();
26.
```

```
27.              $r3 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.StringBuilder
              append(java.lang.String)>("Size: ");
28.
29.           label04:
30.              $i0 = lengthof $r2;
31.
32.           label05:
33.              $r3 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.StringBuilder
              append(int)>($i0);
34.
35.              $r4 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.String
              toString()>();
36.
37.              staticinvoke <android.util.Log: int
              e(java.lang.String,java.lang.String)>("PIKLOG", $r4);
38.
39.           label06:
40.              $i0 = lengthof $r2;
41.
42.              $i1 = 0;
43.
44.           label07:
45.              if $i1 >= $i0 goto label16;
46.
47.              $r5 = $r2[$i1];
48.
49.              $r4 = $r5.<android.accounts.Account: java.lang.String name>;
50.
51.              $r6 = $r5.<android.accounts.Account: java.lang.String type>;
52.
```

```
53.            label08:
54.               $z0 = virtualinvoke $r6.<java.lang.String: boolean
               equals(java.lang.Object)>("com.google");
55.
56.            label09:
57.               if $z0 == 0 goto label12;
58.
59.               $r3 = new java.lang.StringBuilder;
60.
61.            label10:
62.               specialinvoke $r3.<java.lang.StringBuilder: void <init>()>();
63.
64.               $r3 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.StringBuilder
               append(java.lang.String)>("Emails: ");
65.
66.               $r3 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.StringBuilder
               append(java.lang.String)>($r4);
67.
68.               $r6 = virtualinvoke $r3.<java.lang.StringBuilder: java.lang.String
               toString()>();
69.
70.               staticinvoke <android.util.Log: int
               e(java.lang.String,java.lang.String)>("PIKLOG", $r6);
71.
72.            label11:
73.               return $r4;
74.
75.            label12:
76.               $i1 = $i1 + 1;
77.
```

```
78.              goto label07;
79.
80.           label13:
81.              $r7 := @caughtexception;
82.
83.              $r4 = null;
84.
85.              $r8 = $r7;
86.
87.           label14:
88.              virtualinvoke $r8.<java.lang.Exception: void printStackTrace()>();
89.
90.              return $r4;
91.
92.           label15:
93.              $r8 := @caughtexception;
94.
95.              goto label14;
96.
97.           label16:
98.              return null;
99.
100.             catch java.lang.Exception from label01 to label02 with label13;
101.             catch java.lang.Exception from label03 to label04 with label13;
102.             catch java.lang.Exception from label05 to label06 with label13;
103.             catch java.lang.Exception from label08 to label09 with label13;
104.             catch java.lang.Exception from label10 to label11 with label15;
105.          }
```

Listing B3. Location

```
1.      Class: net.intricare.gobrowserkiosklockdown.services.f
2.          public android.location.Location a()
3.          {
4.              net.intricare.gobrowserkiosklockdown.services.f $r0;
5.              android.content.Context $r1;
6.              java.lang.Object $r2;
7.              android.location.LocationManager $r3;
8.              boolean $z0;
9.              android.os.Looper $r4;
10.             android.location.Location $r5;
11.             double $d0;
12.             java.lang.StringBuilder $r6;
13.             java.lang.String $r7;
14.             java.lang.Throwable $r8, $r9, $r10;
15.
16.             $r0 := @this: net.intricare.gobrowserkiosklockdown.services.f;
17.
18.             $r1 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
                android.content.Context h>;
19.
20.          label01:
21.             $r2 = virtualinvoke $r1.<android.content.Context: java.lang.Object
                getSystemService(java.lang.String)>("location");
22.
23.          label02:
24.             $r3 = (android.location.LocationManager) $r2;
25.
```

