An Empirical Study on Android for Saving Non-shared Data on Public Storage

Xiangyu Liu, Zhe Zhou, Wenrui Diao, Zhou Li, Kehuan Zhang

An Empirical Study on Android for Saving Non-shared Data on Public Storage *

Xiangyu Liu[1], Zhe Zhou[1], Wenrui Diao[1], Zhou Li[2], Kehuan Zhang[1]

[1] The Chinese University of Hong Kong, Hong Kong
[2] RSA Laboratories, United States

Abstract. With millions of apps provided from official and third-party markets, Android has become one of the most active mobile platforms in recent years. These apps facilitate people's lives in a broad spectrum of ways but at the same time touch numerous users' information, raising huge privacy concerns. To prevent leaks of sensitive information, especially from legitimate apps to malicious ones, developers are encouraged to store users' sensitive data into private folders which are isolated and securely protected. But for non-sensitive data, there is no specific guideline on how to manage them, and in many cases, they are simply stored on public storage which lacks fine-grained access control and is almost open to all apps.

Such storage model appears to be capable of preventing privacy leaks, as long as the sensitive data are correctly identified and kept in private folders by app developers. Unfortunately, this is not true in reality. In this paper, we carry out a thorough study over a number of Android apps to examine how the sensitive data are handled, and the results turn out to be pretty alarming: most of the apps we surveyed fail to handle the data correctly, including extremely popular apps. Among these problematic apps, some directly store the sensitive data into public storage, while others leave non-sensitive data on public storage which could give out users' private information when being combined with data from other sources. An adversary can exploit these leaks to infer users' location, friends and other information without requiring any critical permission. We refer to both types of data as "non-shared" data, and argue that Android's storage model should be refined to protect the non-shared data if they are saved to public storage. In the end, we propose several approaches to mitigate such privacy leaks.

1 Introduction

The last decade has seen the immense evolution of smartphone technologies. Today's smartphones carry much more functionalities than plain phones, including email processing, social networking, online shopping, etc. These emerging functionalities are largely supported by mobile applications (*apps*). As reported by [1], the number of Android apps on Google Play is hitting 1.55 million.

Most of the apps need to access some kind of users' data, like emails, contacts, photos, service accounts, etc. Among them, some data are to be shared with other apps by

* All vulnerabilities described in this paper have been reported to corresponding companies. We have got the IRB approval before all experiments related to human subjects.

nature, like photos from camera apps, while others are not to be shared (which is called "app-private data" in this paper), like temporary files, user account information, etc. To protect these app-private data, Android has provided multiple security mechanisms. A private folder is assigned in internal storage that can only be accessed by the owner app.

App developers tend to further divide app-private data into sensitive and non-sensitive ones (the reasons are discussed in Section 7). The sensitive data, like user authentication information, are saved to internal storage, while the data deemed as non-sensitive are saved to public storage (including external SD card and shared partition in built-in Flash memory) which lacks fine-grained access control and is open to almost all other apps [3]. At first glance, it seems reasonable and secure to differentiate and save those "non-sensitive" data to public storage. However, this is a dangerous practice for two reasons. First, app developers prone to make mistakes on identifying sensitive data, especially when the data are massive and complicated. Second, some data could be turned into sensitive when combined with data from other apps or publicly available information, even though they are non-sensitive when being examined individually.

In this paper, we investigated a large number of apps, and the results show that many app developers indeed have failed to make right decisions and app-private data that are originally thought as "non-sensitive" could actually leak lots of user privacy (more details are described in Section 4). We also demonstrate one concrete attack example on inferring user's location by exploiting those "non-sensitive" information in Section 5, which further proves the seriousness of the problem.

We argue that the problem identified in this paper is distinct from previous works. It reveals the gap between the assumptions of Android security design (i.e., the security relies on the knowledge of app developers and permissions) and the limitations in real-world (i.e., apps always need to be compatible with all Android versions and all devices and app developers are not security experts). Our study suggests a vast number of Android apps fall into the trap due to this gap, which is much more serious than people's thought before and needs to be addressed urgently. However, such problem neither originates from system vulnerability nor is introduced when user is fooled, and the existing protection mechanisms are therefore ineffective. To bridge the gap, we believe that app developers should scrutinize their code and avoid saving any non-shared data to public storage, no matter if they are identified as sensitive or not. In other words, "public storage" can only be used to save data to be publicly shared with other apps. Another less painful approach requires the update of Android infrastructure by bringing fine-grained access control to current public storage model, which essentially converts shared public storage to non-shared storage.

