TraceDroid: A Robust Network Traffic Analysis Framework for Privacy Leakage in Android Apps

Huajun Cui[1,2], Guozhu Meng[1,2], Yan Zhang[1,2(✉)], Weiping Wang[1,2], Dali Zhu[1,2], Ting Su[3], Xiaodong Zhang[1,2], and Yuejun Li[1,2]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
zhangyan80@iie.ac.cn
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3] Software Engineering Institute, East China Normal University, Shanghai, China

Abstract. Network traffic analysis is an appealing approach for the security auditing of mobile apps. Prior research employs various techniques (e.g., Man-in-the-Middle, TCPDUMP) to capture network traffic from apps and further recognize security/privacy risks inside. However, these techniques suffer from limitations such as traffic mixing, proxy evasion, and SSL pinning. Possible solutions are to modify and customize the Android system. However, existing studies are mainly based on Android OS 6/7. Contemporary apps generally cannot work properly on these archaic Android OS, which has become a stumbling block for further traffic analysis research. To address the above problems, we propose a new network traffic analysis framework-TraceDroid. We first leverage the dynamic hooking technique to hook the critical functions for sending network requests, and then save the request data along with code execution traces. Besides, TraceDroid proposes an unsupervised way to identify third-party libraries (TPLs) inside apps for facilitating the liability analysis between apps and TPLs. Utilizing TraceDroid, we conduct a large-scale experiment on 9,771 real-world apps to make an empirical study of the status quo of privacy leakage. Our findings show that TPLs account for 44.45% of privacy leakage in contemporary apps, and files transmitted from user devices contain much more detailed privacy data than network requests. We bring to light the over-data harvest and cross-library data harvest issues in apps. Furthermore, we unveil the relationship between TPLs and their visiting domains that previous research has never discussed.

Keywords: Network traffic · Privacy · Android · Third-party library

1 Introduction

As the most widely used mobile platform, the Android operating system brings great convenience to our society but also introduces lots of security problems like

C. Su et al. (Eds.): SciSec 2022, LNCS 13580, pp. 541–556, 2022.
https://doi.org/10.1007/978-3-031-17551-0_35

privacy leakage [12,38], advertisement fraud [8,9,16], and malware [13,34], etc. Network traffic analysis has been demonstrated as one of the most prominent methods to mitigate security concerns by plenty of researches. Generally, existing traffic analysis studies [6,17,23–25,32,40] usually employ TCPDUMP or Man-in-The-Middle (MITM) technique to capture network traffic. However, TCPDUMP cannot handle the traffic encryption problem, thus MITM tools have become the most used method to capture and inspect encrypted traffic. The MITM technique runs on a proxy mechanism, which means users need to install the custom certificate of MITM tools, trust the certificate, and configure a proxy (MITM server) on their devices. By doing so, all traffic of the device should go through the proxy server before reaching the target servers. With custom certificates, the proxy server can decrypt the network traffic by acting as a middleman in the communication path. However, this mechanism fails to handle the traffic mixing problem–background traffic (Android OS traffic, background service traffic), app traffic, and TPL traffic are mixed together, which prevents the fine-grained traffic analysis. In addition, some other limitations caused by MITM tools have not been well addressed (Sect. 2.1).

To mitigate the traffic mixing problem, prior studies [19,30,37] use a whitelist of "User-Agent" or "Host" in the HTTP request header to identify ad-library traffic. However, this method requires practitioners to maintain a list containing known domains and TPLs, and the list needs to be updated frequently. Obviously, as new TPLs keep emerging, this approach may result in low accuracy and the out-of-date lists cannot satisfy contemporary apps [39].

To distinguish the network traffic, an alternative mechanism is to modify and customize the Android operating system. However, this way is heavy-weight that necessitates a recompilation of the Android system every time for a code update. What's more, current studies in this area are mainly on Android 6/7 [6,23,24], such archaic Android system versions are not appropriate to run contemporary apps. In addition, on newer Android systems, new problems will appear (Sect. 3.2), which are not addressed by existing works.

To this end, we propose a new framework, TraceDroid, for network traffic analysis. The proposed TraceDroid can simultaneously address the above problems without any modification of the Android system. Specifically, we leverage the dynamic hooking technique to add additional code in the functions which are responsible for performing HTTP(S) requests, and then save the unencrypted data and the corresponding code execution traces. To present a fine-grained liability study of privacy leakage between host apps and TPLs, we propose an unsupervised method to identify TPLs by correlating the requests and code execution traces, and use this method to distinguish the traffic between host apps and TPLs. With the help of TraceDroid and TPL identification, we conduct a large-scale experiment on 9771 real-world apps and make a comprehensive analysis on the collected data to identify the status quo of privacy leakage in modern apps. Our new findings are as follows: 1) 44.45% of privacy leakage requests are initiated by TPLs, which indicates that TPLs have become a non-negligible channel for privacy leakage. 2) Device ID (e.g., IMEI, IMSI, SN) is the most appealing privacy data. 3) Over-data harvest of privacy information widely exists in contemporary apps and TPLs. The user's private data is sent to

multiple back-end servers without noticing them. 4) Files transmitted from the user device tend to contain much more detailed privacy information than that in HTTP(S) requests,which has long been ignored by the research communities. 5) The relationship between TPLs and their visiting domains is a many-to-many correlation, and the domains can be classified into self-owned, authorization, and host app domains (Sect. 4.5).

