A Fine-Grained File Access Monitoring and Control System on Android Platform

Yang Liu, Lei Wang
Department of Computer Science
University of Calgary
{*yang.liu5, lei.wang*}*@ucalgary.ca*

***Abstract*—Smartphones have become an essential part of everyone's daily life. When users enjoy the convenience of applications, some applications steal privacy information, scan photos and videos secretly. For example, because of Android's unreasonable granularity of permissions of reading and writing external storage, uncountable applications could scan user's photos without notifying users. This paper presents a fine-grained file access monitoring and control system on Android platform, to help users examine and control file accesses from applications.**

Keywords: Android, Privacy, File access.

I. Introduction

Mobile phones are becoming an essential part of everyone's daily life. The statistic shows that the number of mobile phone users is predicted to be 4.68 billion, and the percentage of people owning phones over all people in worldwide, is predicted to be 67 percent, in 2019 [1]. Android operating system takes 75.33% of all mobile operating system market, which is the largest market share, in March 2019 [2].

The increasing popularity of Android platform also makes it the biggest target of malicious applications. According to a report by Kaspersky, 98.5% of mobile malicious applications target Android platform, and the number is over 1 million in 2013 [3]. The openness of Android platform and the density of privacy information in smartphones give attackers opportunities to steal for profits.

To restrict the range of malicious behaviors and diminish the bad effect of malware, applications on Android platform are under the control of the permission system. An app can only read related sensitive information when it's granted corresponding permissions, which are decided by end users. Android permissions were granted when an application was installed. Users had to agree with these permissions if they want the application installed. Most recently, Google, the maintainer of Android Open Source Project, has upgraded the permission system and makes it more controllable. In settings, Users can grant or remove a permission at any time and an Ask-On-First-Use model is introduced to Android system. The user is explicitly asked via a popup dialog, for granting a permission to an application, in which way users could be better aware of the permission-granting process.

However, this permission system is still not perfect. Some applications are pretty assertive and they will refuse to work if permissions are not fully granted. The compulsion put users in a dilemma and users has to agree with this if necessary.

Apart from assertions, the other problem existing in the current permission system is the granularity. Some permissions are coarse-grained while users need granting permissions in a fine-grained manner. For example, there are 2 coarse-grained permissions: READ_EXTERNAL_STORAGE and WRITE_EXTERNAL_STORAGE. Each application granted with these 2 permissions could read or write every single file in the external storage, including photos, audios, videos, and downloaded files. These files, which contain a huge load of personal information and privacy, could be leaked very easily if malicious applications are granted these coarse-grained permission. To In this paper, a system is proposed to help users inspecting and controlling the file accesses of every application.

Contributions of this paper are listed as follows:

1) A fine-grained file monitoring and control system is proposed, designed and implemented. This system could track and limit the file accesses of every application.
2) An Android GUI application is developed to show file accesses and set file access policies in an intuitive way.
3) File accesses by 5 popular Android applications are evaluated.

The rest part of this paper is organized as follows.

Section II introduces background of this project. Section III introduces some related work. Section IV discusses the design and implementation details of this project. Section V shows the intuitive interface designed for end users. Section VI discusses the file accesses in mainstream modern Android applications. Section VII is about the existing limitations in this system and our future work. Section VIII is the conclusion of this project.

II. Background

A. Android file system

Android is an open-source mobile operating system built on Linux. The file system of Android is managed by the Linux Kernel. For each application in user space, there are 2 kinds of storage type: Internal storage and External Storage.

Internal storage

For each application, the path of internal storage is different, depending on the package name of the application. By default, the root folder of an internal storage is "/data/data/<package name>/". Internal storage is usually used for storing application-related information or cache. An application are not required any permission to access internal storage.

Applications are not allowed to read other applications' internal storage. Besides, all content of the internal storage is deleted once the application is deleted. In general case, internal storage is considered more secure than external storage.