```
26.              $r0.<net.intricare.gobrowserkiosklockdown.services.f:
                android.location.LocationManager g> = $r3;
27.
28.              $r3 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
                android.location.LocationManager g>;
29.
30.             label03:
31.              $z0 = virtualinvoke $r3.<android.location.LocationManager: boolean
                isProviderEnabled(java.lang.String)>("gps");
32.
33.             label04:
34.              $r0.<net.intricare.gobrowserkiosklockdown.services.f: boolean a> = $z0;
35.
36.              $r3 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
                android.location.LocationManager g>;
37.
38.             label05:
39.              $z0 = virtualinvoke $r3.<android.location.LocationManager: boolean
                isProviderEnabled(java.lang.String)>("network");
40.
41.             label06:
42.              $r0.<net.intricare.gobrowserkiosklockdown.services.f: boolean b> = $z0;
43.
44.              $z0 = $r0.<net.intricare.gobrowserkiosklockdown.services.f: boolean a>;
45.
46.              if $z0 != 0 goto label08;
47.
48.              $z0 = $r0.<net.intricare.gobrowserkiosklockdown.services.f: boolean b>;
49.
50.              if $z0 != 0 goto label08;
```

```
51.
52.             label07:
53.                $r5 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
               android.location.Location d>;
54.
55.                return $r5;
56.
57.             label08:
58.                $r0.<net.intricare.gobrowserkiosklockdown.services.f: boolean c> = 1;
59.
60.                $z0 = $r0.<net.intricare.gobrowserkiosklockdown.services.f: boolean b>;
61.
62.                if $z0 == 0 goto label24;
63.
64.                $r3 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
               android.location.LocationManager g>;
65.
66.             label09:
67.                $r4 = staticinvoke <android.os.Looper: android.os.Looper getMainLooper()>();
68.
69.                virtualinvoke $r3.<android.location.LocationManager: void
               requestLocationUpdates(java.lang.String,long,float,android.location.LocationLis
               tener,android.os.Looper)>("network", 60000L, 10.0F, $r0, $r4);
70.
71.                staticinvoke <android.util.Log: int
               d(java.lang.String,java.lang.String)>("Network", "Network");
72.
73.             label10:
74.                $r3 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
               android.location.LocationManager g>;
```

```
75.
76.             if $r3 == null goto label24;
77.
78.             $r3 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
               android.location.LocationManager g>;
79.
80.           label11:
81.             $r5 = virtualinvoke $r3.<android.location.LocationManager:
               android.location.Location getLastKnownLocation(java.lang.String)>("network");
82.
83.           label12:
84.             $r0.<net.intricare.gobrowserkiosklockdown.services.f:
               android.location.Location d> = $r5;
85.
86.             $r5 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
               android.location.Location d>;
87.
88.             if $r5 == null goto label24;
89.
90.             $r5 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
               android.location.Location d>;
91.
92.           label13:
93.             $d0 = virtualinvoke $r5.<android.location.Location: double getLatitude()>();
94.
95.           label14:
96.             $r0.<net.intricare.gobrowserkiosklockdown.services.f: double e> = $d0;
97.
98.             $r5 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
               android.location.Location d>;
```

```
99.
100.          label15:
101.             $d0 = virtualinvoke $r5.<android.location.Location: double getLongitude()>();
102.
103.          label16:
104.             $r0.<net.intricare.gobrowserkiosklockdown.services.f: double f> = $d0;
105.
106.             $r6 = new java.lang.StringBuilder;
107.
108.          label17:
109.             specialinvoke $r6.<java.lang.StringBuilder: void <init>()>();
110.
111.             $r6 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.StringBuilder
             append(java.lang.String)>("latitude [");
112.
113.          label18:
114.             $d0 = $r0.<net.intricare.gobrowserkiosklockdown.services.f: double e>;
115.
116.          label19:
117.             $r6 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.StringBuilder
             append(double)>($d0);
118.
119.             $r6 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.StringBuilder
             append(java.lang.String)>(" ]");
120.
121.             $r7 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.String
             toString()>();
122.
123.             staticinvoke <android.util.Log: int
             d(java.lang.String,java.lang.String)>("Lac", $r7);
```

```
124.
125.          label20:
126.             $r6 = new java.lang.StringBuilder;
127.
128.          label21:
129.             specialinvoke $r6.<java.lang.StringBuilder: void <init>()>();
130.
131.             $r6 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.StringBuilder
             append(java.lang.String)>("longitude [");
132.
133.          label22:
134.             $d0 = $r0.<net.intricare.gobrowserkiosklockdown.services.f: double f>;
135.
136.          label23:
137.             $r6 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.StringBuilder
             append(double)>($d0);
138.
139.             $r6 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.StringBuilder
             append(java.lang.String)>(" ]");
140.
141.             $r7 = virtualinvoke $r6.<java.lang.StringBuilder: java.lang.String
             toString()>();
142.
143.             staticinvoke <android.util.Log: int
             d(java.lang.String,java.lang.String)>("Lac", $r7);
144.
145.          label24:
146.             $z0 = $r0.<net.intricare.gobrowserkiosklockdown.services.f: boolean a>;
147.
148.             if $z0 == 0 goto label07;
```