Contributions. Our key contributions are summarized as follows:

- *We propose a new framework for Android network traffic analysis.* To our knowledge, TraceDroid is the first work that can solve traffic mixing, proxy evasion, traffic encryption, and SSL pinning simultaneously without any OS modification. Besides, TraceDroid can address the new challenges of running contemporary apps on modern Android systems (Sect. 3). Furthermore, TraceDroid is light-weight, making it easy to expand or modify. To foster further research, we released TraceDroid to the research community at https://github.com/TraceDroid/TraceDroid-SciSec2022.
- *We design a new method to identify TPLs in apps.* An unsupervised method is proposed to identify TPLs by correlating HTTP(S) requests and the code execution traces. Compared to prior works, our method does not require any prior knowledge of TPLs or whitelists. We identified more than 300 TPLs in our app corpus and released the TPLs and their visiting domains on our Github repository.
- *A large-scale analysis of privacy leakage in contemporary apps is conducted.* We perform an empirical study on 9,771 apps, to analyze the transmission manners of leaked privacy data and the liable parties (Sect. 4.2 and 4.3), the over-data harvest and cross-library data harvest (Sect. 4.4), and the relationship between TPLs and their visiting domains (Sect. 4.5).

2 Background

2.1 MITM-based Traffic Capturing

As mentioned in Sect. 1, various MITM tools [7,10,18,20] use the proxy server and the custom certificate to capture and inspect HTTP(S) traffic. Unfortunately, we found some limitations of this method by investigating our app corpus and concluded them as follows:

Traffic Mixing. Mobile systems and apps are running a number of daemon processes, such as Google Framework Service and Push Service. Their traffic is mixed with the app traffic as background noise and is captured by the MITM server. As a result, it is non-trivial to distinguish the network traffic from the background traffic, app traffic, and TPL traffic.

Anti-debugging. Apps can detect whether a proxy is configured and refuse to communicate or change network behaviors. It is largely attributed to preventing apps from being debugged. For instance, the app "*com.outfit7.mytalkingtom.qihoo*" will not show any ads if a proxy is present.

Proxy Evasion. Apps are allowed to establish a direct connection to the target server even when a proxy server is configured on Android. For instance, the public method "*openConnection(Proxy proxy)*" in *java.net.URLConnection* can use a parameter "*Proxy.NO_PROXY*" to ignore the proxy server and create a direct socket connection that bypasses the proxy server.

SSL Pinning. It is the ability to trust specific certificates preinstalled in an app. With SSL pinning, an app can validate the server's certificate to ensure the uniqueness and security of communication between the app and the server. As a result, the custom certificate generated by the MITM tools will not work.

2.2 Hook-Based Traffic Capturing

Hooking (or Instrumentation) refers to injecting additional code into a program to collect runtime information. Once a code is injected into the target process, it has full access to the process memory and can modify its components. Developers can use this ability to alter the target process memory components, allowing them to replace or modify the API functions.

To capture network traffic, we can hook the corresponding APIs and inject additional codes into them to get the request data. Android app developers usually use various networking libraries [1,3,11,21,22,26,27,31] to perform network requests. Take Okhttp [21] as an example, it is a widely used networking library in Android development. The developer usually creates a "*httpClient*" object, builds a "*request*", and uses its method "*request.set()*" to set the key-value parameters like "content-type:application/json" in the request header. The function "*perform(request)*" is finally called for sending out this HTTP request. In this case, we can hook the function "*perform(request)*" and save its parameter "*request*" to get the request data. We will show how to get the unencrypted request data in Sect. 3.1.

Compared with MITM tools, hook-based traffic capturing has the following advantages: 1) Hooking is only for the particular APIs in a certain target app (specific app process ID in the Android operating system), so the captured traffic does not contain traffic other than the app. 2) Hooking does not change the logic integrity of the original program, and it is transparent to apps, so it is not affected by anti-debugging and proxy evasion. Moreover, it does not need a custom certificate, so the pinning cannot stop us from getting request data.

3 Approach

As shown in Fig. 1, TraceDroid proceeds in four phases–*network hooking*, to instrument Android devices for traffic capturing; *traffic triggering*, to make apps produce more network traffic and parse the traffic to restore network requests, files, and call stacks; *TPL identification*, to identify TPLs in apps based on the captured requests and call stacks, which can benefit our liability analysis in Sect. 4.2; and *privacy leakage analysis*, to conduct a large-scale study on our apps corpus to identify the status quo of privacy leakage in modern apps.