Our Contributions. We summarize our contributions as follows:

- We revisit Android's public storage model, including its evolution and access control mechanism.
- Our study is the first to examine the privacy leakage on public storage by investigating real-world apps. Some discovered issues are critical and should be fixed as

[3] Although an app needs the corresponding permissions (like `READ_EXTERNAL_STORAGE` or `WRITE_EXTERNAL_STORAGE`, `READ` and `WRITE` for short) to access public storage, these permissions are very common and are usually granted by users without any hesitation since they are requested by most apps.

soon as possible. We manually checked the most popular Android apps on whether they store the data correctly, and the results show that most of them (with billions of installations in total) leave user's private information on public storage. Our large-scale automated analysis further indicates that such a problem exists in a large number of apps. We also show it is possible to harvest sensitive information from very popular apps through a showcase.
- We suggest several approaches to protect app-private data on public storage.

2 Adversary Model

The adversary studied in this paper is interested in stealing or inferring device owners' sensitive information by exploiting the app-private data located at public storage. In order to acquire app-private data, it is assumed that a malicious app with the ability to **read** data on public storage (by requiring the `READ` or `WRITE` permission) and access the Internet (by requiring `INTERNET` permission) has been successfully installed on an Android device. The app will read certain app-private folders selectively on public storage, extract data that can be used for privacy attacks, and then upload them to a malicious server where the data are analyzed to infer victim's sensitive information.

The above assumptions are easy to be satisfied. First, the app requires only two very common permissions. A statistical analysis on 34369 apps (crawled by us) shows that about 94% of apps request `INTERNET` permission, and about 85% of apps request the permission to access public storage, ranking No.1 and No.3 respectively. This malicious app should hardly raise alarm to the device owners during installation. Second, it only uploads data when Wi-Fi is available and therefore it is hard to be detected by looking into data usage statistics. Finally, the app does not exhibit obvious malicious behaviors, like sending out message to premium number or manipulating the device like bot-client, and could easily stay under the radar of anti-virus software.

3 Background of Android Public Storage

Android provides 5 options for an app to store data, including *shared preferences*, *internal storage*, *external storage* ("public storage" referred in this paper), *SQLite databases* and *remote storage*. In this paper, we focus on public storage since the protection enforced is weaker than the other options.

Evolution. Before Android 3.0, *external storage* only includes real external SD card. Since the size of built-in Flash memory is limited in the early stage of Android, external storage is a preferable option for app developers, especially to store large files, like audio files and images. In recent years, we have seen a significant growth of the size of built-in Flash memory (e.g., 64GB), and it turns out internal storage can also hold large files. However, app developers still prefer to consider external storage to hold app's data for two reasons: First, the Android devices with limited built-in Flash memory are still popular, especially in less developed countries or areas; second, the external and internal storage run with different model and are operated with different APIs, which forces the developers to make unneglectable changes if switching to internal storage.

To maximize the use of its storage without incurring additional overhead to developers, Android adopts FUSE [4] to emulate a sdcard daemon (mounted as */data/media*) inside userdata partition (*/data*). File operations on this partition resembles the ones on external SD card and we consider both as public storage.

Access control model on external storage. Access to external storage is protected by various Android permissions, however, our attack aims to retrieve users' private information and hence we only elaborate the details of read access. Before Android 4.1, there is no permission restricting read operations on public storage. `READ` permission is added to Android since then, and an app has to be granted with such permission to read files on public storage. This permission is supported through attaching a Linux GID `sdcard_r` to all the files on public storage and an app (corresponding to a process in Linux) has to be granted with GID `sdcard_r` as well before visiting any file there. A fundamental issue with this model is that there is no finer-grained control over what files are accessible to one app if `sdcard_r` is granted, which exposes one app's private data to all other apps. It is worth noting that the permission `WRITE_MEDIA_STORAGE` introduced from Android 4.4 enforces finer-grained control over write operations, but nothing has been changed for read operations.

