An Empirical Study of Potentially Malicious Third-Party Libraries in Android Apps

Zicheng Zhang[1], Wenrui Diao[2,3]*, Chengyu Hu[2,3], Shanqing Guo[2,3]*, Chaoshun Zuo[4], and Li Li[5]
[1]School of Computer Science and Technology, Shandong University
[2]School of Cyber Science and Technology, Shandong University
[3]Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University
[4]The Ohio State University [5]Monash University
zhangzicheng1010@gmail.com,{diaowenrui,hcy,guoshanqing}@sdu.edu.cn,zuo.118@osu.edu,Li.Li@monash.edu

ABSTRACT

The rapid development of Android apps primarily benefits from third-party libraries that provide well-encapsulated functionalities. On the other hand, more and more malicious libraries are discovered in the wild, which brings new security challenges. Despite some previous studies focusing on the malicious libraries, however, most of them only study specific types of libraries or individual cases. The security community still lacks a comprehensive understanding of potentially malicious libraries (PMLs) in the wild.

In this paper, we systematically study the PMLs based on a large-scale APK dataset (over 500K samples), including extraction, identification, and comprehensive analysis. On the high-level, we conducted a two-stage study. In the first stage, to collect enough analyzing samples, we designed an automatic tool to extract libraries and identify PMLs. In the second stage, we conducted a comprehensive study of the obtained PMLs. Notably, we analyzed four representative aspects of PMLs: library repackaging, exposed behaviors, permissions, and developer connections. Several interesting facts were discovered. We believe our study will provide new knowledge of malicious libraries and help design targets defense solutions to mitigate the corresponding security risks.

CCS CONCEPTS

• **Security and privacy → Software and application security**.

KEYWORDS

Android apps, malicious third-party libraries, malware

ACM Reference Format:
Zicheng Zhang, Wenrui Diao, Chengyu Hu, Shanqing Guo, Chaoshun Zuo, and Li Li. 2020. An Empirical Study of Potentially Malicious Third-Party Libraries in Android Apps. In *13th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec '20), July 8–10, 2020, Linz (Virtual Event), Austria.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3395351.3399346

*Corresponding authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
WiSec '20, July 8–10, 2020, Linz (Virtual Event), Austria
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8006-5/20/07…$15.00
https://doi.org/10.1145/3395351.3399346

1 INTRODUCTION

According to the recent statistics [38], as of Feb 2020, the market share of Android has reached 73.3%. The rapid development of Android apps primarily benefited from third-party libraries that provide well-encapsulated functionalities, such as social network services, map services, and advertisement services. However, the convenience of third-party libraries does not come without risk.

Recently, the analysis of potentially malicious libraries (*PMLs* for short) has attracted lots of attention, and some recent works have made the first attempts. Here we summarize some typical ones:

- Wu et al. [40] found that advertisements or analytics SDKs could introduce open ports without developers' awareness.
- Li et al. [33] considered that some mutations of third-party libraries potentially indicate malicious behaviors.
- Chen et al. [17] found that some popular libraries may be repackaged with malicious payloads by the adversary.
- Pan et al. [37] found that a time bomb within the library `com.baidu.kirin` shared by 9,710 apps.

Previous related works provide solid ground for understanding the security implications of third-party libraries. However, most of them focused on the advertising libraries or just individual case studies, lacking a comprehensive study on the behaviors and features of PMLs. Several research questions are still unanswered. For instance, how many PMLs are repackaged from benign libraries? Who developed these PMLs? On the other hand, malicious libraries could bring severe security risks to both mobile users and app developers. Without an in-depth understanding, the targeted defense solutions cannot be designed and deployed effectively.

Our work. To fill this research gap, in this paper, we carried out the first empirical study on PMLs based on a large-scale APK dataset, including extraction, identification, and comprehensive analysis. On the high-level, we conducted a two-stage study.

The first stage is to extract and identify enough PMLs as analyzing samples. We designed an automated tool to achieve this target, called LibExtractor. Its design idea is based on extracting and clustering library candidates from large-scale apps. LibExtractor improves the efficiency issue and extends additional PML identification modules. It combines the report from VirusTotal [11] and results of behavior matching to identify PMLs. We also define a list of potentially harmful behaviors as the criterion for behavior matching. As a result, LibExtractor can efficiently extract libraries without previous knowledge from large-scale apps and identify PMLs.

The second stage is to conduct a comprehensive analysis of the PMLs (and the apps containing PMLs) obtained from the previous stage. In particular, we focused on the following four aspects:

1) *Library Repackaging.* If a PML is created by modifying a benign library, its impact may be very severe, especially when the victim library belongs to a popular SDK. To identify the repackaged libraries, we designed a light-weight classification approach.

2) *Exposed Potentially Harmful Behaviors.* In Android, the exposed components (Activities, Services, etc.) can be accessed by other apps without permission. To PMLs, exposed behaviors also could be exploited to conduct confused deputy attacks [22]. We measured the amounts of exposed potentially harmful behaviors in PMLs.

3) *Permissions.* As the fundamental protection mechanism of Android, permissions provide the capabilities of accessing sensitive system resources for PMLs. We measured the frequently used sensitive permissions of PMLs.

4) *Developer Connections With Apps.* The sources and authors of PMLs are important clues for identifying malware families. Based on app certificates, we analyzed the developer connections between PMLs and the corresponding apps.

In practice, our experiments and analysis are based on a large-scale APK dataset containing over 500K apps (217,027 apps and 316,437 malware samples). Several interesting facts are discovered. For example, tracking users (through device ID) is the most common behavior of PMLs. Also, a popular benign library may be used to generate multiple repackaged PMLs.

Contributions. This paper has the following contributions:

- *Systematic Study.* We systematically studied the behaviors and features of PMLs, including library repackaging, exposed behaviors, permissions, and developer connections.
- *New Tool.* We designed an automatic analysis tool, LibExtractor, for PML extraction and identification. It can extract PMLs without previous knowledge.
- *Large-scale Investigation.* We carried out large-scale experiments on over 500K APK files. 4,957 PMLs were discovered and used for the subsequent analysis.

Roadmap. The rest of this paper is organized in the following order: In Section 2, we give the methodology and necessary knowledge of our work. Section 3 describes the design of LibExtractor, including library extraction and PML identification. In Section 4, we demonstrate the results of PML identification and tool evaluation. Section 5 analyzes four aspects of PMLs in-depth. Section 6 discusses some limitations of this paper, and Section 7 reviews previous works. Section 8 concludes this paper.

2 BACKGROUND AND METHODOLOGY

In this section, we illustrate the motivation and methodology for analyzing third-party potentially malicious libraries. Besides, the necessary background will be provided.

2.1 Running Example

In our preliminary study, we discovered a library named `com.wonderтek`, which can be extracted from more than 50 apps. Also, all of these apps are developed by the same developer – Migu, a subsidiary company of China Mobile (the largest mobile network operator in China). This library has plenty of suspicious behaviors such as obtaining the contact information and monitoring the calling status of the phone. It also can download unwanted APKs, and even further install them in the background (by executing shell command), which may waste the user's data traffic and cause additional costs. Also, we uploaded 5 of these APKs to VirusTotal [11], an online malware detection platform, and the result shows that, on average, 29 anti-virus engines reported these APK files contain a Trojan named `wondertek`. Since China Mobile is a well recognized large company, it is quite unusual to find such a malicious library in its apps. This case motivates us to explore the malicious third-party libraries in the wild. We find several aspects of malicious libraries have not been well studied, such as their behaviors, developers, and other security implications. The research gap needs to be filled immediately.