External storage

External storage is shared by all applications installed on a phone. The default path of external storage is "/storage/emulated/0/" and it's mounted to "/sdcard/". External storage is only accessible by the application is granted "READ_EXTERNAL_STORAGE" or "WRITE_EXTERNAL_STORAGE". Applications can store files and cache on external storage as well, but every file on external storage could be read by other applications granted these permissions.

External storage contains photos, videos and audios taken by users, which are personal privacy. Besides, applications

should be aware of the files they put on external storage, which might result in privacy leakages.

Other locations

"/system/" contains system-related files, such as libraries and binaries. "/mnt" directory are intended for use as the temporary mount points. "/root" is the home folder for the root user. "/etc" is a directory for configuration files.

B. Logging system in Android

Kernel logging system

Android Operating System's kernel is based on the Linux kernel's long-term support (LTS) branches. In linux kernel, the function is used for printing messages is "printk", which is also supported in Android kernels. It's a printf-like function, which accepts a string parameter called the format string and an arbitrary number of values. "printf" is a function of standard C library, which is cannot be used in kernel level, hence the need for "printk".

Messages printed by "printk" could be viewed through the command "dmesg". "dmesg" is a command on most Unix-like operating systems, including the Android OS. It could print the message buffer of the kernel.

Android logging system

Both Java Android programs and native Android programs could use Android logging system to print messages. Native programs and Java programs use "liblog" to record messages, which could be viewed by using command "logcat".

For Java programs, Android Software Development Kit (SDK) provides "android.util.Log", which is an API to print messages. For native Android programs, "log.h" is used for printing messages to Android logging system.

C. File Accesses from programs

Java programs

In Unix-like operating systems, files can only be accessed by using system calls in kernel layer. For Java programs in Android OS, to make Java capable of accessing files in Android file systems, Java Native Interface (JNI) is used as a medium layer laid between Java VM layer and Kernel layer.

When an file operation in Java code is triggered, Android SDK would use C code written in JNI, and then call the system functions to perform the file operation.

Here we use the opening operation in Java class "DataInputStream" as an example. The constructor of "DataInputStream" triggers the instructor of "FileInputStream", which calls the "open" method in "FileInputStream". The "open" method would call "open0" in the same class, which is implemented in C Programming Language by Java Native Interface. The method "open0" calls system call "open" to open a file on storage.

Native programs

A native program could written in C or C++, they have the ability to use system calls directly. For example, native programs could call the system call "open" directly to open a file.

Fig. 1 shows the different call stacks of opening operations from different programs.

D. Android applications and User ID

In Android OS, each application is assigned an unique user ID while installed. The mappings from Android applications to user IDs could be fetched from "/data/system/packages.list". Fig. 2 shows part of content of this file, which contains the package name, assigned user ID, and internal storage path, etc.

