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

Abstract—**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 the users' accounts. This preliminary 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, an approach 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.**

Keywords—Taint Flow Analysis, Mobile Applications, Data Leakage, Logging Practices

I. Introduction

As in other software applications, mobile developers use logging to gain insight into the behavior of the application at runtime [1]. Log messages printed during the execution of a program are often the only data source available for developers to diagnose program failures. Logs are also used by DevOps analysts to determine the root cause of problems that occur at the operational level such as finding the causes behind security attacks, hardware failures, and configuration issues. However, as stated in the android developer guide "verbose logging should never be compiled into an application except during development" [2]. Verbose logging may make debugging inefficient and unproductive. Despite all the warnings, poor logging is still a common practice in mobile development [3].

There exist studies that examine the practice of logging in mobile 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. [4] [5] [6]. In this paper, we argue that poor logging practices may lead to a more serious problem, which is exposing user private personal information and leak sensitive data that can be a source of privacy and security vulnerabilities. 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 (more precisely data privacy) in the context of mobile applications for Android.

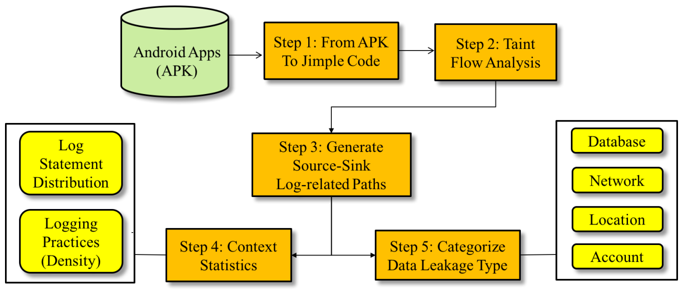

Android Apps (APK)
Step 1: From APK To Jimple Code
Step 2: Taint Flow Analysis
Step 3: Generate Source-Sink Log-related Paths
Log Statement Distribution
Logging Practices (Density)
Step 4: Context Statistics
Step 5: Categorize Data Leakage Type
Database
Network
Location
Account

Fig. 1. MobiLogLeak Approach Overview

II. MobiLogLeak Approach

This paper focuses on studying potential data leakage due to poor logging practices in released Android mobile applications. Fig. 1 shows an overview of our approach. In **Step 1**, we need to convert the Android APK to Jimple Code in order to be able to analyse it. Then, in **Step 2**, we apply taint analysis to the Jimple Code to find the taint flow paths in each application. In our work, we focus only on apps with taint flows as those are the potential sources for data leakage. Moreover, since this study focuses on data leakage as a result of poor logging practices, in **Step 3**, 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. The results will be log specific taint flow paths. In **Step 4**, to obtain a better understanding of these generated log related flows, we perform context analysis to analyze the code structure in order to study, which Android component has the most log related flows. Finally, in **Step 5**, we manually inspect each of the taint flow paths to understand the type of the possible data leakage cases. We elaborate on each step in the subsequent subsections.

A. Step 1: From 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 revers engineer the app APK to an intermediate representation that we can analyze. To this end, we turn to the Soot framework [7], which is a framework for analyzing and transforming Java and Android applications to Jimple, a code format between source code and Bytecode that is commonly used in program analysis [7].

Although Jimple is more cumbersome than normal source code, it 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 log statements. The following listing shows an example of a Jimple code snippet of a log statement that is generated from an APK using Soot.

```
staticinvoke <android.util.Log: int d(java.lang.
 String,java.lang.String)>("deviceIMEI", $r6);
```

B. Step 2: Taint Flow Analysis

Our main focus in this work is to identify sources of data leaks that are related to log statements. One way to identify data leaks is through taint flow analysis. The goal of taint flow analysis is to check whether sensitive data remains within expected application's boundaries [8]. Taint analysis can be either static or dynamic. In this paper, we apply static taint flow analysis on the generated Jimple Code from Step 1. This is performed using Flowdroid [9], 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.

C. Step 3: Generate the Source-Sink Log-Related Paths

The paths that are generated in Step 2 include all sources of data leaks. In this step, we refine the list of path, 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 Jimple, 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 paper, we refer to these paths as ALRS (App Logging Related Sinks).

D. Step 4: Context Analysis of Logging Related Data Leakage

In this step, we measure different aspects related to 'bad' log statements (logs that appear on the taint flow analysis paths).

First we compare the number of log statements in applications with poor logging (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.

For this, we use DroidFax [10], 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.

To measure the density of poor logging, we used the following equation:

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

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

E. Step 5: Types of Logging Related Data Leakage

The aim of this step is to categorize logging related data leakage cases. To this ends, 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.

III. Preliminary Evaluation

A. Dataset

We applied MobiLogLeaks to a randomly collected sample of 5,000 Android applications from AndroZoo [11] (2,500 apps were published 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. 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 with taint flow paths that have log-related sinks. The final refined dataset includes apps that cover various app categories including music, finance, rducation, fitness, etc. Some of these applications are highly popular, more particularly, 29 of these apps were installed more than 100,000 times.