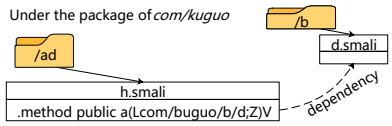

Figure 1: Example of method dependency.

2.2 Methodology

Third-party libraries are widely used in Android app developments to accelerate the development period. On the other hand, more and more malicious libraries are discovered in the wild. These malicious libraries cannot be fully understood without a systematic study. In our study, we try to investigate several aspects of malicious libraries, such as behaviors, repackaging issues, exposed components, permissions, and developer connections.

To provide first-hand knowledge, as the first step, we need to extract and identify potentially malicious libraries (PMLs) based on a large-scale APK dataset. After that, we will analyze these PMLs in-depth to answer valuable research questions.

2.3 PML Extraction and Identification

To capture PMLs from malware samples in the wild, we need to design a tool to achieve automatic PML extraction and identification. It means this tool can extract libraries from large-scale apps directly without previous knowledge and identify PMLs efficiently.

After reviewing the previous research, we did not consider the tools based on a pre-defined database because they cannot identify unknown libraries (not existing in their database), such as LibScout [15], LibRadar [34], and LibPecker [42]. We noticed a library extraction tool named LibD [33], which could meet some of our requirements. However, it also has some limitations, which significantly affect the analysis efficiency, as the below explains:

- **Speed.** LibD generates feature values of libraries by hashing the opcodes within basic blocks after control flow graph construction. However, the process of the CFG construction and subsequent calculation is very time-consuming.
- **Completeness.** LibD uses file inclusion, class inheritance, and call graph to construct library candidates. However, these relations are not enough and may cause a lack of some components in the library candidates. For example, Figure 1

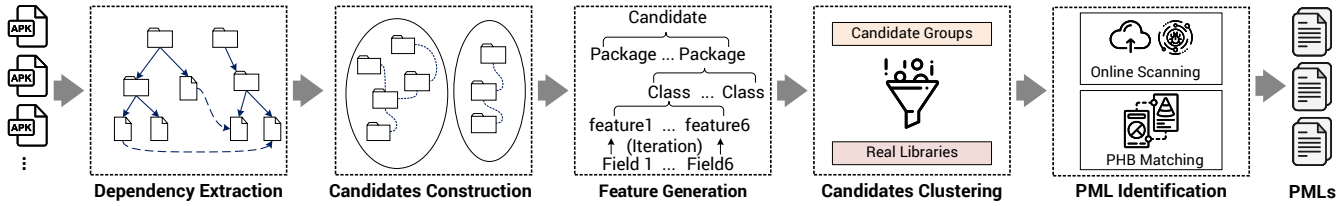

Figure 2: Overview of LibExtractor.

shows the parameter dependency of a method, while LibD omits this dependency during the candidate construction.

Our Tool: LibExtractor. To solve the efficiency issue and meet our requirements for large-scale app scanning, we design LibExtractor. It can quickly extract libraries from large-scale apps and perform PML identification. It uses all possible dependencies between classes so that the completeness of library candidates can be guaranteed. We also design a special algorithm to generate the feature value which greatly improves the speed. Also, it can handle the package name obfuscation which is widely deployed by malware apps. Besides, since our PML analysis is based on potentially harmful behaviors (defined later), we combine the result of the scanning report from VirusTotal [11] and behavior matching to identify PMLs. The detailed design will be illustrated in Section 3.

Potentially Harmful Behaviors. Here we list several types of *potentially harmful behaviors* that may exist in PMLs. This behavior list is based on the behaviors of `com.wondertek` library and the detection rules of some malicious code detection tools [4, 5]. Also, we referred to the behavior list used by Gamba et al. [23] while they were analyzing the behaviors of pre-installed apps. We define 8 types (with 23 sub-types) of potentially harmful behaviors: (each sub-type contains several APIs)

(1) *Telephony*: SMS sending, SMS interception, and calling.
(2) *Location*: accessing GPS, LAC (location area code), CID (cell identity), and cell location.
(3) *Phone Identifier*: reading SIM and phone information.
(4) *Network & Connection*: accessing network connection and reading network configuration information.
(5) *Code Execution*: library loading and Linux command execution.
(6) *Media Interception*: accessing audio and video.
(7) *Personal Information*: reading SMS, MMS, contact, clipboard, calendar, Email, voicemail, and phone number.
(8) *App Installation*: accessing package information.

2.4 PML Analysis

In the analysis phase, we studied four aspects of PMLs: library repackaging, exposed potentially harmful behaviors, permissions, and developer connections with apps.

2.4.1 Library Repackaging. If a PML is created by modifying a benign library without potentially harmful behaviors, we treat this library as a *repackaged library*. The malicious payload may be introduced by developers (unguarded library updating operations) or attackers (intentional attacks). In our investigation, we try to answer "*how many PMLs are repackaged from benign libraries*".

2.4.2 Exposed Potentially Harmful Behaviors. In the manifest file of an Android app, a component could be set as "`android:exported=true`" to make it exposed. These exposed components could be accessed or invoked by any other apps. In this paper, we define *exposed potentially malicious behaviors* as the behaviors that can be accessed by the exposed components through the call paths on the call graph within each library. This operation allows the adversaries to inject payload containing those behaviors into the PMLs, and then access the payload through the exposed components. In our investigation, we try to answer "*does there exist any exposed potentially malicious behaviors in PMLs*".

2.4.3 Permissions. Android uses the permission mechanism to restrict apps to access system resources. In Android 6.0 and later versions, when an app needs sensitive permissions, it should not only declare them in its manifest file but also request the permissions while running [6]. Some malicious libraries may pretend that the app needs certain functions and asks users to grant sensitive permissions for them. The permission used by the PMLs, and their corresponding apps, which is highly related to potentially harmful behaviors, are still worthy of study. In our investigation, we try to answer "*which permissions are frequently used in PMLs*".

2.4.4 Developer Connections With Apps. Though it is difficult to obtain the developer's information from the libraries directly, there might be some connections between the developers of the libraries and their corresponding apps. Therefore, we focused on the connections based on the bipartite graph between PMLs and the signatures of their corresponding apps. In our investigation, we try to answer "*what about the connections between the PMLs and their authors*".

3 SYSTEM DESIGN

The purpose of this paper is to carry out a comprehensive study on the potentially malicious libraries in the wild. To capture massive PML samples, in this section, we present the detailed design of our PML extraction and identification tool – LibExtractor. Our PML analysis is based on the results of LibExtractor. On the high-level, LibExtractor contains five steps (as illustrated in Figure 2):

(1) **Dependency Extraction**: Disassemble the input APK file and extract six kinds of dependencies and the file structure from the Smali code.
(2) **Candidate Construction**: Build an inclusion-dependency graph based on the dependencies between classes and remove redundant packages. Then construct library candidates by dividing the connected components in the graph.