4 Survey on information leaks from public storage

It is true that some app-private data is not sensitive. However, the model of letting apps write their private data to public storage relies on a strong assumption that app developers can make right decisions to tell sensitive data from non-sensitive ones. In this section, we present a survey of the information leaks through app-private data stored on public storage, which shows that such an assumption is problematic. The survey includes two parts: the first is a detailed examination on 17 most popular apps, and the other is a more general and large scale study.

4.1 Investigation on popular apps

What apps have been surveyed. According to our adversary model, the attackers are interested in privacy attacks over app-private data, so we have selected 17 most popular apps from three categories: "social networking", "instant messaging" and "online shopping&payment", which are believed to be more likely to touch users' sensitive information. The categories, versions and total users of these apps are shown in Table 1.

How to check app-private data. These apps are installed on three Samsung Galaxy S3 mobile phones. Then we manually simulate three different users on three phones, including account registration, adding good friends, sending message, and etc. Finally, we check the public storage, search sensitive data for each app and classify them.

How the information is leaked. By studying the popular apps, we found 10 of them leak various sensitive information through app-private data on public storage, as shown in Table 2. Such information is leaked in different forms which are discussed as below, and the details are elaborated in Appendix A.

Leak through text files. Some apps store user's profile into a text file. For example, `Viber` directly saves user's name, phone number into a plain text file without any

Table 1. The categories, versions and total users of the popular apps

Category	App Name	Installed Version	Total Users (Millions)
Social networking	Facebook	13.0.0.13.14	900M
	Instagram	6.2.2	100M
	Twitter	5.18.1	310M
	Linkedin	3.3.5	300M
	Vine	2.1.0	40M
	Weibo	4.4.1	500M
	Renren	7.3.0	194M
	Momo	4.9	100M
Instant messaging	WhatsApp	2.11.186	450M
	Viber	4.3.3.67	300M
	Skype	4.9.0.45564	300M
	Line	4.5.4	350M
	KakaoTalk	4.5.3	100M
	Tencent QQ	4.7.0	816M
	WeChat	5.2.1	450M
	EasyChat	2.1.0	60M
Online shopping &payment	Alipay	8.0.3.0320	300M

Table 2. Sensitive information acquired from the popular apps

Sensitive Information	App Name	Content/Remarks
User Identity	Weibo, Renren	UID
	Linkedin	User's profile photo
Phone Number	Viber, Alipay, EasyChat	User's phone number
Email	Weibo, Renren	Registered email
Account	Tencent QQ,Viber,Renren,Momo,Weibo	UID
	WeChat	QQ UID / Phone number
Connection	EasyChat	Call records
	Linkedin	Profile photos of friends
	KakaoTalk	Chatting buddies
	Renren	Friends' UIDs
	WhatsApp	Phone numbers of friends

encryption. User's username, email address[4] are stored by `Weibo` in a file named by the user's UID. Some apps also keep text logs which reveal quite rich information, i.e., `EasyChat` keeps call records in a file, so caller's number, callee's number and call duration can be easily recovered by simply parsing each record.

Leak through file names. We found several apps organize data related to the user or her friends into a dedicated file named with sensitive or non-obvious sensitive information. For example, a file created by `Weibo` is named as user's UID, `WhatsApp` stores user's friends photos with that friend's phone number as file name. They seem meaningless but could have significant privacy implications when combined with other public information, i.e., the owners of the phone numbers acquired from `WhatsApp` can be found by comparing these acquired portraits with photos from user's social networks.

[4] If the user uses her email to register Weibo account. The user also can use phone number to register an account.

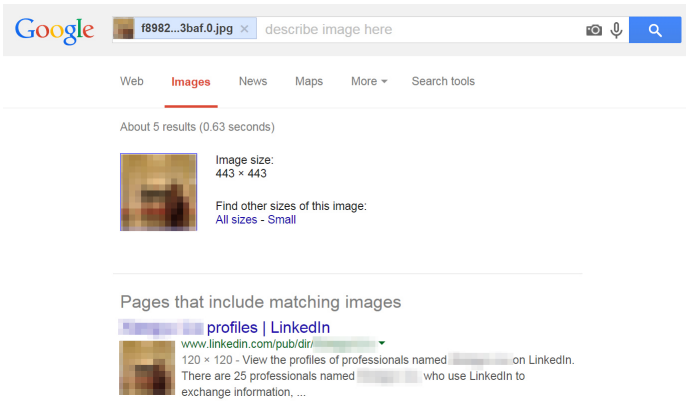

Fig. 1. The result of searching a user's Linkedin profile photo.