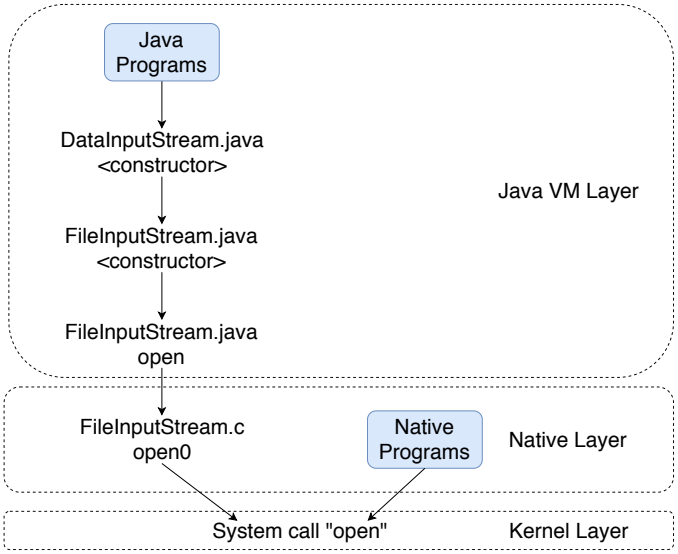


Fig. 1. File opening operations from different programs

III. Related work

Android system has a logging facility that allows systems-wide logging of information, it consist of two parts: Java program and Native program, specially, java program generate java application log and native programs that are written by C/C++ generate system log. A lot of studies have been on analysis on Android logs. Takamasa et al.[4] present a kernel-based behavior analysis system for android malware detection that consists analyzer on both android device and PC, they collected target activities log and analyzed it by signature-based pattern matching. File access in Android require permission, in standard Android platform, the apps can be granted the permission permanently once approved by the user. Primal Wijesekera et al. proposed a system to dynamically grant and revoke the permission depending on the context [5]. A study analyzing on Android storage performance is introduced in [6], in which they present an Android storage performance analysis tool that generating a IO pattern to capture characterisctic for file system.

IV. Design and Implementation

A. Design overview

In this section, the design and the implementation of our file access monitoring and control system are discussed. The system contains 2 parts: a file access monitoring system and a file access control system.

B. File access monitoring system

By adding hooks to procedures of file accesses in various layers, the file access monitoring system could get the record file operations in both kernel layer and native layer. The reasons why both layers are monitored are as follows.

1) Hooks in native layer are able to catch all file accesses from java code. However, it cannot catch file accesses from native programs or native code of java programs.
2) Hooks in kernel layer are able to catch all file accesses from all applications. However, it contains a large number of unimportant file accesses, such as dynamic libraries, which would introduce bias

```
com.android.cts.priv.ctsshim 10007 0 /data/user/0/com.android.cts.priv.ctsshim default:privapp:targetSdkVersion=24 none
com.android.providers.telephony 1001 0 /data/user_de/0/com.android.providers.telephony platform:privapp:targetSdkVersion=27 3002,3003,3001,3007,3006
com.android.providers.calendar 10003 0 /data/user/0/com.android.providers.calendar default:privapp:targetSdkVersion=25 3003
com.android.providers.media 10010 0 /data/user/0/com.android.providers.media media:privapp:targetSdkVersion=27 2001,1023,1015,3003,3007,1024
com.android.wallpapercropper 10028 0 /data/user/0/com.android.wallpapercropper platform:privapp:targetSdkVersion=27 none
com.android.documentsui 10009 0 /data/user/0/com.android.documentsui platform:privapp:targetSdkVersion=27 none
com.android.externalstorage 10011 0 /data/user/0/com.android.externalstorage platform:privapp:targetSdkVersion=27 1023,1015
com.android.htmlviewer 10046 0 /data/user/0/com.android.htmlviewer default:targetSdkVersion=23 none
com.android.companiondevicemanager 10039 0 /data/user/0/com.android.companiondevicemanager default:targetSdkVersion=27 3002,3001
com.android.quicksearchbox 10058 0 /data/user/0/com.android.quicksearchbox default:targetSdkVersion=24 3003
com.android.mms.service 1001 0 /data/user/0/com.android.mms.service platform:privapp:targetSdkVersion=27 3002,3003,3001,3007,3006
com.android.providers.downloads 10010 0 /data/user/0/com.android.providers.downloads media:privapp:targetSdkVersion=27 2001,1023,1015,3003,3007,1024
com.android.messaging 10062 0 /data/user/0/com.android.messaging platform:targetSdkVersion=24 3003
com.google.android.launcher.layouts.bullhead 10032 0 /data/user/0/com.google.android.launcher.layouts.bullhead default:targetSdkVersion=27 none
```

Fig. 2. Part of content of /data/system/packages.list

Kernel hooks

To monitor file accesses in kernel layer, we have to add hooks to the Android kernel. As we mentioned in section II-B, Android kernels use the function "printk" to print messages, and these messages could be collected by using "dmesg".

When the system call "open" is triggered, the kernel uses "printk" to print a message, including time, user ID, process ID and target file path. We could add a tag to output messages, for filtering out the messages not of interest.