(3) **Feature Value Generation**: Generate a unique feature value for each library candidate, which is based on a particular algorithm using the dependency field in each class contained by this candidate.
(4) **Candidate Clustering**: Identify real libraries from candidates using the clustering-based method, which is suitable for large-scale analysis.
(5) **PML Identification**: Compare the results of VirusTotal scanning and potentially harmful behavior matching to identify PMLs.

3.1 Dependency Extraction

In this step, we disassemble the input APK to get all its class dependencies and package structures over the Smali codes using baksmali [8]. The package structure is a file tree, of which leaf nodes are class files, and the other nodes are packages.

We extract six kinds of dependencies in each class file. Here we define, if Class *B* exists in one of the 6 fields listed below in Class *A*, then we regard Class *B* as a dependency of Class *A*:

- *superclass*. Class *A* inherited from Class B.
- *interface*. Class *A* implements Interface *B*.
- *staticField*. Class *B* (as a variable type) exists in the static field of Class *A*.
- *instanceField*. Class *B* (as a variable type) exists in the instance field of Class *A*.
- *directMethod*. Class *B* (as a variable type) exists in the parameters or return values of static, private, and constructor methods of Class *A*.
- *virtualMethod*. Class B (as a variable type) exists in the the parameters or return values of virtual methods of Class *A*.

3.2 Candidate Construction

In this step, we construct library candidates by dividing connected components in an inclusion-dependency graph, with auxiliary information extracted from the manifest file of the input APK.

Inclusion-Dependency Graph. We construct an inclusion-dependency graph based on the six kinds of dependencies and package structure of the app. The nodes in this graph are the packages in the package structure obtained in Section 3.2. Also, it is an undirected graph, and we define the edges as follows: if any dependency exists between the file A_f contained in the package A_p and the file B_f contained in the package B_p, we define that there is an edge between node A_p and node B_p.

Eliminate Redundant Packages. Android official libraries (e.g. `android.support.v4`, etc.) and packages containing apps' own codes and their dependencies will cause multiple libraries connecting to each other in the inclusion-dependency graph. Moreover, they will be grouped into one component (library candidate) while dividing connected components, which will affect the accuracy of our tool's result. Besides, some packages in the app may not contain any files (may contain sub-packages), such as some root packages or middle-level packages. We define these packages as *redundant packages*, and they will be eliminated from the graph. Note that we identify an app's own packages based on the package names of this app shown in its manifest file.

Candidate Construction. To construct library candidates, we divide weakly connected components on the Inclusion-Dependency Graph, and we define each component as a library candidate. Each component contains one or several packages that are connected by the edges defined above. Since there is no dependency between two different components, each component has standalone functionality, which is consistent with the characteristics of a library.

3.3 Feature Value Generation

In this step, LibExtractor generates the feature value of each library candidate for the clustering step in Section 3.4. The candidate feature value is calculated by the feature values of its packages, which can be further divided into class feature values, as shown in the third step in Figure 2. Because the dependencies between classes are complex but easy to obtain, we can quickly obtain unique feature value for each class based on its dependencies.

Class Feature Value. To ensure that each class has a unique feature value and still maintain resilience to package name obfuscation, we design an algorithm to generate the feature values (in the form of hash values), which contains four rounds of calculation. In each round, we will calculate a feature value for every class file in a candidate:

(1) **First Round**. In each class of a candidate, we obtain a special relative path from this class to each of its dependency classes. For example, if `x/y/z` is a dependency of `x/x/x` in the corresponding dependency field, then we will get a relative path `../../A/A`. We use the letter `A` to replace all the package names and file names in this relative path so that the package name obfuscation will not affect the feature value. In each of the six dependency field (Section 3.1), we sort these relative paths by the order of their length, and the sorted relative paths are concatenated into a single space-separated string. If some classes do not exist in the entire file structure, such as `java.lang.String`, etc., we will use their class names directly instead of relative paths. Then, we concatenate the strings of the six dependency fields into one string and hash it to generate the feature value of each class in the first round.
(2) **Next Three Rounds**. In each of the next three rounds, for each dependency field in a class, we concatenate the dependency classes' corresponding feature values (generated in the previous round) into a single string (in the same order as the first round). We directly use the name of those dependency classes that do not exist in the file structure. Then we concatenate the strings of these six fields into a single string and hashed this string to generate the new feature value of this class. For each class, the feature value calculated in the last round will be its *final feature value*, which will be used to calculate the feature values of packages and the candidate.

A simple example of the first and second round calculation is shown in Figure 3. This example shows that even if Class `a1` and `a2` share the same feature value in the first round, their second-round feature values can be different. As a result, each class can get a unique feature value. Since dependencies between classes are easy to obtain and there are no complex operations such as handling

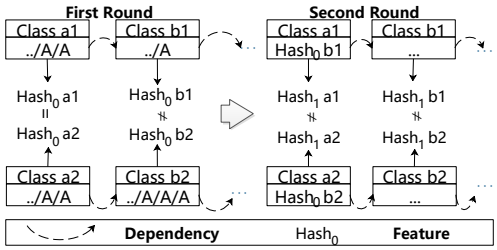

Figure 3: An example of the first two rounds' calculation.

basic blocks, so LibExtractor can quickly generate a feature value for each class.

Package and Candidate Feature Value. To each package in a candidate, we sort the final feature values of all the classes contained by this package in non-decreasing order (numerical order) and concatenate them into a single string. We then hash this string to get the package feature value. And similarly, we sort and concatenate feature values of packages into a single string and hash this string to generate the feature value of this candidate.

3.4 Candidate Clustering

In this step, LibExtractor clusters the candidates using the same way as LibD [33]. The candidates sharing the same feature value are put into the same group. Further, if the number of candidates in one group is equal to or greater than a pre-defined threshold, we consider that this group indicates a real third-party library. The selection of the threshold will be described in detail in Section 4.2.

Obfuscation. Since we use relative paths instead of the class names while generating the feature value for each candidate in Section 3.3, LibExtractor is resilient to package name obfuscation. It can recover the original package names of some of the obfuscated libraries by selecting the most appropriate candidate from each group, for example:

```
0001aad2...ee9b687cebe945 - com/catstudio/plugin
0001aad2...ee9b687cebe945 - com/e/a
0001aad2...ee9b687cebe945 - com/catstudio/c
```

In this example, we select the candidate that has the longest package name in a group to represent the original name, say `com/catstudio/plugins` is recovered. In practice, LibExtractor can recover the obfuscated package names, if a group of candidates meets the following three conditions: (1) All candidates in this group have the same amount of packages. (2) All candidates in this group are not entirely obfuscated (e.g., `a/b/c`). (3) The package names of all candidates share the same prefix. Other obfuscation situations will be discussed in Section 6.2.

3.5 PML Identification

LibExtractor extracts libraries from large-scale malicious apps and then identifies potentially malicious libraries from them. We use a simple method to deal with this problem. It includes 2 parts:

(1) Upload libraries to a malware detection platform for online scanning to obtain PMLs.
(2) Perform a behavior matching based on a predefined list to get PMLs using potentially harmful behaviors.