```
149.
150.             $r5 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
                 android.location.Location d>;
151.
152.             if $r5 != null goto label07;
153.
154.             $r3 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
                 android.location.LocationManager g>;
155.
156.           label25:
157.             $r4 = staticinvoke <android.os.Looper: android.os.Looper getMainLooper()>();
158.
159.             virtualinvoke $r3.<android.location.LocationManager: void
                 requestLocationUpdates(java.lang.String,long,float,android.location.LocationLis
                 tener,android.os.Looper)>("gps", 60000L, 10.0F, $r0, $r4);
160.
161.             staticinvoke <android.util.Log: int
                 d(java.lang.String,java.lang.String)>("GPS Enabled", "GPS Enabled");
162.
163.           label26:
164.             $r3 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
                 android.location.LocationManager g>;
165.
166.             if $r3 == null goto label07;
167.
168.             $r3 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
                 android.location.LocationManager g>;
169.
170.           label27:
```

```
171.             $r5 = virtualinvoke $r3.<android.location.LocationManager:
             android.location.Location getLastKnownLocation(java.lang.String)>("gps");
172.
173.          label28:
174.             $r0.<net.intricare.gobrowserkiosklockdown.services.f:
             android.location.Location d> = $r5;
175.
176.             $r5 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
             android.location.Location d>;
177.
178.             if $r5 == null goto label07;
179.
180.             $r5 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
             android.location.Location d>;
181.
182.          label29:
183.             $d0 = virtualinvoke $r5.<android.location.Location: double getLatitude()>();
184.
185.          label30:
186.             $r0.<net.intricare.gobrowserkiosklockdown.services.f: double e> = $d0;
187.
188.             $r5 = $r0.<net.intricare.gobrowserkiosklockdown.services.f:
             android.location.Location d>;
189.
190.          label31:
191.             $d0 = virtualinvoke $r5.<android.location.Location: double getLongitude()>();
192.
193.          label32:
194.             $r0.<net.intricare.gobrowserkiosklockdown.services.f: double f> = $d0;
195.
```

```
196.             goto label07;
197.
198.          label33:
199.             $r8 := @caughtexception;
200.
201.          label34:
202.             virtualinvoke $r8.<java.lang.SecurityException: void printStackTrace()>();
203.
204.          label35:
205.             goto label07;
206.
207.          label36:
208.             $r9 := @caughtexception;
209.
210.             virtualinvoke $r9.<java.lang.Exception: void printStackTrace()>();
211.
212.             goto label07;
213.
214.          label37:
215.             $r10 := @caughtexception;
216.
217.          label38:
218.             virtualinvoke $r10.<java.lang.SecurityException: void printStackTrace()>();
219.
220.          label39:
221.             goto label24;
222.
223.             catch java.lang.Exception from label01 to label02 with label36;
224.             catch java.lang.Exception from label03 to label04 with label36;
225.             catch java.lang.Exception from label05 to label06 with label36;
```

```
226.             catch java.lang.SecurityException from label09 to label10 with label37;
227.             catch java.lang.SecurityException from label11 to label12 with label37;
228.             catch java.lang.SecurityException from label13 to label14 with label37;
229.             catch java.lang.SecurityException from label15 to label16 with label37;
230.             catch java.lang.SecurityException from label17 to label18 with label37;
231.             catch java.lang.SecurityException from label19 to label20 with label37;
232.             catch java.lang.SecurityException from label21 to label22 with label37;
233.             catch java.lang.SecurityException from label23 to label24 with label37;
234.             catch java.lang.Exception from label09 to label10 with label36;
235.             catch java.lang.Exception from label11 to label12 with label36;
236.             catch java.lang.Exception from label13 to label14 with label36;
237.             catch java.lang.Exception from label15 to label16 with label36;
238.             catch java.lang.Exception from label17 to label18 with label36;
239.             catch java.lang.Exception from label19 to label20 with label36;
240.             catch java.lang.Exception from label21 to label22 with label36;
241.             catch java.lang.Exception from label23 to label24 with label36;
242.             catch java.lang.SecurityException from label25 to label26 with label33;
243.             catch java.lang.SecurityException from label27 to label28 with label33;
244.             catch java.lang.SecurityException from label29 to label30 with label33;
245.             catch java.lang.SecurityException from label31 to label32 with label33;
246.             catch java.lang.Exception from label25 to label26 with label36;
247.             catch java.lang.Exception from label27 to label28 with label36;
248.             catch java.lang.Exception from label29 to label30 with label36;
249.             catch java.lang.Exception from label31 to label32 with label36;
250.             catch java.lang.Exception from label34 to label35 with label36;
251.             catch java.lang.Exception from label38 to label39 with label36;
252.         }
```

Listing B4. Database