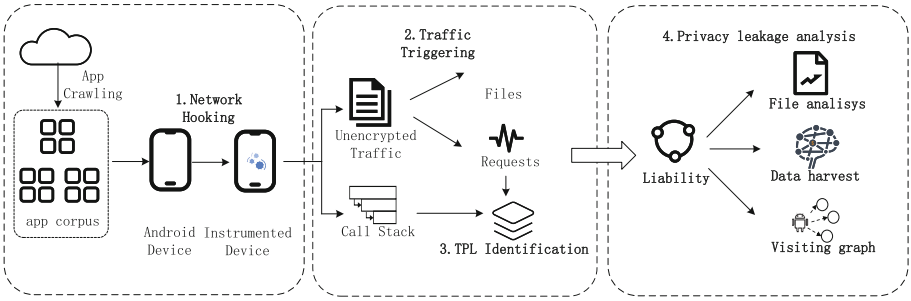

Fig. 1. System overview of TraceDroid

3.1 Network Hooking

As mentioned in Sect. 2.2, we use the hook-based traffic capturing method to overcome the limitations of MITM tools. Our idea is to hook all functions responsible for performing HTTP(S) requests. Nevertheless, there exist dozens of networking libraries [1,3,11,21,22,26,27,31]. It will be labor-intensive and time-consuming to hook all these libraries because this needs to do case by case study and instrument dozens of APIs. To this end, we turn to the underlying APIs that are depended by the networking libraries.

We manually analyzed the networking libraries and found that they rely on the OpenSSL library to perform SSL connections. OpenSSL (or named BoringSSL after Android 6) is the default SSL/TLS library in the Android system [5]. It contains two parts–libssl and libcrypto. Among them, libssl is the implementation of the SSL protocol, and the Android system uses its built-in functions (i.e., "*SSL_Read*" and "*SSL_Write*") to handle HTTPS requests. "*SSL_Read*" is used to read data from an established SSL session and put it into a buffer. "*SSL_Write*", on the contrary, writes data to the buffer and sends it to the remote server. Note that the functions in libcrypto will be invoked to decrypt and encrypt the data before "*SSL_Read*" or after "*SSL_Write*" is invoked, so the data read and written here is plain text (unencrypted data).

To illustrate how our network hooking method works, we take "*SSL_Write*" as an example. Its function prototype is "*int SSL_Write(SSL *ssl, const void *buf, int num)*", the API parameter "*ssl*" is the specified SSL connection, "*buf*" is the buffer that this function writes data into, "*num*" is the data (measured in number of bytes) will be written into "*buf*". As mentioned above, the libcrypto will be automatically invoked to encrypt the data after this function, and finally, the data is sent to the remote server. So we hook the function "*SSL_Write*", inject additional code into it, and save "*num*" bytes data from the "*buf*"–so we get the unencrypted request data (that is, we get the data before it is encrypted).

Similarly, we hook two APIs –"java.net.SocketOutputStream.socketWrite0" and "java.net.SocketInputStream.socketRead0" as they are the default HTTP APIs in the Android system. Table 1 shows the hooking functions in TraceDroid.

Evaluation. Theoretically, developers can customize their own TLS library instead of using the default one provided by the Android system, and our network hooking method will not work in this case. But considering the developing cost and security concerns, we believe few apps do so. To evaluate the effect of our method, we randomly investigated 80 apps, and only one app (package name: "*com.ss.android.ugc.aweme*", version 17.3) was found to have a custom TLS library.

Table 1. Hooking functions for network capturing

Traffic hooking	HTTPS	libssl SSL_Write libssl SSL_Read
	HTTP	java.net.SocketOutputStream.socketWrite0 java.net.SocketInputStream.socketRead0
Call stack hooking	HTTPS	ConscryptFileDescriptorSocket$SSLOutputStream ConscryptFileDescriptorSocket$SSLInputStream
	HTTP	java.net.SocketOutputStream.socketWrite0 java.net.SocketInputStream.socketRead0

3.2 Traffic Triggering

Obstacles Before App Execution. To capture app traffic, we install and execute apps to produce network traffic. However, with the evolution of the Android system, we found new obstacles before we can run apps in modern Android systems. First, apps may ask for permissions before users enter the app. Google introduced a runtime permission authorization mechanism [2] after Android 6.0 (API 23), and dangerous permissions like "android.permission.CALL_PHONE" have to be granted during runtime. As shown in Fig. 2 a), a dialog asking for permissions likely appears and blocks app execution unless the permission request is confirmed by the user. Second, many apps present various pop-up prompts or splash screens when launched. As shown in Fig. 2 b), a prompt dialog may present a user agreement or privacy policy that defines the responsibilities of each party, and the prompt dialog also needs confirmation from the user. As shown in Fig. 2 c), a splash screen is usually an introduction for apps while it is loading, and users have to swipe left/right on these screens to complete the procedure.

To mitigate these obstacles, we design and implement an automatic tool *obsCleaner* to satisfy them. The idea is to recognize the obstacle-related Android widgets on a screen and make certain UI operations to accomplish them. Specifically, for permission requests and pop-up prompts, we randomly tested 200 apps and distilled a keywords list related to these obstacles(e.g., "ok", "agree", "continue", and "start"). According to the keywords list, we locate the coordinate of the obstacles on the screen and imitate human operations to generate related

events to satisfy them. We consider this work a one-time effort because these words are not updated frequently, and the list can be extended easily. For splash screens, we swipe the screen for a certain times K, and we set K to 5 because all the 200 apps can be satisfied by 5 times in our manual test. We have released *obsCleaner* on our Github repository to foster future research.