Table 1: Markets of dataset

Markets	# of APKS	Markets	# of APKs
Apps from Third-party Markets			
Anzhi	54,761	NDuo	19,532
Mumayi	22,085	Angeeks	25,675
Wandoujia	61,197	UnKnown	33,777
		Total	217,027
Malware Samples			
		Androzoo	316,437

3.5.1 Online Scanning. We upload all the libraries to VirusTotal [11], a platform for malware detection using dozens of Anti-Virus engines. We follow the method used by Chen et al. [18], that is, for each library, if two or more engines report it as a malware, then we record this library as a PML.

3.5.2 Behavior Matching. Given the potentially harmful behavior list described in Section 2, to each PML, we match the API of potentially harmful behaviors in each class file and record the amount and locations of these behaviors. Then we collect all the PMLs that contain these behaviors. The reason for performing this step is that we only care about those PMLs having potentially harmful behaviors, thus we can further perform the analysis based on these behaviors in Section 5.

4 LIBRARIES EXTRACTION AND PML IDENTIFICATION

We implemented a prototype of LibExtractor with 8,598 lines of Java code and 196 lines of Python code. To demonstrate the effectiveness of LibExtractor, we also carried out large-scale experiments on real-world data. In this section, we present tool evaluation and the results of potentially malicious libraries. The subsequent analysis will be based on the results of this section.

4.1 Dataset

We collected a large number of malware samples (malicious apps) with the support of Androzoo Project [2], a collection of Android apps (APKs) built by the University of Luxembourg. Every app in Androzoo has been detected by several anti-virus software to label it as benign or malicious.

On the other hand, to evaluate the performance of LibExtractor and make a comparison between PMLs and benign libraries, we also crawled many apps from several third-party app markets, such as Anzhi, Mumayi, and Wandoujia. Note that these apps are not always benign due to the lenient security criterion of markets. If we only adopt benign apps without malicious payload (and malicious libraries), it will be difficult to carry out our subsequent PML analysis. Since the libraries extracted from these apps are not always benign, we call them *Possibly Benign Libraries* (*PBLs* for short).

Finally, we obtained 217,027 apps and 316,437 malware samples as our dataset, as listed in Table 1.

Table 2: Unreported Libraries (in the whitelist).

Threshold	30	20	11	10	9	1
Unreported	9	7	3	2	2	2

4.2 System Parameter Setup

As mentioned in Section 3.4, we need to set a threshold when clustering candidates. Through comparing with the whitelist provided by Chen et al. [17], we evaluate the LibExtractor with different thresholds to determine the most suitable one. This whitelist contains 72 commonly used third-party libraries. For each threshold, we compared the groups of candidates and the libraries listed in the whitelist and gave an unreported number (appears in the whitelist but is not found in the result).

Considering that malicious apps may rarely use the libraries in the whitelist, we use the apps from third-party markets to determine the threshold. Generally, the higher the threshold is, the fewer libraries can be found. The whitelist comparison results are listed in Table 2. It shows that when the threshold decreases to 10, the unreported amount will drop to 2 and not decrease any more. Therefore, we chose 10 as the threshold and obtained 19,938 PBLs (from 217,027 apps) as contrast data for the subsequent experiments in Section 4.4 and 5.1.

4.3 Evaluation

We evaluated the performance and accuracy of the part of library extraction (i.e., Step 1-4 described in Section 3).

Processing Time. Based on the hardware configuration of Xeon E5-2630 2.30 GHz, 128 GB RAM, we applied LibExtractor and LibD to process 217,027 APKs. On average, LibExtractor takes 1.059 seconds to process each APK, while LibD takes 67.429 seconds.

Accuracy. Since there is no well-established ground truth of third-party libraries, we follow the method used by LibD [33] and use randomly selected subset to verify the accuracy of the library extraction step of LibExtractor. In detail, we randomly selected 1000 APKs from our dataset as a test subset. After that, we applied LibExtractor and LibD to extract libraries from these 1000 APKs separately and collect those libraries used by at least 10 APKs in this subset. Then we manually analyzed the extraction results to evaluate the accuracy. The result shows that LibExtractor output 55 libraries in total, with 54 correct libraries, while LibD output 54 libraries with only 48 correct libraries.

Obfuscation. In the 19,938 PBLs obtained in Section 4.2, we found 4,547 libraries with obfuscated package names. It means that LibExtractor is resilient to package name obfuscation. Further, LibExtractor recovered the package names of 1,123 libraries.

In summary, LibExtractor can effectively extract third-party libraries from large-scale APK dataset with good accuracy.

4.4 Potentially Malicious Library Identification

LibExtractor extracted 17,725 libraries from 316,437 malware samples, and further performed PML identification on these libraries.

4.4.1 VirusTotal Scanning. We packed each library to a DEX file and uploaded this file to VirusTotal [11], a platform for malware detection using dozens of anti-virus engines. Here we follow the method used by Chen et al. [18] while identifying PMLs, that is, if two or more engines reported the uploaded DEX file is malware, we will record this library in the result. In total, we recorded 5,492 libraries from VirusTotal's report.

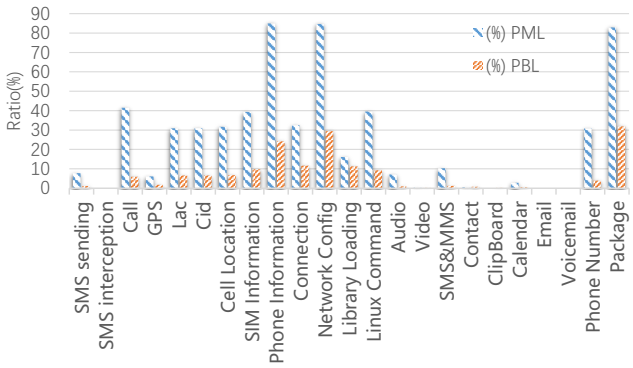

Figure 4: Libs containing potentially harmful behaviors.

4.4.2 Potentially Harmful Behaviors Matching. To the 5,492 libraries reported by VirusTotal, we performed behavior matching described in Section 3.5.2 and obtained 4,957 PMLs that contain potentially harmful behaviors.

After these two steps, 4,957 libraries were labeled as PMLs for subsequent analysis. Also, we recorded 207,137 malware samples that used at least one library from these 4,957 PMLs.

4.5 Potentially Harmful Behavior Comparison

To compare the difference of potentially harmful behaviors between PMLs and PBLs, we also performed behavior matching on the 19,938 PBLs obtained in Section 4.2. In Figure 4, we compare the number of potentially harmful behaviors in PMLs and PBLs. It shows the percentage of PMLs/PBLs containing a specific behavior. We can find, in most categories, the proportions of PMLs are significantly higher than the ones of PBLs, especially the sub-types of *Phone Information*, *Network Config*, *Package*, etc.