Java Native Interface (JNI) hooks

JNI are used as a medium between Android SDK and system calls in kernel, as mentioned in Section II-C. In the monitoring system, some hooks are added to JNI, printing a message using Android logging system, when an application tries to open a file in Java code. Each message contains time, user ID and target file path. A tag is also included in the message for applying filters.

Log filters and forwarders

Log filters are for filtering out unrelated messages and keep the messages of interest. Because a custom tag is added to each message, no matter in kernel layer or in JNI layer, it's convenient for a filter to keep the messages in the same tag.

Original logs are stored in logging systems, but after filtering, log filters forward filtered messages to a local file, for the ease of use by analyzers.

Log analyzers

Log analyzers are required for analyzing the content of each filtered log file. There are 2 log analyzers: a kernel log analyzer and a JNI log analyzer. One duty of log analyzers is to match the time, the application, and the target file path for each message.

As mentioned in Section II-D, each application is assigned an unique user ID. Thus by searching the user ID from "/data/system/packages.list", the specific package name who performs the file operation could be found. Android API provides a way to lookup the application name by a package name.

Another duty of log analyzers is to decide the file type of each record. The basic idea is distinguishing file types by extensions of target files. This method is fast but inaccuracy. This could be improved by using a more sophisticated way, such as machine learning. More about this topic is discussed in future work (Section VII).

C. File access control system

File access control system allows users to define a policy to restrict arbitrary file accesses from an arbitrary application. The restriction policy contains multiple restriction rules, and it is a black list. If an application would like to access a file defined in the policy, the system would automatically redirect the target to a fake file. Users could use the control system to block sensitive files from some applications.

Policy format

Policy is defined in a file, by default, whose path is "/sdcard/log/rules.conf". Each line is a rule, including 3 fileds: application user ID, blocked file path, fake file path. Fields are separated by a comma. Application user ID and blocked file path are mandatory, while the fake file path is optional. If a fake file path is not defined, it will be "/data/system/tmp/fake.txt" by default.

Target Redirection

Target redirection is performed in JNI layer. To open a file, Java code would call the method "open0" in JNI layer, one of whose argument is the target file path. In the control system, A hook is added to "open0", try to matching rules one by one. If one of rules is matched, the argument of file path will be modified to the fake path. If the policy is enabled, the application accesses the fake file without being aware.

D. Intuitive GUI application

This GUI application is for providing an intuitive way to present users file access records from all applications installed on the device. Users could examine file accesses classified by applications, or by file types.

Another function of this GUI application is managing the restriction policy. Users could edit the policy in an intuitive way. For example, the first field of each rule is an application user ID, however users cannot easily obtain the user ID of a application. This application allows users to choose the application name, and it will be converted to user ID once the policy is saved.

E. Implementation

Kernel log and JNI log collection, filter, and forwarding

Each file opening operation in the kernel layer prints one message in kernel logging system, while in JNI layer, messages are printed in Android logging system. Each message is labelled by a tag "YANG". The kernel log collection, filter and forwarding are completed by an Android service, which constantly runs the following command in background:

```
dmesg -w | grep --line-buffered YANG >>
/sdcard/log/kern.log
```

The JNI log collection, filter and forwarding are handled by another Android service, which runs the following command:

```
logcat -w | grep --line-buffered YANG
>> /sdcard/log/jni.log
```

TABLE I. MAPPINGS FROM FILE TYPE TO FILE EXTENSION

File type	File extension
Image	png, jpg, jpeg, tif, png, bmp, svg, ico
Video	mp4, mkv, webm, flv, ogg, mov, wmv, rm, rmvb, m4v, mpg, 3gp
Resource	xml, apk
Library	so, a
Text	txt, log
Audio	aac, ape, au, flac, mp3, ogg, raw, wav, wma
Data	data, db, database, databases

Kernel Modification

The kernel version this system based on is "android-msm-bullhead-3.10-oreo-m7". One hook is added to the kernel, located in file "fs/open.c". In the function "do_sys_open", before the procedure opening the file, a piece of code is added to print message, including a tag "YANG", current time, current user ID, and the target file path. [1]