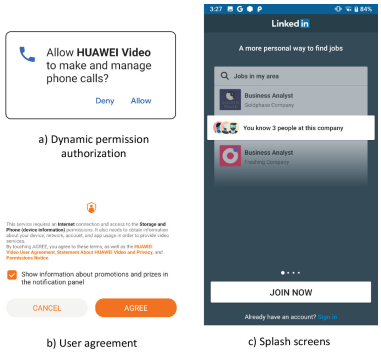

Fig. 2. Obstacles before app execution

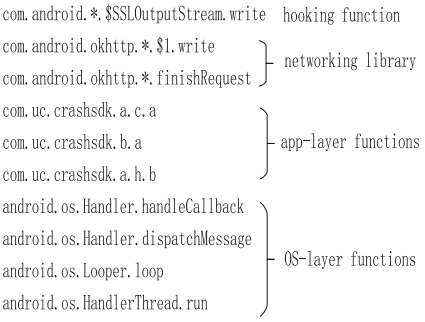

Fig. 3. An example of call stacks

Triggering More Traffic. An important task in network traffic analysis is to generate as much traffic as possible. Previous works usually use the Android UI fuzzing tool Monkey to trigger traffic from apps. However, Monkey's random events are not efficient in triggering traffic. In this paper, we use the tool in our previous work–*AutoClick* [4], to do this. *AutoClick* is a lightweight and efficient tool to trigger network traffic. It is based on the "*click first*" principle, which automatically clicks all "*clickable*" layout elements on an Android activity. Our previous work demonstrated that it outperforms about 85% in generating distinct traffic during a limited time than Monkey.

3.3 TPL Identification

TPL identification refers to detecting the presence of TPLs in apps, and it is an important task to facilitate fine-grained traffic analysis. We manually employed existing TPL identification tools [14,15,28,35] to 20 randomly selected apps to see their accuracy. Unfortunately, the result shows that although they have a good performance on known TPLs (TPLs that they have trained), they lead to very low accuracy for the newly-emerged TPLs like "*com.bytedance.**", and this conclusion is in line with the prior study [39]. Nevertheless, the newly-emerged TPLs usually produce large traffic as a majority of apps integrate them, making them an indispensable part of our research.

The lucky thing is that previous work [29] shows that network traffic generated from the same TPL has similar network behaviors across different apps. That is to say, their HTTP(S) requests are similar in structure.

Algorithm 1: TPL identification

Input: $G : request \rightarrow call\_stack$: a visited graph mapping HTTP(S) requests to the corresponding call stacks
Output: *libs*: TPLs in apps

1 $libs \leftarrow \emptyset$;
2 Identify *domains* visited by all requests in G;
3 **for** $domain \in domains$ **do**
4 Obtain *reqs* that visit *domain*, and $reqs \in G\langle request \rangle$;
5 $clusters \leftarrow performClustering(reqs)$;
6 **for** $cluster \in clusters$ **do**
7 **for** $req \in cluster$ **do**
8 $cstack \leftarrow G(req)$;
9 $cstack' \leftarrow filter(cstack)$ by removing hooking and networking libraries;
10 **for** $call \in cstack'$ **do**
11 $lib \leftarrow truncate(call)$;
12 $libs.append(lib)$;
13 $libs.deduplicate()$ by removing redundant libs;
14 **return** libs;

Hence, we propose an unsupervised manner of associating network requests with call stacks to identify TPLs. Algorithm 1 shows our TPL identification process. First, we obtain the domains visited by more than M apps, and M is empirically set to 10 (line 2). For each domain, we extract signatures from all the corresponding requests as *reqs* (line 4), and a *req* is $\langle protocol, method, URL, host, path, key\_list \rangle$. A clustering analysis is performed where the requests with the same signature are grouped together (line 5). Then, we get the corresponding call stacks for each request to obtain the candidate TPL (line 6). A call stack is a list of functions with a bottom-to-up code execution trace shown in Fig. 3. We filter the stack by deleting our hooking functions in Sect. 3.1 and networking libraries such as Okhttp3 (line 9) mentioned in Sect. 3.1 to get the candidate TPL. Next, we truncate the top three package names of the candidate TPL and remove redundant ones to get final TPLs (line10-13).

Here we take a call stack as an example in Fig. 3. For ease of presentation, we have omitted irrelevant parts. The code execution trace contains four parts: OS-layer functions, app-layer functions, networking libraries, and hooking functions. TraceDroid first deletes the hooking functions, networking library–"*com.android.okhttp.**", and the OS-layer functions (e.g., "*android.os.**"). In this way, we get the remaining package names with the prefix of "*com.uc.crashsdk.**". We truncate the top three package names and do de-duplication of these package names to obtain the TPL –"*com.uc.crashsdk*".

