A Dataset of Protected Health Information (PHI) Leakage in Android Applications

Ashfak Md Shibli*
ams323@njit.edu
New Jersey Institute of Technology
Newark, New Jersey, USA

Tahiatul Islam†
ti54@njit.edu
New Jersey Institute of Technology
Newark, New Jersey, USA

Abstract

Mobile applications are widely used individually and organizations in the health industry. As a health conscious user every one of us have many health related applications in our phones. To provide services, these applications collect users' personal data which is protected health information i.e. PHI. It is obligatory for these applications to provide safeguards for such information in addition to platform level permissions. If there is any possibility of an information breach, the application should be able to detect those as early as possible. In this study, we examine the applications using static analysis method of taint analysis in identifying information leakage from applications and gather empirical data of possible application provider's goals and if this leakage is in wrong hands (Attacker) what can be major consequences. The sole propose of our study is to make a organized dataset (which are currently not widely available) for future research applications and provide possible resolutions to avoid PHI compromises from application developers and also regulatory bodies can use this data for safeguard users providing compliance information.

CCS Concepts: • **Software and its engineering** → *Software testing and debugging.*

Keywords: static analysis, software verification, intent, phi

ACM Reference Format:
Ashfak Md Shibli and Tahiatul Islam. 2022. A Dataset of Protected Health Information (PHI) Leakage in Android Applications. In *Proceedings of X*. ACM, Newark, NJ, USA, 7 pages.

1 Introduction

Protected Health Information (PHI) is a crucial aspect of health-related applications. Failure to protect PHI can ban the company from continuing future operations. So, it is critical for a health application to ensure that PHI does not leak from the software. To research and build a model for data compromise patterns in mobile health applications there is very few organized dataset available. Our goal is provide a well formulated database of analyzed application leakage with probable intents from different views, so that in future research we or the security research community can build a framework to check a app leaking data, possible intent and countermeasure instantly without running the application. Data leakage can occur for several reasons, like human error (naively declaring a constant with a passcode of the server) or insecure software interface/protocol [23]. Or it can happen potential codes that leaks data unknowingly by the developer. "Universal Remote Control" [21] having 10M downloads exposing location data to attacker or hacker who can exploit the data by different intentions [10]. Data loss prevention techniques [20] [9] that can detect data leakage would help developers avoid future risks. We will study the application of static analysis [5] to detect such leakage of information from sources like Android applications for mass usage. Our goal is to study with FlowDroid [6] to detect the possible leakage sources and check the patterns of leakage of data by app genres. By application genre, we mean a utility app like "Universal Remote Control" should not expose or use any data of user location. But in our analysis we found evidence of leaks by which users exposed with different kind of data. We also study empirically by reverse engineering the source code by ApkTool [4] to detect false positives, patterns of this application's data leakage, and the intent behind that from developer perspective and what can be attacker threat using the data.

2 Background

Mobile technological advancements have drawn millions of smartphone users. Currently there are 2.5 billion active android users worldwide [3] which is currently almost 72 percent market share. Along with smartphone, there are lots of sensors in them and personal-protected data-gathering tools in applications is being used by millions of users [11]. Among them facebook takes almost 70 of our day-to-day personal data. Also the users do not trust applications with their data with recent evident incidents of data compromises [19] [15]. If this data is leaked from the millions of users will be at risk of attacks. There are more third-party app markets and forums that provide free Android software in addition to the Google Play Store [the official android app store]. In the majority of these app marketplaces, the submitted App is not subject to rigorous security checks. Attackers can post their malicious programs to the market and infect a huge number of consumers since security checks are lacking.

An attacker can create malicious applications or change the already innocuous apps to capture sensitive information from the victim [22]. The sensitive information sent from the infected device by these Apps is then communicated with by the attacker, posing a serious information security and privacy risk to the device owner. Therefore, tools are a must that examines rogue apps to see if they provide any sensitive data to the attacker. The more information we can get from the malicious App, the more security measures we can offer to protect the Android user's important data. So in this study, we will use FlowDroid to detect whether sensitive data is leaked or not and get insights into which genre of application leaks which type of data or what is the intent of the creator and from creator how it can be exploited. This dataset can be used for future research by practitioners who will get prepared data out of the box. It will also give users and regulatory bodies idea of the applications which can be vulnerable for day to day usage.