Leak through folder names. Some apps use account name as folder name directly, like `Renren` and `Momo`. While `KakaoTalk` will create folders with the same name in both two users' phones if they chat with each other, files (i.e., photos) sent to each other will be saved in the folder and also with the same name. Such a naming convention reveals the connections among people, and even can be leveraged to infer user's chat history.

Leak through photos. The social networking apps usually cache user's profile photos in public storage, like `LinkedIn` in our study. The photo itself is non-sensitive if without knowing who is in the photo, however, our study shows that user's `LinkedIn` profile photo can be linked to her identity by Google image search, as shown in Fig. 1.

Leak through specific patterns. We could use the command */system/bin/sh -c grep -r @xxx.com path* to match and extract email addresses from files in public storage. If the files are stored by apps, the corresponding emails are very likely belong to the phone's owner. It is worth noting that the email found by grep command in Table 2 is from a *log.txt* file left by Renren old version (`5.9.4`).

What information has been leaked. As shown in Table 2, there is indeed some important sensitive information leaked through the app-private data. To better understand the privacy implication of such leaks, we use Personal Identifiable Information (PII) [11], a well-known definition for private data, to classify and evaluate the leaked information. We defined two categories of sensitive data, *Obvious sensitive data* and *Non-obvious sensitive data*, by refining the concept of PII as below:

- *Obvious sensitive data*. It contains identifiers in PII related to user's real-world identity, including full name, phone numbers, addresses, date of birth, social security number, driver's license id, credit card numbers, and etc.
- *Non-obvious sensitive data*. It contains identifiers in PII related to user's virtual-world identity and also her friends' information. The virtual-world identifiers include email addresses, account name, profile photos, and etc.

How to infer user's identity. As shown in Table 2, attackers can exploit the sensitive data left by several apps to infer user's identity information. For example, A user's

Table 3. Privacy protection level of the 17 popular apps

Privacy Level	App Name	Issues
★★★	Facebook, Twitter, Instagram, Skype	-
★★	Line, Vine, WeChat	Audio files without encryption
★	WhatsApp, Linkedin, Viber, KakaoTalk, Momo Tencent QQ, Alipay, Renren, Weibo, EasyChat	Detailed problems are shown in Appendix A

identity can be acquired from her personal homepage by using her Renren UID, Weibo Username/UID, Linkedin profile photo. Moreover, We could find someone on Facebook with a high probability by the email addresses extracted from public storage and also the usernames acquired from other apps, since people prefer to use the same username and email address among their various social networking apps [5].

We divided the apps into three categories with privacy protection level from high to low, and shown in Table 3. Apparently, leaking obvious sensitive data should be prohibited and requires immediate actions from the app developers and Android development team. While our study also demonstrates the feasibility to infer obvious sensitive data from non-obvious ones, therefore the latter should also be well protected.

4.2 Investigation on apps with a large scale

Our study on the popular apps indicates that the sensitive information of device owners could be leaked even from very popular apps. To understand the scale of this problem, we launched a large-scale study on more apps through a customized static analysis tool. Specifically, we first decompile app's apk to `smali` code using Apktool [2] and then search for APIs or strings which indicate storing private data to public storage. Our analysis is conducted on `smali` code instead of decompiled Java code (done by [10,8]) since information could be lost during the code transformation of latter approach. Dynamic analysis, though usually producing more accurate results, is not used here, because it takes long time for even one app to reach proper states (e.g., registration, sending messages, etc.). Again, we focus on the categories described in Table 1 and totally select 1648 different apps from our app repository (34369 apps) for analysis.

Ultimately, our tool should be able to classify the information kept on public storage as sensitive or not, which turns out to be very challenging or nearly impossible without intensive efforts from human. Whether a piece of data is truly sensitive to the device owner depends on the context. We therefore simplifies this task and only checks whether an app **intends** to store sensitive information on public storage. Particularly, if the names of the private folders or files on public storage created by an app contain specific keywords, it is considered as suspicious. Our keywords list include `log`, `files`, `file`, `temp`, `tmp`, `account`, `meta`, `uid`, `history`, `tmfs`, `cookie`, `token`, `profile`, `cache`, `data`, and etc., which are learned from the problematic apps and are usually associated with sensitive content. Some keywords (e.g., `cache` and `data`) appear to be unrelated to sensitive information, but they turn out to be good indicators based on our study, as shown in Appendix A.