Parameter Setting. We conducted a statistic showing how many apps visit each domain. It exhibits a long-tailed distribution with 95.8% of domains visited

by less than 10 apps, so we set M to 10 to obtain the candidate domains for further analysis. We have manually checked 53 TPLs identified by our method, and all of them are real TPLs (results are also released on our Github repository).

4 Evaluation and Measurement

Implementation and Experiment Setup. We implement TraceDroid with about 6K lines of Python and JavaScript code based on UIAutomator2 and Frida. For obstacle elimination, we use UIAutomator2 API "*XPath()*" to locate the coordinates of obstacles according to the keywords list; for network hooking, we use the Frida API "Interceptor.attach()" to hook the functions. For each app, TraceDroid installs it, triggers it for 10 min, and collects network traffic during runtime. After the test time, TraceDroid terminates the app and uninstalls it. TraceDroid sends the collected data to our server, and the device is ready for the next test app. Our experiment was conducted with two Pixel3 phones with Android 9 for about 33 days.

4.1 Dataset Construction

We crawled 9,771 apps from multiple app stores, among which 5,503 apps are from alternative app stores like Xiaomi [36] and 4,268 apps from Google Play. For each app store, we download the TOP list apps of each category. As shown in Table 2, TraceDroid collected 16.9 GB .pcap files and 6.2 GB call stack files. By parsing these files, we get 301,381 HTTP(S) requests and their corresponding call stacks. We restored 55,843 downloaded files and 2,914 uploaded files from the captured traffic.

Table 2. Manifest of experimental data and result

Data	Count	Description
Android apps	9,771	5,503 from alternative markets and 4,268 from Google
.pcap files	16.9 GB	The volume of captured traffic
Call stack	6.2 GB	The volume of function call stacks
Requests	301,381	HTTPS: 174,486 (57.9%) HTTP: 126,895 (42.1%)
Files	58,757	55,843 downloaded files, 2914 uploaded files

4.2 Liability Analysis

To conduct a liability analysis of privacy leakage between host apps and TPLs, we follow the privacy definition of the previous work [25] to categorize privacy data into three types: 1) device information (e.g., IMEI, Serial Number); 2) Location information (e.g., base station, GPS); 3) Network information (e.g., WiFi state, IP address). We collect the above information of our two Pixel3 phones for further analysis.

In our experiment, TraceDroid identifies 357 TPLs using the method proposed in Sect. 3.3. Based on the result, we conducted a fine-grained liability analysis of privacy leakage, that is, to show whether privacy data is leaked by TPLs or host apps. We denote the requests which transmit privacy data as *privacy requests*, and TraceDroid uses 5-tuple–$\langle appPackageName, source\ IP, sourcePort, destIP, destPort\rangle$ to correlate HTTP(S) requests with call stacks. TraceDroid searches the call stacks of *privacy requests*, if a TPL package name exists in the call stack, we consider the TPL initiates the privacy request; otherwise, it is initiated by the host app. We find that TPLs initiate 44.45% of privacy requests. More detail, about 39% of device information, 45.8% of location information, and 42.6% of network information leakage are transmitted by TPLs. Our result demonstrates that TPLs have become a non-negligible channel for privacy leakage and more regulation should be given to TPLs.

4.3 Privacy Leakage Through Files

Prior works mainly focus on network request analysis. However, files are also an essential part of network traffic, and surprisingly, no work analyzes the content in files, which makes it long been ignored by the research community. In this section, we make the first attempt to examine the contents of files. Benefiting from our packet-level hooking ability, TraceDroid restored 2,914 uploaded files (involving 329 apps). Using our liability analysis, we find that 1,793 files were transmitted by host apps, and 1,121 files were transmitted by TPLs (involving 84 TPLs).

First, we manually analyze files sent by TPLs and find that all these files are not human readable, and most of them do not have a suffix indicating the file type. For further investigation, we inspect these files with a binary viewer to analyze the encoding or encryption method and find that: 1) Files sent by different TPLs have similarities in the file name. For example, 810 file names end with "*stm_d*" or "*stm_p*", and 154 file names end with "*.*.pvuv.log*". 2) The first ten bytes in "*stm_d*" and "*stm_p*" files are the same.

Second, TraceDroid inspects all files generated by host apps and tries to decompress them if necessary. However, due to the various encoding formats, it is difficult to deal with them automatically, so we perform a semi-automatic analysis on these files. We first manually analyze some files to see whether they are human-readable and automatically analyze similar files (either with the same file names or sent by the same host app). Surprisingly, we find that these files tend to contain much more detailed privacy information than requests. For example, a file contains network status (mobile operator APN name, it indicates the operator name or WiFi name), app name, app version, distribution channel, client time stamp, device model, OS version, Android ID, and the rest of this file contains user operations like "user clicks the button in the Main activity at 11:00am". More details are shown in Table 3.