B. Results

1) Analyzing the Types of Logging: Table I shows the results of comparing the logging *density* between applications with logging related taint flows (ALRS) and those without (referred to as 'Good' apps).

TABLE I. Log Practices in Android

Dataset	SLOC	LLOC	Density
Good 4,800 APKs	436,167,099	1,205,557	1/362
ALRS 200 APKs	34,686,096	64,296	1/539

The table shows that 'good' apps have higher log density. At the first glace, this looks as a surprising result. To further explore this, we separated the data into two datasets; each consists of 2,500 apps from the same year of publication and recalculate the density. The results were consistent with the previous findings. One possible explanation for this is that log heavy applications are those that are normally written by experienced developers, who also apply good logging practices, such as removing log statements that could review sensitive data, but also use more logs for exception handling and explaining errors. Further analysis is required to draw a firm conclusion.

Table II shows the results of comparing the context in which the log statements appear (i.e., the distribution of log statements in different Android components). The results show that most logging occur in *Activity* components, with almost

TABLE II. Log Distribution in Android Component

Component	Good 4,800 APKs	ALRS 200 APKs
Activity	96.25%	98.5%
Service	0.29%	0%
Broadcast Receiver	0.13%	0.5%
Content Provider	0%	0%
Other	3.33%	1%

no logging in content provider (zero in our case). The most remarkable result is that the *Service* components can be used as a determinant feature to indicate good logging practices. Again this was consistent when we applied the analysis on the 5,000 applications and when we separate them into two datasets (each consisting of 2,500 apps) in a random manner. Similar to the previous result, this finding requires further investigation. However, it also draw attention to the differences in the behaviour in poor and good logging practices. It is worth mentioning that in the case of poor logging (ALRS), the dataset contains 200 apps with 293 log-related sinks. All of them are in *Activity* components.

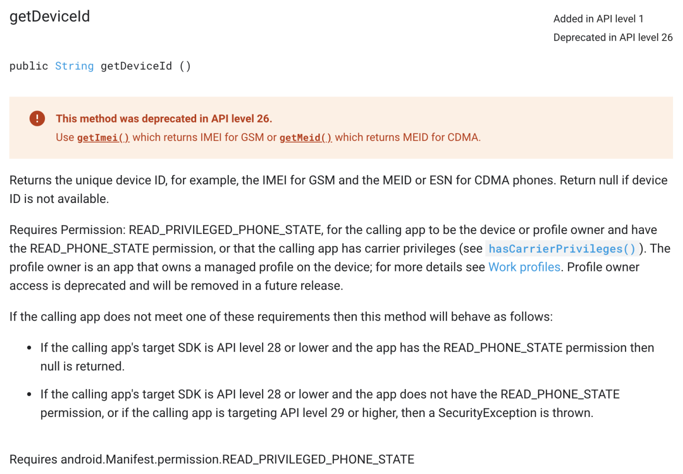
As part of this analysis, we studied if sinks can appear in in *reflective* methods and *exception* blocks as those two constructs normally contain log statements (i.e., we reported a consistent percentage of 25% of the total log statements are in the exception blocks and reflection methods). No poor logging (i.e., sink related log statements) was reported in those two constructs.

2) Analyzing the Types of Logging: In this section, we show by example, how we manually inspect each of the taint flow paths generated in *Step 3*, in order to categorise 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.

For example, in the listing below, the method getDeviceId() in line 4 is a source that is related to the log-relate sink that leaks the device IMEI in line 6. 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).

```
1  public static java.lang.String i(android.content.
    Context)
2     { ...
3      label05:
4        $r3 = virtualinvoke $r2.<android.telephony.
          TelephonyManager: java.lang.String
          getDeviceId()>();
5        ...
```



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

Fig. 2. API: GetDeviceId

```
6        staticinvoke <android.util.Log: int d(java.
          lang.String,java.lang.String)>("deviceIMEI
          ", $r6);
7      ...
8     }
```

Listing 1. Code Snippet: Taint Flow Example of Device ID

Fig. 2 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. For the previous example, the data leakage is related to Network.

- Network: Mac Address, Device Id Sim Serial Number, Country and Package Manager.
- Account: Name and Size.
- Location: Latitude, Longitude, and Last Know Location.
- Database: ID, Password, Subdomain, Website Link Name, etc.