```
1.      Class: net.intricare.gobrowserkiosklockdown.firebasenotify.MyFirebaseMessagingService
2.          private boolean a(java.lang.String)
3.          {
4.
             net.intricare.gobrowserkiosklockdown.firebasenotify.MyFirebaseMessagingService
             $r0;
5.              java.lang.String $r1, $r3;
6.              java.lang.StringBuilder $r2;
7.              net.intricare.gobrowserkiosklockdown.b.b $r4;
8.              int $i0;
9.
10.             $r0                                  :=                                  @this:
             net.intricare.gobrowserkiosklockdown.firebasenotify.MyFirebaseMessagingService;
11.
12.             $r1 := @parameter0: java.lang.String;
13.
14.             $r2 = new java.lang.StringBuilder;
15.
16.             specialinvoke $r2.<java.lang.StringBuilder: void <init>()>();
17.
18.             $r2  =  virtualinvoke  $r2.<java.lang.StringBuilder:  java.lang.StringBuilder
             append(java.lang.String)>("oldpassword [");
19.
20.             $r3                                =                                specialinvoke
             $r0.<net.intricare.gobrowserkiosklockdown.firebasenotify.MyFirebaseMessagingSer
             vice: java.lang.String a()>();
21.
```

```
22.             $r2 = virtualinvoke $r2.<java.lang.StringBuilder: java.lang.StringBuilder
            append(java.lang.String)>($r3);
23.
24.             $r2 = virtualinvoke $r2.<java.lang.StringBuilder: java.lang.StringBuilder
            append(java.lang.String)>("]");
25.
26.             $r3 = virtualinvoke $r2.<java.lang.StringBuilder: java.lang.String
            toString()>();
27.
28.             staticinvoke <android.util.Log: int
            i(java.lang.String,java.lang.String)>("MyFirebaseMsgService", $r3);
29.
30.             staticinvoke <android.util.Log: int
            i(java.lang.String,java.lang.String)>("MyFirebaseMsgService", "oldpassword and
            newpassword matches. save new password");
31.
32.             $r4 =
            <net.intricare.gobrowserkiosklockdown.firebasenotify.MyFirebaseMessagingService
            : net.intricare.gobrowserkiosklockdown.b.b a>;
33.
34.             $i0 = virtualinvoke $r4.<net.intricare.gobrowserkiosklockdown.b.b: int
            c(java.lang.String)>($r1);
35.
36.             $r4 =
            <net.intricare.gobrowserkiosklockdown.firebasenotify.MyFirebaseMessagingService
            : net.intricare.gobrowserkiosklockdown.b.b a>;
37.
38.             virtualinvoke $r4.<net.intricare.gobrowserkiosklockdown.b.b: void b()>();
39.
40.             if $i0 <= 0 goto label1;
```

```
41.
42.             return 1;
43.
44.          label1:
45.             return 0;
46.          }
47.      ----------------------------------------------------------------------------------
48.          Class:
              net.intricare.gobrowserkiosklockdown.firebasenotify.MyFirebaseMessagingService
49.          private java.lang.String a()
50.          {
51.
              net.intricare.gobrowserkiosklockdown.firebasenotify.MyFirebaseMessagingService
              $r0;
52.             java.lang.String $r1;
53.             net.intricare.gobrowserkiosklockdown.b.b $r2;
54.
55.             $r0                                  :=                                  @this:
              net.intricare.gobrowserkiosklockdown.firebasenotify.MyFirebaseMessagingService;
56.
57.             $r2                                                                          =
              <net.intricare.gobrowserkiosklockdown.firebasenotify.MyFirebaseMessagingService
              : net.intricare.gobrowserkiosklockdown.b.b a>;
58.
59.             $r1     =     virtualinvoke     $r2.<net.intricare.gobrowserkiosklockdown.b.b:
              java.lang.String e()>();
60.
61.             return $r1;
62.          }
63.      ----------------------------------------------------------------------------------
```