This figure also indicates that some potentially malicious behaviors are frequently used by PMLs, say more than 80%. For instance, the behaviors related to *Package* (82,87% vs. 31.81%) may be used to check whether a particular app has already been installed. In particular, malicious advertising libraries usually induce the user to click on the download links to obtain advertising revenue. The considerable difference in *Network Config* (84.71% vs. 29.36%) is also strong support for this point. In addition, the behaviors related to *Phone Information* (85.01% vs. 24.18%) are frequently used to identify victim users and track devices.

To summarize, the typical behaviors of PMLs include:

(1) Read the unique device ID to identify users.
(2) Access network for downloading files or transmitting data.
(3) Access device locations for locating users.
(4) Get the app installation list to check whether a particular app has been installed.
(5) Execute Linux shell commands.

5 ANALYSIS OF POTENTIALLY MALICIOUS LIBRARIES

In this section, we studied several aspects of PMLs and try to answer the following research questions.

- *How many PMLs are repackaged from benign libraries?*
- *Does there exist any exposed potentially harmful behavior in PMLs?*
- *Which permissions are frequently used in PMLs?*
- *What about the connections between the PMLs and their authors?*

The analyzed dataset is based on the **4,957 PMLs** and their corresponding **207,137 malware samples** obtained in the stage of PML identification (Section 4.4).

5.1 Library Repackaging

If a PML is created by modifying a benign library, its impact may be severe, especially when the victim library belongs to a popular SDK. Malicious payloads can be injected into massive apps with these libraries. Similar attacks ever occurred on the iOS platform [1].

Identify Repackaged Libraries. To identify those repackaged libraries in our PML set, we applied a lightweight method based on clustering the package names and feature values. In detail, we clustered the PMLs and PBLs sharing the same package list into a group and eliminated the libraries with duplicate feature values. At the same time, we also eliminated the obfuscated libraries which can not be recovered by LibExtractor. All libraries in the same group will be considered as mutations of the same library. After that, in each group, we counted the number of potentially harmful behaviors of each library. Once a PBL without such behaviors was found in a group, all PMLs in this group were considered as *repackaged libraries* of this PBL.

Results. Finally, we discovered 162 groups containing 2,257 libraries in total. Following our method, we further identified 218 repackaged libraries in 35 groups.

Impact. To measure the impact of these repackaged libraries, we also recorded the number of corresponding malware apps using these repackaged libraries in our dataset. For each repackaged library, we multiplied the number of its potentially harmful behaviors by the number of associated malware apps, and then used the product to evaluate the impact of this repackaged library. The top ten repackaged libraries with the most significant impact are listed in Table 3. In particular, up to 7 PMLs repackaged `net.youmi.android` and shared the same name. The repackaged PML ranked No. 2 (`com.mobclick.android`) affected the most apps (3,637).

Our assessment:

- At least 4.4% of PMLs are repackaged libraries.
- A popular benign library may be used to generate multiple repackaged PMLs.

Table 3: Top 10 repackaged libs with the largest impact.

No.	Origin Lib Name	Behaviors	APKs	Impact
1	net.youmi.android	47	1,947	91,509
2	com.mobclick.android	9	3,637	32,733
3	net.youmi.android	44	715	31,460
4	net.youmi.android	42	254	10,668
5	net.youmi.android	29	357	10,353
6	net.youmi.android	26	252	6,552
7	net.youmi.android	46	141	6,486
8	net.youmi.android	21	304	6,384
9	com.iapppay.pay	48	124	5,952
10	com.mobclick.android	9	593	5,337

5.2 Exposed Potentially Harmful Behaviors

In this sub-section, we identified exposed components in our PMLs and their corresponding apps. Further, we also identified exposed potentially malicious behaviors based on the call graph.

5.2.1 Exposed Components. To our PML analysis, *exposed components* are defined as the components (of a library) which could be accessed or invoked by other apps. It means these components declared one of the following parameters in the app's manifest file:

(1) `android:exported=true`;
(2) `intent-filter`;

In practice, we extracted the manifest files from the 207,137 malware apps containing PMLs and collected all the exposed components. Then we matched them with those components belonging to PMLs to obtain the exposed components of PMLs.

Results. After scanning, we discovered 101 PMLs containing 133 exposed components in total. Note that, since LibExtractor is resilient to package name obfuscation, some of the obfuscated package names were recovered to their original names. On the other hand, this operation brought some obstacles to the components matching between the apps and libraries. Therefore, it reduced the discovered number of libraries containing exposed components to some extent.

Particularly, `com.admogo.UpdateService` is an exposed component found in 23 PMLs, which were used by 761 apps. The package names of these 23 libraries are quite similar and may be the mutations of `com.admog` library. A method of this component is `downloadUpdateFile(String downloadUrl, File saveFile)` which can download files from a given URL. Since this component is exposed, this method may be called by other apps and download unknown files on the user's phone, which may pose a risk to users.

5.2.2 Exposed Harmful Behaviors. Our further analysis focused on the potentially harmful behaviors that could be accessed by the exposed components through the call paths. We identified those exposed behaviors through static taint analysis.

In the practical analysis, we built a call graph for each PML. Each node in the call graph is a method within the library. The edges of this call graph are the function calls between those methods. Also, we defined *sinks* as the methods containing potentially harmful behaviors, and the *sources* are the methods within exposed components. If the source could reach the sink, it means the corresponding potentially harmful behaviors in this sink are exposed.

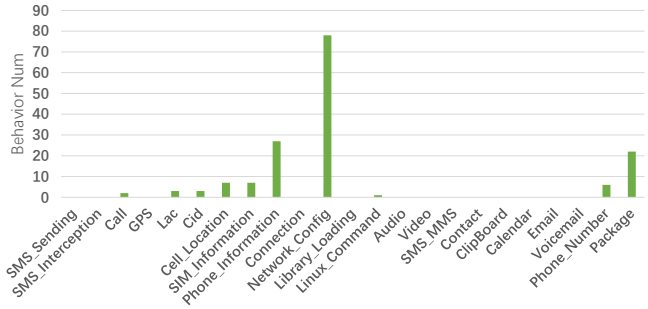

Figure 5: Exposed potentially harmful behaviors.

Note that, in Section 3.5.2, the locations of identified potentially harmful behaviors have been recorded. Thus, the sinks have been obtained. To obtain the call paths, we performed the backtracking algorithm from each sink to visit the nodes in the call graph. When a node(method) of an exposed component is visited, it will be regarded as a source, and we output those nodes in the reverse direction of the visit order as a call path (from the source to the sink). Each time when the algorithm could not find an unvisited node, it would terminate.

Results. Among the 101 PMLs containing exposed components, 40 PMLs contains 156 exposed potentially harmful behaviors, as shown in Figure 5. These 40 libraries were used by 21,756 malware apps, which accounts for 6.9% of all malware samples in our dataset. Specifically, 45 sources can reach 119 sinks through 286 call paths. Each sink contains multiple potentially harmful behaviors. 25 sources contain potentially malicious behaviors themselves, which means these behaviors may be directly accessed from other apps.

In Figure 5, to the exposed potentially harmful behaviors, most of them belong to the sub-categories of *Network Config*, *Phone Information*, and *Package*. Since these behaviors are the typical characteristics of PMLs, they may be exploited by other PMLs, which may lead to confused deputy attacks [22].