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 more 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. Hence, the analysis normally includes tracking the *real source*. For example, in the following listing, starting from the source in line 32, Flowdroid pointed to a source for a database leakage. In order to identify the actual data leaked, 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.

```
1  private boolean a(java.lang.String)
2     {
3        ...
4
5        $r3 = specialinvoke $r0.<net.intricare.
          gobrowserkiosklockdown.firebasenotify.
          MyFirebaseMessagingService: java.lang.
          String a()>();
6
7        ...
8
9        staticinvoke <android.util.Log: int i(java.
          lang.String,java.lang.String)>("
          MyFirebaseMsgService", $r3);
10
11       staticinvoke <android.util.Log: int i(java.
          lang.String,java.lang.String)>("
          MyFirebaseMsgService", "oldpassword and
          newpassword matches. save new password");
12
13       ...
14    }
15 ------------------------------------------------
16 private java.lang.String a()
17    {
18       ...
19
20       $r1 = virtualinvoke $r2.<net.intricare.
          gobrowserkiosklockdown.b.b: java.lang.
          String e()>();
21
22       return $r1;
23    }
24
25 ------------------------------------------------
26 public java.lang.String e()
27    {
28    ...
29
30       $i0 = interfaceinvoke $r3.<android.database
          .Cursor: int getColumnIndex(java.lang.
          String)>("password");
31
32       $r4 = interfaceinvoke $r3.<android.database
          .Cursor: java.lang.String getString(int)>(
          $i0);
33
34       return $r4;
35    ...
36    }
```

Listing 2. Code Snippet: Sink Example of Database(Password)

For example, in Listing 2, to track the data and confirm the real path, we start from the sink in line 9 in Listing 2. It is the log statement that contains the variable *$r3*. We go back and look for the value of *$r3* and in line 5, 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 *$r1* in line 22. It comes from the function 'public java.lang.String e()' in line 20. Then we trace this function shown at the line 26. In the function, public java.lang.String e(), we could find the final return value is *$r4*. Based on FlowDroid, the source is in line 32.

However, if we stop here, we only could know the data is from the database. What is the real source? Which part value is leaked in the database? To solve these kinds of confusion, 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 30.

In this case, the source is transformed in 2 other functions, and we can check the static text of logging in line 11. It is "oldpassword and newpassword matches. save new password". It means this logging happens during change the password. As we mentioned before, most taint flows are most 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 Logging Related Sinks (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 app. We need to dig further to understand (1) the scale of this problem, and (2) the causes.

IV. Related Work

Chen et al. [5] 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 afterthought 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, Y et al. [12] did a research of the characteristics of logging practices in 1,444 open source mobile apps from the F-droid repository [6]. They checked the logging density, rational behind logging, and performance impact. The author compared the results to two previous studies [4] [5] 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.

Chen et al. [3] 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 duplications. 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.

Khanmohammadi et al. [13] conducted research on Android repackaged apps, which are considered as one the top

10 risks in mobile security [14]. They tested more than 15,000 apps from AndroZoo [11] 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.

V. Threats to Validity

Internal Validity: We manually analyze 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 remain preliminary and should be qualified as an Early Research, and may not be generalizable to other apps .

VI. Conclusions

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, non 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, 200 or 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 380 sources 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. Finally, our preliminary results suggest a correlation between the context of logging practices and whether these practices are good or bad, further investigation is required.

References

[1] J. Zhu, P. He, Q. Fu, H. Zhang, M. R. Lyu, and D. Zhang, "Learning to log: Helping developers make informed logging decisions," in *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15. IEEE Press, pp. 415–425, event-place: Florence, Italy. [Online]. Available: http://dl.acm.org/citation.cfm?id=2818754.2818807

[2] Android api, log class, android.util.log. [Online]. Available: https://developer.android.com/reference/android/util/Log

[3] B. Chen and Z. M. J. Jiang, "Characterizing and detecting anti-patterns in the logging code," in *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17. IEEE Press, pp. 71–81, event-place: Buenos Aires, Argentina. [Online]. Available: https://doi.org/10.1109/ICSE.2017.15

[4] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, pp. 102–112. [Online]. Available: http://ieeexplore.ieee.org/document/6227202/

[5] B. Chen and Z. M. (Jack) Jiang, "Characterizing logging practices in java-based open source software projects – a replication study in apache software foundation," vol. 22, no. 1, pp. 330–374. [Online]. Available: http://link.springer.com/10.1007/s10664-016-9429-5

[6] F-droid - free and open source android app repository. [Online]. Available: https://f-droid.org/en/

[7] A. Bartel, J. Klein, M. Monperrus, and Y. L. Traon, "Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot," pp. 27–38. [Online]. Available: http://arxiv.org/abs/1205.3576

[8] 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*. ACM Press, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2614628.2614633

[9] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14*. ACM Press, pp. 259–269. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2594291.2594299

[10] 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)*, pp. 643–647, ISSN: null.

[11] AndroZoo++: Collecting millions of android apps and their metadata for the research community. [Online]. Available: https://arxiv.org/abs/1709.05281

[12] Y. Zeng, J. Chen, W. Shang, and T.-H. P. Chen, "Studying the characteristics of logging practices in mobile apps: a case study on f-droid." [Online]. Available: https://doi.org/10.1007/s10664-019-09687-9

[13] K. Khanmohammadi, N. Ebrahimi, A. Hamou-Lhadj, and R. Khoury, "Empirical study of android repackaged applications." [Online]. Available: https://doi.org/10.1007/s10664-019-09760-3

[14] Mobile top 10 2016-top 10 - OWASP. [Online]. Available: https://www.owasp.org/index.php/Mobile$_Top1 0_2 016 - Top1 0$