```
64.          Class: net.intricare.gobrowserkiosklockdown.b.b
65.          public java.lang.String e()
66.          {
67.              net.intricare.gobrowserkiosklockdown.b.b $r0;
68.              android.database.sqlite.SQLiteDatabase $r1;
69.              java.lang.String[] $r2;
70.              android.database.Cursor $r3;
71.              boolean $z0;
72.              int $i0;
73.              java.lang.String $r4;
74.
75.              $r0 := @this: net.intricare.gobrowserkiosklockdown.b.b;
76.
77.              $r1              =              $r0.<net.intricare.gobrowserkiosklockdown.b.b:
              android.database.sqlite.SQLiteDatabase b>;
78.
79.              $r2 = newarray (java.lang.String)[1];
80.
81.              $r2[0] = "password";
82.
83.              $r3     =     virtualinvoke     $r1.<android.database.sqlite.SQLiteDatabase:
              android.database.Cursor
              query(java.lang.String,java.lang.String[],java.lang.String,java.lang.String[],j
              ava.lang.String,java.lang.String,java.lang.String)>("login",  $r2,  null,  null,
              null, null, null);
84.
85.              $z0 = interfaceinvoke $r3.<android.database.Cursor: boolean moveToFirst()>();
86.
87.              if $z0 == 0 goto label1;
88.
```