For each app, we build a control flow graph (CFG) based on its `smali` code to confirm whether the "*sensitive*" data is truly written to public storage. We demonstrate

Table 4. Methods of accessing public storage

Category	Methods
API Call	*getExternalFilesDir()*, *getExternalFilesDirs()* *getExternalCacheDir()*, *getExternalCacheDirs()* *getExternalStorageDirectory()*, *getExternalStoragePublicDirectory()*
Hardcoded Path	"*/sdcard*", "*/sdcard0*", "*/sdcard1*"

our approach as follows: we start from extracting the method block in `smali` code by finding the texts between keywords *.method* and *.end method*. Next, we select instructions beginning with keywords *invoke-static*, *invoke-direct*, *invoke-virtual* to construct CFG for the method. Then, we check if the methods listed in Table 4 are used to access public storage and whether files or folders are created there (by inspecting methods like `mkdir` and `FileOutputStream`). Finally, we check whether the strings sent to these methods contain keywords in our list. Each function f in the CFG will be marked based on the three criteria. To notice, we do not consider special methods like `touch` as they are also used for other purposes by developers.

We implemented Algorithm 1 on the marked CFG, the depth parameter of the *DFS* procedure was set as 3 empirically, since it resulted in reasonable resources consumption and also yielded high accuracies when examining the known problematic apps. The results show that 497 apps from the 1648 apps being analyzed intend to write some "*sensitive*" app-private data on public storage, which indicate that the privacy leakage problem revealed in this paper is widely exist among apps. However, this method may lead to false positives when an app stores "*sensitive*" data in other places. To have a verification, we randomly chose 30 apps from the suspicious apps, and manually checked them. We found that as large as 27 apps truly wrote "*sensitive*" app-private data on public storage, suggesting this simple static method is valid. This result also suggests there are common patterns among app developers on dealing with sensitive data.

5 Inferring user's location

In this section, we present an example attack based on the non-obvious sensitive information extracted from app-private data. We begin with a brief introduction to the design of a malicious Android app called SAPD ("Smuggle App-Private Data"), followed by detailed description of the attack.

5.1 Attack preparation

The weakest part of the malicious app might be the potential outstanding network traffic footprint, especially for users with limited 3G plan. We implemented two optimizations in our app prototype SAPD to get around this limitation. First, try to minimize the uploaded data since it is reasonable to assume that attackers have already studied the vulnerable apps. Another optimization is to upload data only when Wi-Fi network is available. Instead of using `WiFiManager` which needs to require `ACCESS_WIFI_STATE` permission, SAPD is able to know whether a WiFi network is connected or not by

Algorithm 1 : Detecting suspicious apps

Input: $Class\_set\ C$, $Keyword\_Patterns\_set\ KS$, $Path\_set\ PA$, $Write\_set\ WA$
Output: $bool\ sensitive$

```
1:  for class c in C do
2:      for function f in c do                ▷ Each f has been marked following the rules
3:          condition.clear();
4:          DFS(f, depth);
5:          if condition == Union(KS, PA, WA) then      ▷ All the criteria are met
6:              return true;                            ▷ Marked as suspicious app
7:          end if
8:      end for
9:  end for
10: return false;
11:
12: procedure DFS(function f, int depth)
13:     if depth == 0 then
14:         return;
15:     end if
16:     for all element e in f.mark do
17:         condition(e) = true;
18:     end for
19:     for all callee ce of f do
20:         DFS(ce, depth − 1);
21:     end for
22: end procedure
```

reading public files (*procfs*), since Android puts the parameters of ARP in the file `/proc/net/arp` and other wireless activities in `/proc/net/wireless`. In addition, to minimize the possibilities of being caught due to suspicious CPU usage or abnormal battery consumptions, SAPD will only reads and uploads filtered useful data, and it will never perform any kind of intensive computations.

5.2 Attack framework

Location of a phone user is considered as sensitive from the very beginning and there are already a lot of research works on inference attacks and also protections [15,16,17]. In recent years, location-based social discovery (LBSD) is becoming popular and widely adopted by mobile apps, i.e., `WeChat` and `Momo` investigated in this paper. Though apps adopt some protection mechanisms, i.e., only distance between the user and the viewer is revealed, such location inference attacks are still feasible. Our attack also aims to infer user's location from LBSD networks, but we make improvements since the profile information is extracted from victim's phone, thus leads to a realistic threat. As a showcase, we demonstrate our attack on `WeChat` app.