Table 3. Information transmitted through files

Process info	**Memory info**	**CPU info**
process id, thread id	RAM total,free,available	abi,processor,manufacturer, utility
Device info	**Battery info**	**Disk Info**
model, brand	voltage, health, temperature	disk total, available, block size
Root status	**Kernel info**	**Network info**
isRoot, su permission	Linux kernel info, Dalvik version	WiFi, mobile operator, SIM card
Installed app list	**Location info**	**Device IDs**
apps installed in the device	gps, city	IMEI,IMSI,SN,UUID,UDID
User action info	**Permission info**	**Phone number**
pvuv info	permissions app applied	user phone number

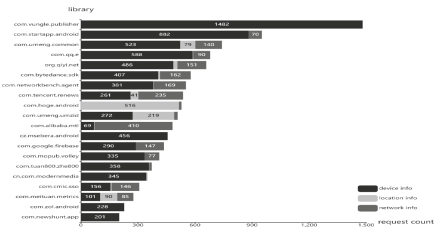

Fig. 4. Privacy leakage from TPLs

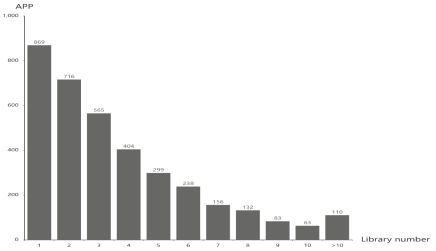

Fig. 5. TPL numbers per app

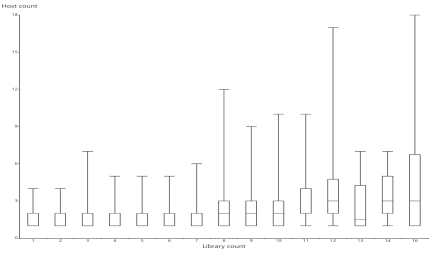

Fig. 6. TPL number&privacy domain

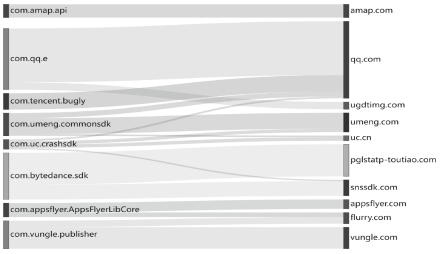

Fig. 7. TPL visiting Graph

To sum up, in this section, we present a fine-grained privacy leakage analysis on files transmitted from user devices. Our findings are as follows: 1) Files transmitted by TPLs have good encoding/encryption, which could prevent the file content from being extracted. However, some TPLs may use the same encoding/encryption method, if someone reverses one TPL, he may have the ability to decode other TPL's files. 2) Files transmitted by host apps do a poor job of protection, and they contain much more detailed privacy data than HTTP(S) requests, which has long been ignored by the research community.

4.4 Privacy Data Harvest

Over-Data Harvest. Figure 4 shows the privacy data transmitted by TOP 20 TPLs. It shows that 4 TPLs only transfer one type of privacy data, 6 TPLs transfer two types of privacy data, and 10 TPLs transfer three types of privacy data. Our further analysis shows that 33% of apps simultaneously transmit privacy data with TPLs, which means both the host apps and TPLs collect privacy data to deliver their service. However, users cannot control when and how often these data be collected and how they will be used.

Another question we intend to answer is, how many TPLs are integrated into an app, and does over-data harvest exists among TPLs? Fig. 5 shows that 78.5% of apps integrate less than 5 TPLs, 18.5% integrate 6 to 10 TPLs, and only about 3% of apps integrate more than 10 TPLs. Figure 6 shows the relationship between the TPL number and the domain number receiving privacy data. We can see that the more TPLs in an app, the more domains receive privacy data, indicating that over-data harvest does exist among TPLs.

Our findings demonstrate that over-data harvest widely exists among host apps and TPLs. Considering the broad integration of TPLs in apps, we point out that app developers should pay more attention to the collection behavior of TPLs, because if the TPLs collect privacy information in violation of regulations, it will have a destructive impact on the reputation of apps.

Cross-Library Data Harvest. Instead of collecting privacy data from the user device or the server, cross-library data harvest (XLDH) is a new attack model in recent years. XLDH refers to the threat model in that a malicious TPL collects privacy data from other TPLs in the same host app. Wang et al. [33] first point out the threat of XLDH. In their study, a malicious TPL checks the presence of the victim TPL in its host app, and uses the Java reflection mechanism to invoke the API of the victim TPL, thus acquiring the privacy data from it. We call this threat model reflection-based XLDH, and Wang et al. proposed a method to identify the abnormal Java reflection mechanism in order to detect this threat.

In our experiment, we bring to light a new XLDH model - instrumentation-based XLDH. In this model, a TPL uses instrumentation techniques to harvest data from other TPLs in the same host app. Specifically, we conducted an analysis based on the idea of "call stack chain inconsistency", that is to say, we check how a TPL is invoked across different apps to see whether there is any inconsistency among them. We select the 20 most commonly used TPLs in our dataset, extract their call stacks, compare their call stack chain across different apps, and search for the "call stack chain inconsistency". For example, we analyze all call stacks of a TPL "*com.uc.crashsdk*" and find that a TPL named "*com.networkbench.agent*" is invoked by it in the host app "*com.wondertek.paper*". However, "*com.uc.crashsdk*" does not invoke "*com.networkbench.agent*" in other apps. To figure out why this phenomenon occurs, we manually read the developer guide for these two TPLs. We found they belong to different vendors, and "*com.uc.crashsdk*" should not invoke "*com.networkbench.agent*" according to their developer guide, which means that such a phenomenon is suspicious. To find out the truth, we then decompile the