```
89.              $i0      =      interfaceinvoke      $r3.<android.database.Cursor:      int
              getColumnIndex(java.lang.String)>("password");
90.
91.              $r4   =   interfaceinvoke   $r3.<android.database.Cursor:   java.lang.String
              getString(int)>($i0);
92.
93.              return $r4;
94.
95.            label1:
96.              return null;
97.           }
```

Bibliography

Android 17 "Android, Log," Android Official Developers Online Document. https://developer.android.com/reference/android/util/Log (accessed 2017).

Arzt 14 S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, and A. Bartel, "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, Edinburgh, United Kingdom, 2013, pp. 259–269.

Asf 16 "Welcome to The Apache Software Foundation!" https://www.apache.org/ (accessed Sep. 01, 2020).

Bartel 12 A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon, "Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot," Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis - SOAP '12, pp. 27–38

Cai 17 H. Cai and B. G. Ryder, "DroidFax: A Toolkit for Systematic Characterization of Android Applications," in 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), Sep. 2017, pp. 643–647

Chen 17a B. Chen and Z. M. (Jack) Jiang, "Characterizing logging practices in Java-based open source software projects – a replication study in Apache Software Foundation," Empir Software Eng, vol. 22, no. 1, Feb. 2017, pp. 330–374

Chen 17b B. Chen and Z. M. (Jack) Jiang, “Characterizing and Detecting Anti-patterns in the Logging Code,” in Proceedings of the 39th International Conference on Software Engineering, Piscataway, NJ, USA, 2017, pp. 71–81.

Cinque 10 M. Cinque, D. Cotroneo, R. Natella, and A. Pecchia, “Assessing and improving the effectiveness of logs for the analysis of software faults,” in 2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN), Jun. 2010, pp. 457–466.

Cinque 12 M. Cinque, D. Cotroneo, and A. Pecchia, “Event Logs for the Analysis of Software Failures: A Rule-Based Approach,” IEEE Transactions on Software Engineering, vol. 39, no. 6, Jun. 2013, pp. 806–821.

Ding 15 R. Ding, H. Zhou, J. Lou, H. Zhang, and Q. Lin, “Log2: a cost-aware logging mechanism for performance diagnosis,” in Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference, USA, Jul. 2015, pp. 139–150.

Durase 06 J. A. Duraes and H. S. Madeira, “Emulation of Software Faults: A Field Data Study and a Practical Approach,” IEEE Transactions on Software Engineering, vol. 32, no. 11, pp. 849–867, Nov. 2006.

El-Masri 20 D. El-Masri, F. Petrillo, Y.-G. Guéhéneuc, A. Hamou-Lhadj, and A. Bouziane, “A systematic literature review on automated log abstraction techniques,” Information and Software Technology, vol. 122, p. 106276, Jun. 2020.

F-Droid 17 “F-Droid - Free and Open Source Android App Repository.” https://f-droid.org/en/ (accessed 2017).

Feng 17 P. Feng, J. Ma, and C. Sun, “Selecting Critical Data Flows in Android Applications for Abnormal Behavior Detection,” Mobile Information Systems, Apr. 30, 2017.

Fu 14 Q. Fu et al., “Where do developers log? an empirical study on logging practices in industry,” in Companion Proceedings of the 36th International Conference on Software Engineering, New York, NY, USA, May 2014, pp. 24–33.

Hamou-Lhadj 02 A. Hamou-Lhadj and T. C. Lethbridge, “Compression techniques to simplify the analysis of large execution traces,” in Proc. of the10th International Workshop on Program Comprehension, 2002, pp. 159–168.

Hamou-Lhadj 06 A. Hamou-Lhadj and T. Lethbridge, “Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system,” in Proc. of the 14th IEEE International Conference on Program Comprehension (ICPC’06), 2006, pp. 181–190.

Islam 18 Md S. Islam, W. Khreich, and A. Hamou-Lhadj, “Anomaly detection techniques based on kappa-pruned ensembles,” IEEE Transactions on Reliability, 67(1), 2018, pp. 212–229.

Hassan 08 A. E. Hassan, D. J. Martin, P. Flora, P. Mansfield, and D. Dietz, “An Industrial Case Study of Customizing Operational Profiles Using Log Compression,” in Proceedings of the 30th international conference on Software engineering, New York, NY, USA, May 2008, pp. 713–723.

He 18 P. He, Z. Chen, S. He, and M. R. Lyu, “Characterizing the natural language descriptions in software logging statements,” in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, New York, NY, USA, Sep. 2018, pp. 178–189.

Huang 14a W. Huang, Y. Dong, and A. Milanova, “Type-Based Taint Analysis for Java Web Applications,” in Proceedings of the 17th International Conference on Fundamental

Approaches to Software Engineering - Volume 8411, Berlin, Heidelberg, Apr. 2014, pp. 140–154.

Huang 14b J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: detecting stealthy behaviors in Android applications by user interface and program behavior contradiction," in Proceedings of the 36th International Conference on Software Engineering, New York, NY, USA, May 2014, pp. 1036–1046.

Huang 16 J. Huang, X. Zhang, and L. Tan, "Detecting sensitive data disclosure via bi-directional text correlation analysis," in Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, New York, NY, USA, Nov. 2016, pp. 169–180.