The LBSD module in `WeChat` is called "People Nearby", through which, the user can view information of other users within a certain distance, including nick name, profile photo, posts (called *What's Up*), region (city-level) and gender. Though `WeChat` UID is not stored on public storage, QQ UID and phone number are stored instead, they are bound to WeChat account and has to be unique for each user. As described in section 4.1, this information has been collected by SAPD and sent to one of our servers (denoted as S_1). These servers are installed with emulated Android environment for running `WeChat` app. S_1 will first create a database by querying the server of Tencent

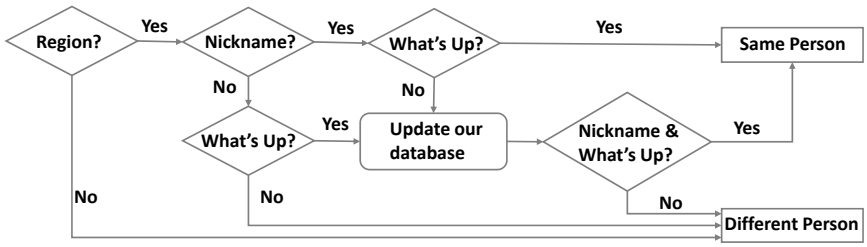

Fig. 2. The diagram of profile information comparison.

(the company operating `WeChat`) for profile information. Then, the attacker needs to instruct another server (denoted as S_2) to run WeChat using fake geolocations, to check People Nearby and to download all the profile information and their corresponding distances. The profile information stored on S_1 is then compared with the grabbed profile information (downloaded by S_2) in another server (denoted as S_3) followed the steps shown in Fig. 2. If a match happens, S_2 will continue to query People Nearby for two more times using different geolocations (faked) to get two new distances. Finally, the target's location can be calculated using the three point positioning method. We elaborate the details of two key steps as below:

Getting Users' Profile Information. The attacker uses QQ userid or phone number to query Tencent server for user's profile information. The returned profile consists of 5 fields: nick name, profile photo, posts (*What's Up*), region and gender. Our task is to assign the location information for each profile. Unfortunately, this profile is updated according to the user's location. What we do here is to frequently retrieve profiles and distances information by faking to different locations. A challenge here is to extract the profiles and distances from `WeChat`, as there is no interface exposed from WeChat to export this information. After we decompile its code, we found the app invokes an Android API `setText` from `android.widget.TextView` to render the text on screen whenever a profile is viewed. We therefore instrument this API and dump all the texts related to profiles into log files. This helps us to extract three fields of a profile, including *Region, Nickname and What's Up* and also its corresponding distance.

Comparing process. The comparison processes performed in S_3 are shown in Fig. 2, which is based on such an observation: People can be distinguished from each other by the three fields (*Region, Nickname and What's Up*). Note that we ignore the special case that different people have the same values of the three fields, since such a possibility is very low due to we require that any of the three fields should not be blank. To avoid the situation that people may have changed her *Nickname* or *What's Up* information before our comparison, we will update her profile by querying Tencent server if only one of them is matched with the data stored in our database.

An app called "Fake GPS location" [3] is leveraged to fake server's GPS to different places. For the densely populated places, we added several more anchor points, since People Nearby only display limited amount of users (about 100). In addition, we use a monkeyrunner script to automatically refresh People Nearby. For each point, to load all the people's profile information, the script will scan people's profile one-by-one through

triggering event KEYCODE_DPAD_DOWN until loading the last one's information. This process has to request data from Tencent. To avoid raising alarm from Tencent, the script sleeps a while before changing to a new anchor point.

Attack evaluation. We evaluate our attack on 20 participants. Each participant has installed *WeChat* with People Nearby turned on (so their profiles will be open to view). Our attack successfully revealed the live locations for 17 participants and have been verified by them. Note that some of the inferred locations are not exact where the user stays, but they are all within the acceptable range, i.e., in a specific residential district.