2.1 Motivation

Health apps take lots of personal data for diagnosis purposes [16]. Health Insurance Portability and Accountability Act (HIPPA) lists 18 different information identifiers that, when paired with health information, become PHI [14]. Initially, information can be leaked due to just mistake of the developer that also somehow got out of code review phases. As a result that data can be leaked [1] and eventually end up in the production software which poses a risk to the software of being exploited by third parties. Logging tools can also leak [24] personal data due to the absence of data-hiding features in the production environment. Sometimes developers may also leak information unknowingly due to the use of an unsecured network interface. Thus, we study the potential apps that might have leakage in production and we are getting the data to realize the reason or intent behind those. By static analysis of the reverse-engineered source code, our study will show where static analysis and related tools reside in the industry. the potential information leakage source. We will try to give information so that the industry is more vigilant and careful while developing applications that put the personal data of millions put in risk.

2.2 Data and Security Risks

The user's position can be marked on a map using location coordinates collected from a GPS receiver. Smartphone users may be uniquely identified using DeviceID. Wireless network information on the network can reveal the access points' wireless security. Data kept in phones' internal or external storage, such as contacts, messages, phone logs, sim card information, and camera photographs, may potentially expose a great deal of information about the user in addition to data gathered via hardware sensors. The user is exposed to extra risks related to the criticality of the data if the user's sensitive information is disclosed, for example, a GPS coordinate can be exploited by an attacker to monitor the user's whereabouts. An attacker may exploit wifi-related data to ascertain the operating environment's wireless security for Android users. Malware can be used by an attacker to leak private data from Android smartphone users to a malicious website or logs. Also recent covid gave attackers many edges to exploit that 56 peercent [12] health data breach increased recently. If can get insights of how are these data actually being breached we can prevent this in future.

2.3 Static Analysis

Our primary goal is to check the data leakage of the applications and that is not possible by running the apps and checking different views. So understanding the instructions that are carried out by a program behind the views is necessary to assess it for malicious activity or data leakage. We can reverse engineer the Application to a suitable intermediate form so that we may follow these procedures. Assembly language or the source code itself might be used as the intermediate representation. Now that we have the application's source code or intermediate representation, we may begin analyzing it to find harmful instructions. We refer to this study as a static analysis as we do not need to execute the program. Static analysis has the key benefit of allowing us to directly apply intelligence to malicious code rather than merely recognizing harmful behavior. In our case we are not building tools from scratch as there are available tool (like FlowDroid) for our study. In next sections we discuss about the tool and inside algorithms.

2.4 FlowDroid

FlowDroid is a field, object and flowsensitive taint analysis which considers the Android application lifecycle and UI widgets, and which features a novel, particularly precise variant of an on-demand alias analysis. FlowDroid does taint analysis using context flow analysis and the Android lifecycle. There are several access points for Android apps. Four basic sorts of components are used by the Android application: activities for user actions, services for background chores, content providers for storage, and broadcast receivers for world events. All of these parts have registrations in the AndroidManifest.xml file. All of these elements are combined by the FlowDroid to build the application's lifecycle. The FlowDroid creates a fake Main method and call graph as input for the taint analysis engine using the layout. Figure 1 shows the workflow.

The three-address code intermediate format Jimple and the exact call-graph analysis framework Spark [13] are two key components of the Soot framework [8], which is extended by FlowDroid. FlowDroid is able to translate the Dalvik bytecode of Android into Jimple thanks to a plugin called Dexpler [7]. In addition to Soot and Dexpler, Heros [8], a scalable,

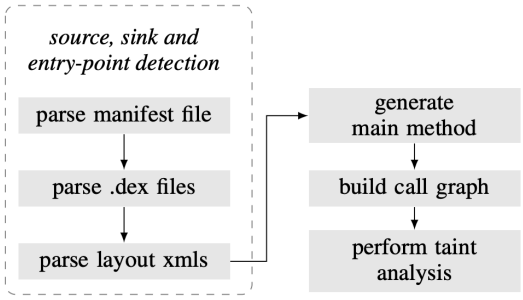

Figure 1. FlowDroid overview from paper

massively multi-threaded version of the IFDS framework [18], is also used by FlowDroid.