app, manually analyze the source code, call stack, and network requests, finding that "*com.networkbench.agent*" instruments "*com.uc.crashsdk*" to harvests privacy data from it, and finally transmits privacy data to the domain "*networkbench.com*", which is in line with the requests we observed in our dataset. Based on the "call stack chain inconsistency" idea, we found 2 apps that have the behavior of instrumentation-based XLDH in our app corpus, and both of them have tens of thousands of downloads from the app store. What's worse, instrumentation-based XLDH can evade the detection method proposed by Wang et al. [33] as it does not rely on the Java reflection mechanism. We believe this is a new XLDH model, and large-scale, automatic approaches should be studied in future research to deal with this threat.

To sum up, in this section, we present an empirical analysis of data harvest in modern apps. Our findings are as follows: 1) Privacy over harvest exists between host apps and TPLs. Developers should pay more attention to the behavior of TPLs, as they may damage the app's reputation if they violate the regulatory regulations. 2) Cross-library data harvest is a new and covert way to collect privacy data, and new methods should be proposed to detect this threat automatically.

4.5 Relationship Between TPLs and Domains

In this section, we aim to give a clear relationship between TPLs and their visiting domains. Libspector [40] concluded that there is no strict 1-to-1 correlation between TPLs and domains, but what exactly is the correlation remains unclear.

We collected the visiting domains of the 53 manually checked TPLs. Figure 7 is the visiting graph of some example TPLs: the left is TPL names, the right is their visiting domains, and the flow between left and right represents the traffic volume. The graph shows that TPL usually visits more than one domain, and a domain may be visited by more than one TPL, which is in line with the conclusion of Libspector [40]. Nevertheless, why does this phenomenon occur? What is the relationship between them? To figure things out, we read the TPL development documents and searched domains in the app source code to study their connections. We found that, for a particular TPL, the visiting domains can be categorized into three types: 1) self-owned domain: owned by the TPL provider, and the domain will be visited by the TPL across different apps; 2) authorization domain: domains providing an authorization mechanism for those who want to use its service. This is useful when some TPLs need to cooperate with each or provide a public service. The authorization mechanism can be a "key" that anyone who wants to use the service must register or apply for a "key" to access the API provided by the domain; 3) host app domain: owned by the host app provider. Apps use this domain to deliver their service.

Here we take an app named "*com.antutu.ABenchMark*" as an example. It integrates a library named "*com.umeng.commonsdk*". For this library, "*umeng.com*" is a self-owned domain, and "*qq.com*" is an authorization domain. Note that there is a request initialized by "*com.umeng.commonsdk*" that visits a domain

named "*autovote.antutu.net*". Figure 7 does not show the domain because the traffic volume between them is too small. But indeed, "*autovote.antutu.net*" is the host app domain for "*com.umeng.commonsdk*". The reason for "*umeng*" visiting "*antutu.net*" is that the TPL has a call-back API for the caller to deliver their service, and when developers invoke it, they can use the call-back API to visit their domains.

To sum up, in this section, we make an effort toward a clear correlation between TPLs and domains. Based on a large number of data analyses, we conclude that there is a many-to-many relationship between TPLs and domain. Further, we categorize the domains into three types and illustrate the reasons behind the visiting phenomenon between TPLs and domains.

5 Conclusion

This paper proposed a new framework TraceDroid for network traffic capturing and analysis. Compared with existing works, TraceDroid is more robust and efficient as it addresses the limitations of traffic mixing, proxy evasion, and SSL pinning. With the help of TraceDroid, we proposed an unsupervised way to identify Third-party libraries (TPLs), and then conducted a large-scale and comprehensive analysis of privacy leakage on 9,771 real-world apps. We present new findings on the privacy leakage caused by TPLs and files, and we also evaluated the phenomenon of over-data harvest between TPLs and host apps. Besides, we bring to light a new and covert data harvest way that should be studied further–instrumentation-based cross-library data harvest. Finally, we make the first attempt to give a clear relationship between TPLs and their visiting domains. To foster future research, we released all of the source code and experiment results on our Github repository.

Acknowledgment. This work is supported by the National Key Research and Development Program of China (No.2019YFB1005205).