6 Mitigations

We demonstrate the feasibility of our attacks through the examples above. Without probable countermeasures, more devastating consequences would be caused. Hence, we suggest two approaches enforced by app developers and Android system. The details are described below:

Fixing by app developers. The first suggestion is to ask developers to write *ALL* app-private data to internal storage, which can only be accessed by the folder owner. Though the threat is mitigated, app's functionality could be interrupted when running on devices with limited capacity of internal storage. Moreover, millions of developers are expected to make such change and it is hard to be achieved in the near future.

Patching Android system. On the contrary, modifying the Android system and pushing the upgrades to users' devices would be a more practical way to mitigate the security issues. For this purpose, we propose to augment the existing security framework on public storage by instrumenting the API `checkPermission()`, the framework is described as below:

Architecture. We design a new module named *ownership checker*, which works on Android Middleware layer and can achieve mandatory access control (MAC) for app-private data. Specifically, when the targets are public resources, like music directory, the access is permitted. When the target files are placed under app's private folder, the access is only permitted when the calling app matches the owner. Otherwise, *ownership checker* will return `PERMISSION_DENIED` even if the app has been granted `READ` or `WRITE` permission. To enforce such rule, we create a system file owner_checker.xml storing the mapping between apps and resources, similar to Access Control Lists (ACL) of Ext4 file system. The system code within `checkPermission()` is modified to read the mapping and check the ownership before actual file operations happen. An exception will be thrown if mismatch happens. Alternatively, we could leverage other frameworks like SEAndroid [6] to enforce MAC and protect app-private data.

Ownership inference. The ownership mappings between apps and resources need to be established. This task turns out be non-trivial, since we have to deal with the case that the public storage has already stored apps' data before our module is installed and the owner of data is not tracked therefore. To fix the missing links, we exploit the naming convention: an app usually saves data to a folder whose name is similar to its package name, which can be acquired from packages.xml under `/data/system`). As a starting point, we initialize the mappings by scanning all the resources. For a given resource, we assign the owner app if the resource location and app package name

share a non-trivial portion. To notice, this initialization step could not construct the mapping when an app stores the data in a folder whose name is irrelevant. The access to such resources will be blocked, and we provide an interface for users to manage the ownerships. A new mapping will be added if the ownership is assigned by the user. To reduce hassles to users, user-driven access control model [13] can be integrated to automatically assign ownership based on user's actions.

7 Discussion

Why does the problem persist? In the early stage, Android phone has limited on-board Flash memory (only 256MB for the first android phone, HTC Dream). On the other hand, its storage can be expanded through large volume external SD card, which is usually shipped together. This storage model forces app developers to differentiate sensitive data from non-sensitive data and save the latter (most of the data) to public storage. App developers follow this practice even after recent changes on Android's s-torage model which offers more flexible storage options (i.e., the sdcard dameon (fused) and userdata */data* share the same partition dynamically).

Limitations of app study. We built a tool running static analysis on app's `smali` code and use a set of heuristics to determine if the app saves "sensitive" app-private data to unprotected public storage. This simple tool identifies a large number of potentially vulnerable apps and shows reasonable accuracy from our sampling result. However, it is inevitably suffers from false negatives (e.g., the file name does not contain the keywords we used) and false positives (e.g., the information saved is not sensitive). We leave the task of building a more accurate detector as future work.

8 Related Work

Attacks like stealing users' chat history [7] have been proved feasible in the real world. However, these attacks usually depend on certain vulnerabilities identified from the victim apps, while our attacks exploit a more general problem related to Android's storage model. In addition to steal user's sensitive information directly, a lot of research focused on inferring user's location. The authors of [15] showed a set of location traces can be de-anonymized through correlating their contact graph with the graph of a social network in spite of the data has been obfuscated. Based on a large-scale data set of call records, Zang et al. [16] proposed an approach to infer the "top N" locations for each user. A recent work [17] by Zhou et al. targeted to infer information of users from more perspectives, including identities, locations and health information.

To defend against the existing or potential attacks tampering user's privacy, a bunch of defense mechanisms have been proposed. Ongtang et al. proposed a finer-grained access control model (named Saint) over installed apps [12]. FireDroid [14], proposed by Russello et. al., was a policy-based framework for enforcing security policies on An-droid. Roesner et al. proposed user-driven access control to manage the access to private resources while minimizing user's actions [13]. Besides, efforts have also been paid on code analysis to block the information leakage. Enck et al. developed TaintDroid [9] to prevent users' private data from being abused by third party apps.