JNI Modification

JNI is a part of Android OS. The Android OS version this system based on is "android-8.1.0_r52". The function modified is "FileInputStream_open0" in "libcore/ojluni/src/main/native/FileInputStream.c". First, to support monitoring file accesses, a function is created and called to print current time, user id and target file path. Second, to support file access restrictions, another function is created and called to redirect the target if one of rules is matched. [2]

File access monitor and controller

The file access monitor and controller is an intuitive Android application written in Java. It provides an unified way to view file accesses from different applications, and define rules to block file accesses from applications. [3] File type classification is developed in it. For now, file type are decided exclusively based on the file extension. According to Table I.

File reader

This is a Android application to read content from "/sdcard/log/test.txt", and then show the content on screen. [4] This application is mainly used for testing if the file restriction policy is enabled.

F. Obsolete design

V. INTERFACE

This section shows the intuitive interface of the proposed monitoring and control system by this paper.

The interface of our monitoring and control system consists of the 5 parts as show in main menu in Fig. 3. Users could choose the logging system they would like to examine, and the characteristic to partition them, by application name or by file types.

[1]Refer to source code: https://gist.github.com/Yang-Jace-Liu/9fd78d7b3f78776233205397ca03c99f

[2]Refer to source code: https://gist.github.com/Yang-Jace-Liu/9fd78d7b3f78776233205397ca03c99f

[3]Refer to source code: https://github.com/Yang-Jace-Liu/Android-file-access-monitor

[4]Refer to source code: https://github.com/Yang-Jace-Liu/Android-file-access-monitor

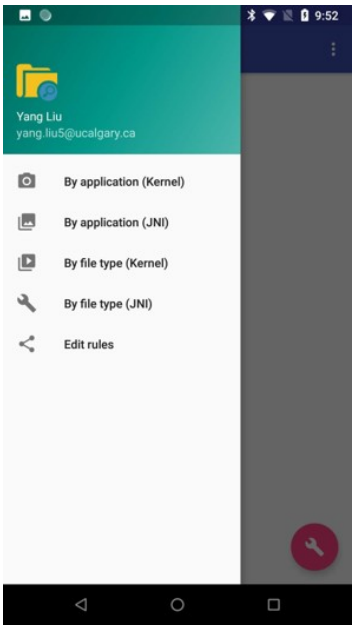
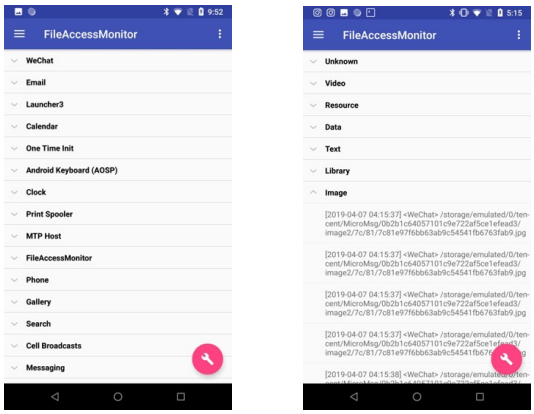


Fig. 3. Interface of main menu



(a) Partitioned by app (b) Partitioned by file type

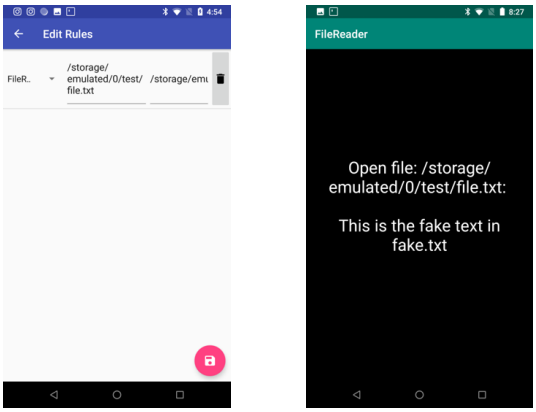
Fig. 4. Interface to examine file accesses

A. Interface of file access monitoring system

Fig. 4(a) and Fig. 4(b) are the interface for users to examine file accesses from applications. In the interface of Fig. 4(a), users are able to examine file accesses from applications of interest. Application names are shown because package names and user IDs are confusing to end users. In the interface of Fig. 4(b), users could examine file accesses in terms of file types, which could help end users be aware of when and who has scanned their photos, videos, and files.