Besides, Hu et al. [25] mentioned that adversaries could inject a `BroadcastReceiver` into apps that can run malicious code while a specific event occurs. We also found such a case. An exposed component called `com.kuguo.ad.MainReceiver` exists in 11 mutations of the `com.kuguo` library. It can monitor apps' installation, screen lock, and network connection status changes, etc.

Our assessment:

- Exposed components are not common in PMLs. In total, 156 exposed potentially harmful behaviors are identified in 4,957 PMLs.
- About 6.9% of malware samples can be affected by exposed potentially harmful behaviors.

5.3 Permissions

In this sub-section, we investigated the permissions requested by the 4,957 PMLs and the corresponding 207,137 malware samples.

5.3.1 Declared Permissions. As the first step, we extracted the sensitive permissions used by malware apps. Thus, we defined a sensitive permission list, which is based on the dangerous permissions and groups given by the Android developer document [19], as shown in Table 4 (the first two columns). Then, we used Androguard [7] to get the permissions declared by each malware app. Note that, in Android 6.0 or later versions, permissions not only need to be registered in the manifest files but also need to be dynamically requested while the app is running [6].

Table 4: Sensitive Permissions Used By PMLs.

Group	Permission	Num	Rate
CONTACTS	WRITE_CONTACTS	16	0.71%
	GET_ACCOUNTS	186	8.20%
	READ_CONTACTS	126	5.56%
PHONE	READ_CALL_LOG	8	0.35%
	READ_PHONE_STATE	1,719	75.83%
	CALL_PHONE	376	16.59%
	WRITE_CALL_LOG	1	0.04%
	USE_SIP	0	0.00%
	PROCESS_OUTGOING_CALLS	2	0.09%
	ADD_VOICEMAIL	0	0.00%
CALENDAR	READ_CALENDAR	111	4.90%
	WRITE_CALENDAR	113	4.98%
CAMERA	CAMERA	60	2.65%
SENSORS	BODY_SENSORS	0	0.00%
LOCATION	ACCESS_FINE_LOCATION	1,705	75.21%
	ACCESS_COARSE_LOCATION	1,655	73.00%
STORAGE	READ_EXTERNAL_STORAGE	19	0.84%
	WRITE_EXTERNAL_STORAGE	1,395	61.54%
MICROPHONE	RECORD_AUDIO	73	3.22%
SMS	READ_SMS	53	2.34%
	RECEIVE_WAP_PUSH	1	0.04%
	RECEIVE_MMS	1	0.04%
	RECEIVE_SMS	24	1.06%
	SEND_SMS	144	6.35%
	READ_CELL_BROADCASTS	0	0.00%

Results. We successfully obtained the permissions of 206,722 malware samples and failed in 415 samples due to the code packing protection. On the aspect of usage percentage, four permissions are used by over 60% samples. Specifically, `READ_PHONE_STATE` (96.97%) and `WRITE_EXTERNAL_STORAGE` (94.72%) are the most common permissions. The former one is usually used to obtain device identity information (e.g., IMEI) for tracking the mobile user. The latter one is used to store data on the disk. In addition, location information related permissions are also popular, say 61.1% for `ACCESS_FINE_LOCATION` and 69.4% for `ACCESS_COARSE_LOCATION`.

5.3.2 Permissions Used by PMLs. Further, we studied the sensitive permissions used by PMLs. Based on the API-permission mapping list of PScout [14], we obtained the used permissions through searching sensitive API invocations.

Results. In total, 2,267 PMLs used sensitive permissions. The categories and quantities of permissions are listed in Table 4. According to the statistics, `READ_PHONE_STATE` is the most frequently used permission, say 75.83% (1,719/2,267). Recall that in the malware samples, the percentage is 96.97%. Similarly, two

locations related permissions – ACCESS_FINE_LOCATION and ACCESS_COARSE_LOCATION are also popular (75.21% and 73% respectively).

Our assessment:
- The permissions used by PMLs could correlate to the potentially harmful behaviors contained by PMLs.
- Almost every app corresponding to PMLs has declared the permission READ_PHONE_STATE and WRITE_EXTERNAL_STORAGE.

5.4 Developer Connections with Apps

In this sub-section, we analyzed the developer connections between PMLs and their corresponding apps. Since it is quite challenging to identify the developers of PMLs directly, we explored the connections based on the app certificates, which are signed by the developers' private keys. Following this approach, we first collected the certificates of all malware apps associated with those PMLs. Then we built a bipartite graph between the PMLs and certificates signatures to explore the hidden connections.

5.4.1 App Certificates. First, we extracted the app certificates from the malware samples containing PMLs. To each APK file, we uncompressed it and extracted the file with the suffix .DSA or .RSA under the original/META-INF folder. This file is the certificate that contains the signature information. Then we used Keytool [9] to parse these certificates and obtained the signatures.

In general, if an app is signed by a CA (certificate authority), we can find the developer information from its app certificate. In the certificate, the *owner* field refers to the developer's information, and the *issuer* field refers to the CA that signed this certificate. However, most app certificates are self-signed for convenience, and the contained information may not always be real. Despite all this, based on a large-scale of certificates, we still can obtain some valuable information from their signatures. The limitation will be discussed in Section 6.4.

Results. We collected 207,102 certificates in total and failed to extract certificates in 35 malware samples due to the incomplete APK files. Also, among these certificates, 530 certificates cannot be parsed by Keytool. Finally, we obtained signature information from 206,572 certificates, of which only 726 certificates (0.35%) were issued by the CA, and 205,846 certificates (99.65%) are self-signed. In particular, 516 certificates were issued by "CMCA app signing CA", 209 certificates were issued by "CMCA code signing CA", and one certificate was issued by "WoSign OV Code Signing". However, we are unable to obtain information about the developers of these apps directly from these certificates because the *owner* field is represented by special codes and cannot be identified.

5.4.2 Bipartite Graph. We built a bipartite graph between the PMLs and the self-signed certificates of their corresponding malware apps, which could present the connections between the libraries and the apps. According to our observations, in most self-signed apps, the owner and issuer field of each signature are the same. Thus we use the owner as of the unique identifier for each certificate. If two certificates have the *owner* with the same arguments (including EmailAddress, CN (Common Name), OU (Organization Unit), O (Organization), L (Locality Name), ST (State Name), C (Country)), they will be treated as the same certificate.

Note that some certificates cannot provide useful information and should be eliminated from the bipartite graph. For example, there are 74 libraries of which certificates are all signed by "Android Debug". We also found 329 libraries that connected to empty certificates (all seven parameters in the owner field are null) and 24 libraries of which corresponding certificates are either debug-signed or empty. In some certificates, the CN field is "test" or "unknown". Since these certificates are meaningless for our analysis, we eliminated them from the bipartite graph.

Results. We obtained a bipartite graph between 17,127 certificates and 4,526 PMLs. We also found some interesting cases. For example, in the certificate of android.dzs.cql.hcham (an app about novels), its CN field is baidusoso, and it appears in 331 apps. However, after our investigation, we found that this name does not belong to the Baidu company, which indicates that the developers of these apps took advantage of Baidu to mislead app developers and users.