9 Conclusion

It is known that public storage on Android is insecure, due to its coarse-grained access model. Therefore, it is highly recommended that the sensitive data should be avoided from saving there. In this paper, we carry out a large-scale study on existing apps on whether app developers follow this rule and the result turns out to be glooming: a significant number of apps save sensitive data into the insecure storage, some of the problematic apps are even ranked top in Android market. By exploiting these leaked data, it is possible to infer a lot of information about the users, drastically violating users' privacy. We urge app developers to fix the vulnerabilities. Besides, we also propose an approach to patch Android system with MAC support and envision it could mitigate the threat in the short term.

Acknowledgments. We would like to thank the anonymous reviewers for their valuable comments. This work was supported by the Direct Grant of The Chinese University of Hong Kong with project number C001-4055006.

References

1. Android apps on google play. http://www.appbrain.com/stats/number-of-android-apps.
2. Apktool. http://code.google.com/p/android-apktool/.
3. Fake gps location. https://play.google.com/store/apps/details?id=com.lexa.fakegps.
4. Filesystem in userspace. http://fuse.sourceforge.net/.
5. 'like' it or not, sharing tools spur privacy concerns. http://usatoday30.usatoday.com/tech/news/2011-07-05-social-media-privacy-concerns_n.htm.
6. Seandroid. http://seandroid.bitbucket.org/.
7. Whatsapp user chats on android liable to theft due to file system flaw. http://www.theguardian.com/technology/2014/mar/12/whatsapp-android-users-chats-theft.
8. K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.
9. W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, volume 10, pages 1–6, 2010.
10. C. Gibler, J. Crussell, J. Erickson, and H. Chen. *AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale*. 2012.
11. E. McCallister. *Guide to protecting the confidentiality of personally identifiable information*. Diane Publishing, 2010.
12. M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 5(6):658–673, 2012.
13. F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Security and Privacy (SP), 2012 IEEE Symposium on*.
14. G. Russello and et.al. Firedroid: hardening security in almost-stock android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, pages 319–328. ACM, 2013.

15. M. Srivatsa and M. Hicks. Deanonymizing mobility traces: Using social network as a side-channel. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 628–637. ACM, 2012.
16. H. Zang and J. Bolot. Anonymization of location data does not work: A large-scale measurement study. In *Proceedings of the 17th annual international conference on Mobile computing and networking(MobiCom)*, pages 145–156. ACM, 2011.
17. X. Zhou, Demetriou, and et.al. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1017–1028. ACM, 2013.

A The details of user private data

Viber. The text file *.userdata* in *.../.viber/* [5] reveals lots of user's information, including real name, phone number, and the path of user's profile photo.

WhatsApp. The user's profile photo is stored in *.../.shared/* with file name *tmpt*. The profile photos of user's friends are saved under *.../Profile pictures/*, and they are named by profile owners' phone numbers without any obfuscation.

Linkedin. This app cache the photos into the directory */Android/data/.../li_images/*. The user's profile photo can be distinguished by file size and modified time.

KakaoTalk. If user A has chatted with user B, the app will create a content folder with the same name in both users' phones, under the path */Android/data/.../contents/*. The files, i.e., photos, on the two phones also have the same name, size and the same path.

Tencent QQ. User's account can be got from log files in the path *.../mobileqq/*.

Weibo. A file named as user's UID is saved under the path *.../page*, and we can acquire the user's username and her email address. User's username and UID can be leveraged to access her homepage by constructing specific URLs, i.e., *http://www.weibo.com/UID*.

Alipay. User's phone number can be obtained from the *_meta* file in *.../cache/*, it also points out the other file which discloses the user's phone number.

Renren. A folder named by user' UID is stored in */Android/data/.../cache/*. Even user' visit histories are also stored in this folder, which contains the name, UID of user's friends. The audio files are named as the format *UID+hash value*. We can find the user's personal home page by the URL *http://www.renren.com/UID* in a browser.

Momo. A folder named as user's account is saved in *.../users/*. By the account, we can not only get her profile information, but also infer her location.

EasyChat. The file *pjsip_log.txt* in */Yixin/log/* contains all the call records information.

Audio files. Instant message apps, like WhatsApp, Line, WeChat, Tencent QQ, and KakaoTalk, store the audio files into public storage without encryption.

[5] We use ... to represent part of the full path since sometimes the full path is too long.