2.5 Taint Analysis

Taint analysis is a method of information flow analysis that examines data transported by various variables and program objects. The information linked to the variable is changed when an instruction is carried out. Taint analysis may be used to follow the information flow and identify sensitive data flow from source to sink. In the context of our study, we refer to sensitive information as a taint. To trace the spread of taint and identify the data leakage of the application, there are code analysis techniques.

Listing 1. Pseudocode for Taint Analysis

```
1  str a = getSensitiveData();
2  str b = a ;
3  str z = b + 'SOME_SUFFIX';
```

To retrieve sensitive data from the phone we give an example method which gets some data from native android API or a higher level method for that. Line 1 of the code "getSensitiveData()" resembles a similar method. Then string a that holds sensitive information is now marked as tainted since we refer to sensitive information as "taint". We can store a table in memory that contains variables and their taint state to continue tracking the propagation. String b is given the value of a in line 2. The operation assigns variable b with the tainted value of a. Then eventually it comes to line 3 where z is also tainted as b is added which was tainted on line 2. The aforementioned actions are a straightforward illustration of taint tracking. In this study, we use FlowDroid for taint analysis to manage difficult data assignment procedures and other required process to get the leakage warnings.

2.6 Dexpler

Although Android applications are written in Java, Dalvik bytecode is used to run them instead of Java Bytecode. Using the dx tool, the Java code is translated from Java bytecode to Dalvik bytecode. Dalvik bytecode must first be transformed into an intermediate format appropriate for code analysis in order to be analyzed for static analysis. A software program called Dexpler transforms Dalvik bytecode into Jimple. Dedexer (a tool for disassembling DEX files) and Soot serve as the foundation for the Dexpler (framework for analyzing and transforming java and Android applications).

2.7 IFDS

The inter-procedural, finite, distributive, subset (IFDS) analysis framework is a dataflow analysis approach for handling IFDS issues. Flow functions f must be distributive over the union meet operator (i.e., f(a) Uf(b) = f(a U b)) and are defined over a finite domain of dataflow facts D. These flow functions are established by the particular analysis (in this example, the taint analysis), which identifies the impact of the statement's execution at the specified program point on the dataflow facts. Dataflow issues are effectively resolved by the IFDS analysis approach by being reduced to graph reachability issues. A node's reachability in the graph indicates if a specific dataflow fact is true at a specific program step.

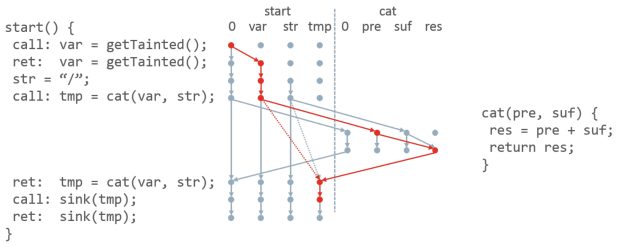

Figure 2. Simple IFDS Taint Analysis

The IFDS analysis framework uses function summarization to accomplish effective inter-procedural analysis. Function summaries are produced as needed during the analysis and show how far back a collection of start facts may be reached from an end fact. For a single function, many summaries may be produced, one for each important end fact. The inter-procedural controlflow graph (ICFG) that an application's exploding supergraph is based on is used by IFDS. Each ICFG node is broken down into as many nodes as there are dataflow facts. Listing 2 displays an illustration of an IFDS analysis—in this case, a straightforward taint analysis—applied to the application in question:

Listing 2. Taint Analysis

```
1 void start() {
2   String var = getTainted();
3   String str = "/";
4   String tmp = cat(var, str);
5   sink(tmp);
6 }
7 String cat(String pre, String suf) {
```

```
8   String res = pre + suf;
9   return res;
10}
```

If and only if the related node in the exploding supergraph is accessible, a certain dataflow fact holds at a specific statement. The flow 2 functions are encoded by the edges in the exploding supergraph. In Figure 2, the existence of an execution route, marked in red, where tainted data reaches a security-sensitive sink is shown by the fact that the sink node (sink(tmp), tmp) is reachable from the entrance node (var = getTainted(), 0).