The certificate having the most corresponding libraries is "apkide" (shown in the CN field), which has 370 corresponding PMLs. However, "apkide" is an APK modification tool used by plenty of Chinese developers. They use this tool to decompile, modify, and repackage APKs. These 370 PMLs are possibly injected into those APKs by modifying and repackaging these APKs using this tool.

If a library is associated with multiple different certificates, it is difficult to explore the connection between them. Thus we focused on the libraries that only have one kind of the corresponding certificate on the bipartite graph. We found 1,211 PMLs that only associated with one corresponding certificate. The top ten libraries with only one certificate are shown in Table 5. CN refers to the developer, and O refers to the organization. The library org.dp.pp, which ranks top in Table 5 and its corresponding signature is "eastegde". The certificate is also connected to 3 other PMLs: com.revmob.ads, com.dianru.adsdk, and com.fancypush.pushnotifications. These 4 PMLs have 4,145 corresponding apps, and these apps are about novels and readings. Since these PMLs are only associated with "eastedge", they may share the same author, which earned revenue by luring users to click on the ad links while reading novels.

Our assessment:
- Nearly all PMLs are used by self-signed apps.
- Some popular PMLs may be created by the same author.

Table 5: Top 10 libs with a single corresponding certificate.

No	Lib Name	CN	O	APKs
1	org.dp.pp	eastedge	null	4,060
2	com.fw.tzfive	luo	baixiu	1,059
3	com.dlf.myp	aaaaaa	null	1,057
4	com.jeef.wapsConnection	luo	baixiu	871
5	com.nyc.a	Andrew Vasiliu	Qbiki Networks	728
6	net.mz.callflakessdk	null	neoline	412
7	cn.appmedia.ad	Andrew Vasiliu	Qbiki Networks	367
8	com.revmob	zhuyg	zhuyg	328
9	cn.appmedia.ad	zhuyg	zhuyg	321
10	com.fgbljllzq.rvmzmugcz208126	Android	Android	287

6 DISCUSSIONS AND LIMITATIONS

6.1 PML Identification

VirusTotal may omit some PMLs. For example, we found an ad library called `com.mobclix.android`, which invokes the Linux command `su` to try to get the root privilege. We believe that this behavior should not exist in a benign library, but VirusTotal failed to report this library as a malware. Since it is costly to manually find out those omitted libraries, we only used the PMLs in the report.

6.2 Obfuscation

As described in Section 3.3 and 3.4, due to the special relative paths, LibExtractor is resilient to package name obfuscations, like Proguard [10], the Android official obfuscation tool. Besides, since LibExtractor sorts relative paths within each dependency field in order, obfuscations tools that modify the order in which the methods appear in the code will not affect LibExtractor. However, some obfuscation tools perform code encryption, which will lead to the failure of dependency extraction and impede LibExtractor. This situation will be described in Section 6.3. On the other hand, we cannot handle code shrinking and obfuscations that can change the package structure, which will change the relative path between classes. It is hard to distinguish the libraries that are obfuscated in this way, thus we treated them as library mutations in the analysis.

6.3 Clustering Candidates

In Section 3.4, we found that some candidates in the same group may have completely different package names, like

```
09e422c...97b636e411b78f9be - com/zcp/xgsdk
09e422c...97b636e411b78f9be - com/umeng/customview
09e422c...97b636e411b78f9be - com/umeng/qqsdk
```

The reason is the failure of parsing smali codes. In Section 3.1, baksmali [8] failed to parse the Smali code of some apps but only obtained the class and package names. Each of these packages was divided as a candidate in Section 3.2 since they have no dependency. These packages generated the same hash value in Section 3.3, thus different candidates shared the same feature value and were grouped. Since these candidates would reduce the accuracy of our results, we removed them while clustering candidates.

6.4 Self-Signed Certificates

In Section 5.4.1, we got an app named `com.moon.t-ingchejiemi2`. Its certificate shows that the APK comes from "California, US", but it should be a Chinese app according to its package name. This example shows that the information in self-signed certificates may not be real. However, since we have large-scale corresponding certificates, we still can obtain valuable information. In addition, the result also shows that our approach is promising.

7 RELATED WORK

Third-Party Library Identification. Previous third-party library identification tools have three mainstream approaches. (1)The first is to create a whitelist of library names for known libraries. Grace et al. [24] and Book et al. [16] built a list of well-known ad libraries and identified the ad libraries in the app by matching package names. However, the package name obfuscation will significantly reduce the accuracy of this approach. (2) The second approach is to generate feature values for the known libraries and built a database to identify libraries by feature matching and similarity comparison. LibScout [15] used a high-level package organization to generate the feature value. Wukong [39] and LibRadar [34] generated the feature value of each library based on system APIs used in the libraries. LIBPECKER [42] based on class dependency and assigned different weights to each class and matched the third-party libraries by calculating the similarity between the classes. However, it is costly to identify unknown libraries using this approach, because they need to update the database frequently and perform similarity comparison. (3) The third approach is to extract library candidates from large-scale apps and cluster the candidates based on their feature values. [29, 32, 33]. For example, LibD [33] used inheritance, inclusion, and call relations to construct candidates and generates the features by hashing the opcodes within basic blocks. Our tool uses the third approach and improves efficiency. We use class dependency for both candidate construction and feature generation and generate each class's feature value based on a novel algorithm.

Malware Detection. Due to the pervasiveness of Android devices and apps, researchers and practitioners have proposed various approaches to detect and dissect Android malware [12, 20, 26, 41, 43]. Androguard [7] has been proposed for conducting a security analysis of Android files, which has also used by plenty of malware detection approaches [3, 35, 36]. TaintDroid [21] has been proposed to track privacy leaks in smartphones at runtime. Similarly, FlowDroid [13] and IccTA [27] have been proposed to detect privacy leaks in Android apps statically. Android malware might be repackaged (or piggybacked) from other apps [31]. Hence, previous research proposed various approaches to detect and dissect repackaged Android apps [28]. Besides, Li et al. [30] have developed a prototype tool called SimiDroid for detecting repackaged apps.

8 CONCLUSION

In this paper, we give an in-depth analysis of potentially malicious Android third-party libraries. Firstly, we propose LibExtractor that can quickly extract libraries from large-scale apps and identify potentially malicious libraries based on the report of VirusTotal and their potentially harmful behaviors. We obtained 4,957 potentially malicious libraries for analysis and 19,938 possible benign libraries for behavior contrast. We performed a comprehensive analysis of the PMLs, including library repackaging, exposed components, permissions and developer connections with corresponding apps. The result shows the typical characteristic of PMLs with some interesting cases.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their insightful comments. This work was partially supported by National Natural Science Foundation of China (NSFC) under Grant No. 61902148, No. 91546203, Key Science Technology Project of Shandong Province No. 2015GGX101-046, Major Scientific and Technological Innovation Projects of Shandong Province, China No. 2017CXGC0704, No. 2018CXGC0708, No. 2019JZZY010132 and Qilu Young Scholar Program of Shandong University.