Jdt 15 J. team, "Eclipse Java development tools (JDT) | The Eclipse Foundation." https://www.eclipse.org/jdt/ (accessed 23 October 2015).

Kabinna 16 S. Kabinna, C.-P. Bezemer, W. Shang, and A. E. Hassan, "Logging library migrations: a case study for the apache software foundation projects," in Proceedings of the 13th International Conference on Mining Software Repositories, New York, NY, USA, May 2016, pp. 154–164.

Kernighan 99 B. W. Kernighan and R. Pike, The practice of programming. Reading, MA: Addison-Wesley, 1999.

Khanmohammadi 19 K. Khanmohammadi, N. Ebrahimi, A. Hamou-Lhadj, and R. Khoury, "Empirical study of android repackaged applications," Empir Software Eng, vol. 24, no. 6, pp. 3587–3629, Dec. 2019.

Khatuya 18 S. Khatuya, N. Ganguly, J. Basak, M. Bharde, and B. Mitra, “ADELE: Anomaly Detection from Event Log Empiricism,” in IEEE INFOCOM 2018 - IEEE Conference on Computer Communications, Apr. 2018, pp. 2114–2122.

Klieber 14 W. Klieber, L. Flynn, A. Bhosale, L. Jia, and L. Bauer, “Android taint flow analysis for app sets,” in Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis - SOAP ’14, Edinburgh, United Kingdom, 2014, pp. 1–6.

Le 15 T.-D. B. Le, X.-B. D. Le, D. Lo, and I. Beschastnikh, “Synergizing Specification Miners through Model Fissions and Fusions (T),” in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Nov. 2015, pp. 115–125.

Li 17a H. Li, W. Shang, and A. E. Hassan, “Which log level should developers choose for a new logging statement? (journal-first abstract),” in 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), Mar. 2018, pp. 468–468.

Li 17b L. Li et al., “AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community,” arXiv:1709.05281 [cs], Sep. 2017.

Li 19 Z. Li, T.-H. Chen, J. Yang, and W. Shang, “DLFinder: Characterizing and Detecting Duplicate Logging Code Smells,” in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, May 2019, pp. 152–163.

Lu 12 L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, “CHEX: statically vetting Android apps for component hijacking vulnerabilities,” in Proceedings of the 2012 ACM

conference on Computer and communications security, New York, NY, USA, Oct. 2012, pp. 229–240.

Mariani 08 L. Mariani and F. Pastore, “Automated Identification of Failure Causes in System Logs,” in 2008 19th International Symposium on Software Reliability Engineering (ISSRE), Nov. 2008, pp. 117–126.

Miransky 16 A. Miranskyy, A. Hamou-Lhadj, E. Cialini, and A. Larsson, “Operational-Log Analysis for Big Data Systems: Challenges and Solutions,” IEEE Software, vol. 33, no. 2, pp. 52–59, Mar. 2016.

OWASP 16 “Mobile Top 10 2016-Top 10 - OWASP.” https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10 (accessed 2016).

Pecchia 15 A. Pecchia, M. Cinque, G. Carrozza, and D. Cotroneo, “Industry Practices and Event Logging: Assessment of a Critical Software Development Process,” in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, May 2015, vol. 2, pp. 169–178.

Rahul 13 R. Pandita, X. Xiao, W. Yang, W. Enck, and T. Xie, “WHYPER: Towards Automating Risk Assessment of Mobile Applications,” p. 16.

Rasthofer 14 S. Rasthofer, S. Arzt, and E. Bodden, “A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks,” presented at the Network and Distributed System Security Symposium, San Diego, CA, 2014.

Shang 14 W. Shang, M. Nagappan, A. E. Hassan, and Z. M. Jiang, “Understanding Log Lines Using Development Knowledge,” in 2014 IEEE International Conference on Software Maintenance and Evolution, Sep. 2014, pp. 21–30.

Xdadevelopers 19 “How to take logs in Android,” xda-developers. https://www.xda-developers.com/how-to-take-logs-in-android/ (accessed 2019).

Xu 09 W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan, “Detecting large-scale system problems by mining console logs,” in Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, New York, NY, USA, Oct. 2009, pp. 117–132.

Yen 13 T.-F. Yen et al., “Beehive: large-scale log analysis for detecting suspicious activity in enterprise networks,” in Proceedings of the 29th Annual Computer Security Applications Conference on - ACSAC ’13, New Orleans, Louisiana, 2013, pp. 199–208.

Yuan 12a D. Yuan, S. Park, and Y. Zhou, “Characterizing logging practices in open-source software,” in 2012 34th International Conference on Software Engineering (ICSE), Zurich, Jun. 2012, pp. 102–112.

Yuan 12b D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, “Improving software diagnosability via log enhancement,” in Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems, New York, NY, USA, Mar. 2011, pp. 3–14.

Zeng 19 Y. Zeng, J. Chen, W. Shang, and T.-H. (Peter) Chen, “Studying the characteristics of logging practices in mobile apps: a case study on F-Droid,” Empir Software Eng, Feb. 2019.

Zhu 15 J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, “Learning to Log: Helping Developers Make Informed Logging Decisions,” in 2015 IEEE/ACM 37th

IEEE International Conference on Software Engineering, Florence, Italy, May 2015, pp. 415–425.

Zhou 20 R. Zhou, M. Hamdaqa, H. Cai, and A. Hamou-Lhadj, “MobiLogLeak: A Preliminary Study on Data Leakage Caused by Poor Logging Practices,” in Proc. of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER'20), 2020, pp. 577–581.