3 Methodology

News says applications (example: Universal TV Remote Control, Find My Kids: Child Cell Phone Location Tracker, Hybrid Warrior: Dungeon of the Overlord Read, Remote for Roku: Codematics, Novan Health) are compromising user data, and we can get this apps from play store. Also there are sample apps in DroidBench which helps us to identify the potential risks in codes written in Java (Android) as a testbed to experiment with the study. We use taint analysis tools (FlowDroid) to identify the possible source of leakages.

The tools utilize a configuration text file that holds the source/sink information for the data flow analysis. The definition of the source/sink depends upon the problem that is intended to be identified. The analyzer will detect leakages based on the configuration file data.

We run FlowDroid on the Application source file (i.e. the apk file) to detect the possible leakage. Then from the output of FlowDroid leakage information, we extract the functions which are having malicious intent. Afterwards, we check the source code decompiled by ApkTool independently to verify the actual leakage, find the relationship of the leaked data to creator intent and possible attacker exploitation. Eventually, what could be the countermeasure can be decided from this data.

We have gathered possible sources of PHI data from Android Developer [2] APIs. Table 1 shows those identified APIs with parameters and sensitive information.

We have also extracted some possible sink functions (in table 2) from native android library usage to pop out the data outside of the app. This sinks can get the data from sources and publish the data to outside of the app environment for exploitation of attackers. Sometimes there can be false positives like if a media recorder app actually gets permission from user to record from environment initiated by user.

We map this apis to our 18 PHI data points mentioned. Table 3 shows the relationship of PHI data to Android source methods. If any of these methods are triggered by FlowDroid analysis we see the actual java implementation from decompiled file and analyze the possible countermeasure and possible intent revelation from both the creator and attacker side.

4 Implementation

FlowDroid uses a SourceSink.txt file mentioned before. The definition file for sources and sinks defines what shall be treated as a source of sensitive information. For example, the following method is a piece of sensitive information as it takes Latitude from Location services.

```
<android.location.Location: double getLatitude()>
```

Another property of that file is what shall be treated as a sink that can possibly leak sensitive data to the outside world. An example of sink can be which Logs location data runtime.

```
<android.util.Log: int d(java.lang.String, java.lang.
    String)> ("Latitude", $r2)
```

We have written a python script to run this analysis one by one on the apps we have in our dataset. We have organized the dataset as cleanly as possible so that after running the analysis the output is as human readable as possible. Our current database has around 50 applications. https://github.com/ashfakshibli/PHI_FLOW

Listing 3. FlowDroid leakage identitfication

```
[main] INFO soot.jimple.infoflow.android.
    SetupApplication$InPlaceInfoflow
- The sink staticinvoke <android.util.Log: int d(java.
    lang.String,java.lang.String)> ("Longtitude", $r2
    ) in method <de.ecspride.LocationLeak1: void
    onResume()> was called with values from the
    following sources:

[main] INFO soot.jimple.infoflow.android.
    SetupApplication$InPlaceInfoflow - - $d1 =
    virtualinvoke $r1.<android.location.Location:
    double getLongitude()>() in method <de.ecspride.
    LocationLeak1$MyLocationListener: void
    onLocationChanged(android.location.Location)>

[main] INFO soot.jimple.infoflow.android.
    SetupApplication$InPlaceInfoflow - Data flow
    solver took 0 seconds. Maximum memory consumption
    : 449 MB

[main] INFO soot.jimple.infoflow.android.
    SetupApplication - Found 2 leaks
```

Sometimes a property or method can be both Source and Sink. These definitions are specific to our use case. By these location-related source sinks, we will be able to identify leakage of Location information outside of the application. We have applied this on a example application in Droidbench named “LocationLeal1.apk”. FlowDroid generates callgraphs and uses IFDS to taint the decompiled apk file. Finally, it could determine above two lines consist of leakage. This