References

1. https://developer.android.com/reference/java/net/HttpURLConnection (2021)
2. https://developer.android.google.cn/about/versions/marshmallow/android-6.0-changes?skip_cache=false (2021)
3. Async-http (2021). https://github.com/android-async-http/android-async-http
4. AutoClick (2021). https://github.com/BlcDle/AutoClick
5. BroingSSL (2021). https://boringssl.googlesource.com/boringssl/
6. Caputo, D., Pagano, F., Bottino, G., Verderame, L., Merlo, A.: You can't always get what you want: towards user-controlled privacy on android. arXiv preprint arXiv:2106.02483 (2021)
7. Charles (2021). https://www.charlesproxy.com/
8. Dong, F., et al.: Frauddroid: automated ad fraud detection for android apps. In: Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 257–268 (2018)

9. Dong, F., Wang, H., Li, L., Guo, Y., Xu, G., Zhang, S.: How do mobile apps violate the behavioral policy of advertisement libraries? In: Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications, pp. 75–80 (2018)
10. Fiddler (2021). https://www.telerik.com/fiddler
11. HttpClient (2021). https://hc.apache.org/httpcomponents-client-5.1.x/
12. Li, L., et al.: ICCTA: detecting inter-component privacy leaks in android apps. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 280–291. IEEE (2015)
13. Li, L., Li, D., Bissyandé, T.F., Klein, J., Le Traon, Y., Lo, D., Cavallaro, L.: Understanding android app piggybacking: a systematic study of malicious code grafting. IEEE Trans. Inf. Forensics Secur. **12**(6), 1269–1284 (2017)
14. Li, M., et al.: Libd: scalable and precise third-party library detection in android markets. In: 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), pp. 335–346. IEEE (2017)
15. LibRadar (2021). https://github.com/pkumza/LibRadar
16. Liu, T., Wang, H., Li, L., Bai, G., Guo, Y., Xu, G.: Dapanda: detecting aggressive push notifications in android apps. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 66–78. IEEE (2019)
17. Liu, T., et al.: Maddroid: characterizing and detecting devious ad contents for android apps. In: Proceedings of The Web Conference 2020, pp. 1715–1726 (2020)
18. Lumen (2021). https://www.haystack.mobi/
19. Ma, Z., Wang, H., Guo, Y., Chen, X.: Libradar: fast and accurate detection of third-party libraries in android apps. In: Proceedings of the 38th International Conference on Software Engineering Companion, pp. 653–656 (2016)
20. Meddle (2021). https://meddle.mobi/
21. Okhttp: https://square.github.io/okhttp/ (May 2021)
22. OpenFeign (2021). https://github.com/OpenFeign/feign
23. Razaghpanah, A., et al.: Haystack: In situ mobile traffic analysis in user space, pp. 1–13. arXiv preprint arXiv:1510.01419 (2015)
24. Reardon, J., Feal, Á., Wijesekera, P., On, A.E.B., Vallina-Rodriguez, N., Egelman, S.: 50 ways to leak your data: an exploration of apps' circumvention of the android permissions system. In: 28th USENIX security symposium (USENIX security 2019), pp. 603–620 (2019)
25. Ren, J., Rao, A., Lindorfer, M., Legout, A., Choffnes, D.: Recon: revealing and controlling pii leaks in mobile network traffic. In: Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services, pp. 361–374 (2016)
26. RestTemplate (2021). https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html
27. Retrofit (2021). https://square.github.io/retrofit/
28. Soh, C., Tan, H.B.K., Arnatovich, Y.L., Narayanan, A., Wang, L.: Libsift: automated detection of third-party libraries in android applications. In: 2016 23rd Asia-Pacific Software Engineering Conference (APSEC), pp. 41–48. IEEE (2016)
29. Taylor, V.F., Spolaor, R., Conti, M., Martinovic, I.: Robust smartphone app identification via encrypted network traffic analysis. IEEE Trans. Inf. Forensics Secur. **13**(1), 63–78 (2017)
30. Tongaonkar, A., Dai, S., Nucci, A., Song, D.: Understanding mobile app usage patterns using in-app advertisements. In: Roughan, M., Chang, R. (eds.) PAM 2013. LNCS, vol. 7799, pp. 63–72. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36516-4_7

31. Volley (2021). https://developer.android.com/training/volley/index.html/
32. Wang, H., et al.: Beyond google play: a large-scale comparative study of Chinese android app markets. In: Proceedings of the Internet Measurement Conference 2018, pp. 293–307 (2018)
33. Wang, J., et al.: Understanding malicious cross-library data harvesting on android. In: 30th USENIX Security Symposium (USENIX Security 2021), pp. 4133–4150 (2021)
34. Wang, W., et al.: Constructing features for detecting android malicious applications: issues, taxonomy and directions. IEEE Access **7**, 67602–67631 (2019)
35. Wang, Y., Wu, H., Zhang, H., Rountev, A.: Orlis: obfuscation-resilient library detection for android. In: 2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft), pp. 13–23. IEEE (2018)
36. XiaoMi: Xiaomi app store (2021). https://app.mi.com/
37. Xu, Q., Erman, J., Gerber, A., Mao, Z., Pang, J., Venkataraman, S.: Identifying diverse usage behaviors of smartphone apps. In: Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference, pp. 329–344 (2011)
38. Yang, Z., Yang, M., Zhang, Y., Gu, G., Ning, P., Wang, X.S.: Appintent: analyzing sensitive data transmission in android for privacy leakage detection. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 1043–1054 (2013)
39. Zhan, X., et al.: Automated third-party library detection for android applications: are we there yet? In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 919–930. IEEE (2020)
40. Zungur, O., Stringhini, G., Egele, M.: Libspector: context-aware large-scale network traffic analysis of android applications. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 318–330. IEEE (2020)