REFERENCES

[1] 2015. *Apple's App Store infected with XcodeGhost malware in China.* Retrieved March 1, 2020 from https://www.bbc.com/news/technology-34311203

[2] 2016. *Androzoo.* Retrieved March 1, 2020 from https://androzoo.uni.lu/

[3] 2019. *AndroPyTool.* Retrieved March 1, 2020 from https://github.com/alexMyG/AndroPyTool

[4] 2019. *Androwarn.* Retrieved March 1, 2020 from https://github.com/maaaaz/androwarn

[5] 2019. *Argus-SAF.* Retrieved March 1, 2020 from https://github.com/arguslab/Argus-SAF

[6] 2019. *Permission Requesting in Android system.* Retrieved March 1, 2020 from https://developer.android.com/training/permissions/requesting

[7] 2020. *Androguard.* Retrieved March 1, 2020 from https://github.com/androguard/androguard

[8] 2020. *JesusFreke/smali/bacsmali.* Retrieved March 1, 2020 from https://github.com/JesusFreke/smali/tree/master/baksmali

[9] 2020. *Keytool.* Retrieved March 1, 2020 from https://docs.oracle.com/javase/8/docs/technotes/tools/windows/keytool.html

[10] 2020. *Proguard.* Retrieved March 1, 2020 from https://www.guardsquare.com/en/products/proguard

[11] 2020. *VirusTotal.* Retrieved March 1, 2020 from https://www.virustotal.com/gui/

[12] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. 2014. Drebin: Effective and explainable detection of android malware in your pocket.. In *Ndss*, Vol. 14. 23–26.

[13] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Acm Sigplan Notices*, Vol. 49. ACM, 259–269.

[14] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. 2012. PScout: analyzing the Android permission specification. In *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012.*

[15] Michael Backes, Sven Bugiel, and Erik Derr. 2016. Reliable Third-Party Library Detection in Android and Its Security Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (Vienna, Austria) *(CCS '16)*. 356–367.

[16] Theodore Book, Adam Pridgen, and Dan S. Wallach. 2013. Longitudinal Analysis of Android Ad Library Permissions. *CoRR* abs/1303.0857 (2013). arXiv:1303.0857 http://arxiv.org/abs/1303.0857

[17] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. 175–186.

[18] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou. 2016. Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios. In *Security and Privacy (SP), 2016 IEEE Symposium on*. IEEE, 357–376.

[19] Dangerous Permissions and Groups 2019. *Dangerous Permissions and Groups.* Retrieved March 1, 2020 from https://developer.android.com/guide/topics/security/permissions.html#normal-dangerous

[20] Feng Dong, Haoyu Wang, Li Li, Yao Guo, Tegawendé F Bissyandé, Tianming Liu, Guoai Xu, and Jacques Klein. 2018. FraudDroid: Automated Ad Fraud Detection for Android Apps. In *The 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2018)*.

[21] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)* 32, 2 (2014), 5.

[22] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. 2011. Permission Re-Delegation: Attacks and Defenses.. In *USENIX Security Symposium*, Vol. 30. 88.

[23] Julien Gamba, Mohammed Rashed, Abbas Razaghpanah, Juan Tapiador, and Narseo Vallina-Rodriguez. 2019. An Analysis of Pre-installed Android Software. *CoRR* abs/1905.02713 (2019). arXiv:1905.02713 http://arxiv.org/abs/1905.02713

[24] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. 2012. Unsafe Exposure Analysis of Mobile In-app Advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks* (Tucson, Arizona, USA) *(WISEC '12)*. 101–112.

[25] W. Hu, J. Tao, X. Ma, W. Zhou, S. Zhao, and T. Han. 2014. MIGDroid: Detecting APP-Repackaging Android malware via method invocation graph. In *2014 23rd International Conference on Computer Communication and Networks (ICCCN)*. 1–7. https://doi.org/10.1109/ICCCN.2014.6911805

[26] Yangyu Hu, Haoyu Wang, Yajin Zhou, Yao Guo, Li Li, Bingxuan Luo, and Fangren Xu. 2019. Dating with Scambots: Understanding the Ecosystem of Fraudulent Dating Applications. *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2019).

[27] Li Li, Alexandre Bartel, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. 2015. IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*.

[28] Li Li, Tegawendé F Bissyandé, and Jacques Klein. 2019. Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark. *IEEE Transactions on Software Engineering (TSE)* (2019).

[29] Li Li, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. 2016. An investigation into the use of common libraries in android apps. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 403–414.

[30] Li Li, Tegawendé F Bissyandé, Haoyu Wang, and Jacques Klein. 2019. On Identifying and Explaining Similarities in Android Apps. *Journal of Computer Science and Technology (JCST)* (2019).

[31] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Yves Le Traon, David Lo, and Lorenzo Cavallaro. 2017. Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. *IEEE Transactions on Information Forensics & Security (TIFS)* (2017).

[32] Li Li, Timothée Riom, Tegawendé F Bissyandé, Haoyu Wang, Jacques Klein, and Yves Le Traon. 2019. Revisiting the Impact of Common Libraries for Android-related Investigations. *Journal of Systems and Software (JSS)* (2019).

[33] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo. 2017. LibD: Scalable and Precise Third-Party Library Detection in Android Markets. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 335–346.

[34] Ziang Ma, Haoyu Wang, Yao Guo, and Xiangqun Chen. 2016. LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps. In *Proceedings of the 38th International Conference on Software Engineering Companion* (Austin, Texas) *(ICSE '16)*. 653–656.

[35] A. Martín, R. Lara-Cabrera, and D. Camacho. [n.d.]. *A new tool for static and dynamic Android malware analysis.* 509–516. https://doi.org/10.1142/9789813273238_0066

[36] Alejandro Martín García, Raul Lara-Cabrera, and David Camacho. 2018. Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset. *Information Fusion* 52 (12 2018). https://doi.org/10.1016/j.inffus.2018.12.006

[37] Xiaorui Pan, Xueqiang Wang, Yue Duan, XiaoFeng Wang, and Heng Yin. 2017. Dark Hazard: Learning-based, Large-Scale Discovery of Hidden Sensitive Operations in Android Apps.. In *NDSS*.

[38] Statcounter 2020. *Mobile Operating System Market Share Worldwide.* Retrieved March 1, 2020 from http://gs.statcounter.com/os-market-share/mobile/worldwide

[39] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. 2015. WuKong: A Scalable and Accurate Two-phase Approach to Android App Clone Detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. 71–82.

[40] Daoyuan Wu, Debin Gao, Rocky KC Chang, En He, Eric KT Cheng, and Robert H Deng. 2019. Understanding open ports in Android applications: Discovery, diagnosis, and security assessment. In *Proceedings of Network and Distributed System Security Symposium 26th (NDSS 2019)*. San Diego, CA. https://doi.org/10.14722/ndss.2019.23171

[41] Xinli Yang, David Lo, Li Li, Xin Xia, Tegawendé F Bissyandé, and Jacques Klein. 2017. Characterizing malicious Android apps by mining topic-specific data flow signatures. *Information and Software Technology* (2017).

[42] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen. 2018. Detecting third-party libraries in Android applications with high precision and recall. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Vol. 00. 141–152.

[43] Yajin Zhou and Xuxian Jiang. 2012. Dissecting android malware: Characterization and evolution. In *2012 IEEE symposium on security and privacy*. IEEE, 95–109.