Table 1. Sensative data Source APIs

Source Method	Returns	Data Carries
getSubscriberId	String[]	Sim card Subscriber Info
getIpAddress	int	IP address
getAuthToken	AccountManagerFuture	Authentication token
getAllCellInfo	List	Information
getCellLocation	CellLocation	Cell tower information
getDeviceId	String	DeviceId
getLine1Number	String	Sim1 number
getVoiceMailNumber	String	Voice Mail Number
getLastKnownLocation	Location	GPS cordinates
getAltitude	double	GPS altitude
getLatitude	double	GPS latitude
getLongitude	double	GPS longitude
getAccounts	Account[]	Account information
getPassword	String	Password
getRunningAppProcesses	List	Running process information
getWifiMacAddress	String	Wifi Mac Address
getConnectedDevices	BluetoothDevice	Remote Device Name
getRemoteDevice	String	Password
getSensorList	Cipher	Cipher info
getCipher	String	Password
getDetailedState	NetworkInfo.DetailedState	Phone state info
getHost	String	Host Address
getPort	int	Port Number
getKey	SecretKey	Key

Table 2. Sensative data Sinks

Class	Sink Function
org.json.JSONObject	put
org.apache.http.client.methods.HttpPost	execute
android.util.Log	int d
java.io.OutputStreamWriter	append
android.telephony.SmsManager	sendTextMessage
android.content.Context	sendBroadcast
android.media.MediaRecorder	start
android.content.Intent	putExtra
java.io.OutputStream	write
android.graphics.Canvas	drawText
android.graphics.Bitmap	setPixel
android.os.Bundle	putBinder
org.apache.http.message.BasicNameValuePair	>
java.util.Map	put
android.media.ExifInterface	setAttribute
org.json.JSONObject	accumulate

makes a promising outcome for our proposal though it is a very small part of the whole idea.

However if we are looking for privacy issues like 18 PHI data points in our case, we are using FlowDroid's default file "SourcesAndSinks.txt" in the "soot-infoflow-android" in

Table 3. PHI Data Mapping with Source/Sinks

PHI Data Point	Source/Sink Function
name	getAccounts| android.util.Log | sendTextMessage
address	android.util.Log | sendTextMessage
dates	android.util.Log | sendTextMessage|
phone number	getLine1Number|getVoiceMailNumber
email address	sendTextMessage|android.util.Log|
Social Security number	android.util.Log
medical record number	android.util.Log
health plan beneficiary number	android.util.Log
account number	android.util.Log
certificate or license number	android.util.Log
vehicle identifiers, such as serial numbers, license plate numbers	android.util.Log
device identifiers and serial numbers;	getDeviceId
web URL	getHost | android.util.Log | sendTextMessage
Internet Protocol (IP) address	getHost | getPort
biometric IDs, such as a fingerprint or voice print	getKey
full-face photographs and other photos of identifying characteristics	getKey
any other unique identifying characteristic	android.util.Log

FlowDroid as a starting point and expand around that. This source/sinks is identified properly from possible source APIs in Android Platform which have potential leakage. We could get that data by decompiling the Android apps and getting the XML, and Java class-method signature files using Apktool [4] by examining many applications. But as our focus is mainly to generate dataset we have taken the source/sinks from Android platform given in FlowDroid. We look for potential leakage generative code from our applications only to verify the leakage and formulate possible intents. As we are doing this manually there can be imprecise data but that can be verified by future work on our organized dataset.

5 Formulation of our dataset

We present our work by testing apps from 50 app dataset which may contain PHI data. We check the apps with FlowDroid's source/sinks and more sources sinks we find in the process of empirical checks (if found). Then we manually check false positives from actual decompiled java code. We also analyze the pattern of the leaks and the intent empirical manner. We decide the intent using poissible threats of exposing particular data from research [22] and exploit news for the particular type of applications [17]. Our dataset will consist of final data in tabular form shown in Table 4 which we gather empirically as mentioned. This table grows from here with other analyzed applications with proper dataset to work on in future research.

6 Future Work

We can feed our empirical data to a machine learning model to find similar intent based applications and those applications can be marked as malicious easily and efficiently. And Google Play or regulatory agencies can warn the user about this applications to prevent PHI data leakage or block applications instantly before any major exploit happens.

References

[1] 2019. Global developer report: DevSecOps finds security roadblocks divide teams. https://about.gitlab.com/blog/2019/07/15/global-developer-report/

[2] 2022. Android mobile App Developer tools. https://developer.android.com/

[3] 2022. Android Statistics. https://www.businessofapps.com/data/android-statistics/

[4] 2022. Apktool - reverse engineer Android APK files. http://www.javadecompilers.com/apktool

[5] 2022. Static program analysis. https://en.wikipedia.org/wiki/Static_program_analysis

[6] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid. *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2014). https://doi.org/10.1145/2594291.2594299

[7] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. 2012. Dexpler. *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis - SOAP '12* (2012). https://doi.org/10.1145/2259051.2259056

[8] Eric Bodden. 2012. Inter-procedural data-flow analysis with IFDS/IDE and soot. *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis - SOAP '12* (2012). https://doi.org/10.1145/2259051.2259052

[9] Resha Chheda. 2022. What is DLP? data loss prevention for Critical Business Information. https://www.exabeam.com/dlp/data-loss-prevention-policies-best-practices-and-evaluating-dlp-software/

[10] Mike Fong. 2021. The unforeseen risks of sharing smartphone location data. https://www.securitymagazine.com/articles/96557-the-unforeseen-risks-of-sharing-smartphone-location-data

[11] James Gelinas. 2020. See which apps are collecting the most data on you each day. https://www.komando.com/security-privacy/data-grabbing-apps/762166/

[12] Lisa Gentes-Hunt. 2021. Covid-19 pandemic sparks upswing in healthcare data breaches. https://healthitsecurity.com/news/covid-19-

Table 4. Dataseet Insight

APK	Type	Leak Count	Place	Probable Developer Intent	Probable Attacker Exploit
`com.universalremote.app`	Utility	4	Location Module	Mistake	Attack User using location point
`com.abc.malware [example]`	Heart Rate Monitor	6	Location Module	Potential Malicious Intent	User data broker

pandemic-sparks-upswing-in-healthcare-data-breaches

[13] Ondřej Lhoták and Laurie Hendren. 2003. Scaling java points-to analysis using Spark. *Lecture Notes in Computer Science* (2003), 153–169. https://doi.org/10.1007/3-540-36579-6_12

[14] Ben Lutkevich, Scott Wallask, and Alex DelVecchio. 2021. What is phi (protected/personal health information)? https://www.techtarget.com/searchhealthit/definition/personal-health-information

[15] No Name. 2022. Mobile app statistics to keep an eye on in 2022. https://cybersecurity.asee.co/blog/mobile-app-statistics-to-keep-an-eye-on/

[16] Ben Joseph Philip, Mohamed Abdelrazek, Alessio Bonti, Scott Barnett, and John Grundy. 2022. Data collection mechanisms in health and wellness apps: Review and analysis. *JMIR mHealth and uHealth* 10, 3 (2022). https://doi.org/10.2196/30468

[17] Brian Reed and Andrew Hoog. 2018. U.S. Soldiers and Athletes using STRAVA social fitness apps: Finding Risky Mobile Apps that Share and Leak Data. https://www.nowsecure.com/blog/2018/01/30/soldiers_strava_fitness_risky_mobile_apps_leak_data/

[18] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '95* (1995). https://doi.org/10.1145/199448.199462

[19] Emma Roth. 2022. Meta fined $276 million over Facebook data leak involving more than 533 million users. https://www.theverge.com/2022/11/28/23481786/meta-fine-facebook-data-leak-ireland-dpc-gdpr

[20] ScienceDirect. 2022. Data Leakage Prevention - an overview | ScienceDirect Topics. https://www.sciencedirect.com/topics/computer-science/data-leakage-prevention

[21] TIMESOFINDIA.COM. 2021. These Android apps can leak your private messages, email addresses - times of India. https://timesofindia.indiatimes.com/gadgets-news/these-android-apps-can-leak-your-private-messages-email-addresses/articleshow/86683720.cms

[22] Paweł Weichbroth and Łukasz Łysik. 2020. Mobile security: Threats and best practices. *Mobile Information Systems* 2020 (2020), 1–15. https://doi.org/10.1155/2020/8828078

[23] Zack Whittaker. 2019. Samsung spilled SmartThings app source code and Secret Keys. https://techcrunch.com/2019/05/08/samsung-source-code-leak/

[24] Rui Zhou, Mohammad Hamdaqa, Haipeng Cai, and Abdelwahab Hamou-Lhadj. 2020. MobiLogLeak: A preliminary study on data leakage caused by poor logging practices. *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2020). https://doi.org/10.1109/saner48275.2020.9054831

Received October 2022; revised 30 November 2022