B. Interface of file access control system

To facilitate users in controlling file accesses, an interface for editing file restriction rules is provided. As shown in Fig. 5(a), in which the option box on left enable user to select the app that they want to restrict, the other two text fields hold the path of the restricted file (middle) and the fake file to be redirected to (right), that is, when the application selected in the option box wants to access the file in the middle text field, it will access the file in the right text field instead.



(a) Rules editor (b) File reader

Fig. 5. Interface of file access control system

To show the effect of this rule, we designed a simple file reader as shown in Fig. 5(b). File reader would read the content of a file located at "/storage/emulated/0/test/file.txt", and then print the content on the screen. The Fig. 5(b) shows that, because of one of rules is active, target being redirected to a fake file, the file reader is reading a fake file named "fake.txt".

VI. Case Study

In this section, 5 mainstream Android applications are selected: Facebook, Messenger, Instagram, Wechat and Twitter. These apps are installed on the test phone (Nexus 5X) and each runs 6 munites in foreground and 24 minutes in background and then we analyze their file access information. We focus on the following 3 metrics of measurement:

- Top file extensions being most accessed.
- Number of file access times of each application.
- Distributions of file access targets of each application.

A. File extensions being most accessed

The first measurement we present in this paper is the top file extensions being most accessed on both JNI layer and kernel layer. Fig. 6 and Fig. 7 are top 10 most accessed file types in kernel layer and JNI layer.

Fig. 6 indicates that the most accessed file type in kernel layer are "so" files, which are dynamic libraries used in Linux and Android operating systems. In general case, "so" files are not directly used in applications, but used in API calls. It proves one of reasons why we need file access records in JNI layer in Section IV-B: There are unimportant file accesses which could introduce bias.

Fig. 7 shows that the most accessed file type in JNI layer are "xml" files. "xml" files are a type of resource wildly used in different applications.

B. Number of file accesses

Fig. 8 and Fig. 9 show the number of file accesses of different applications in the duration of experiment. In both layers, Wechat contributes the highest number file accesses while Twitter contributes the lowest number in these 5 applications.

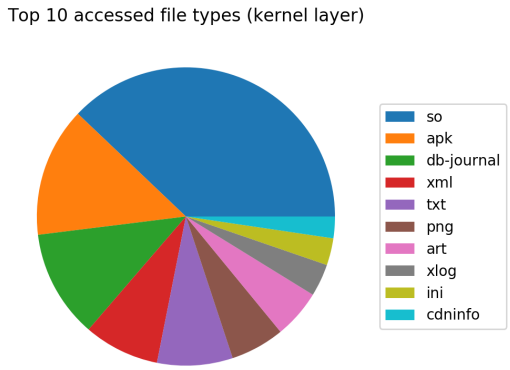


Fig. 6. Top 10 file extensions accessed in kernel layer

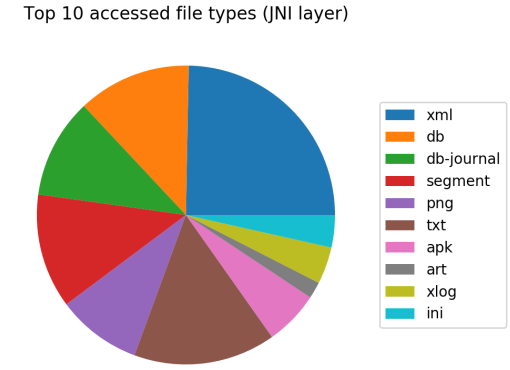


Fig. 7. Top 10 file extensions accessed in JNI layer

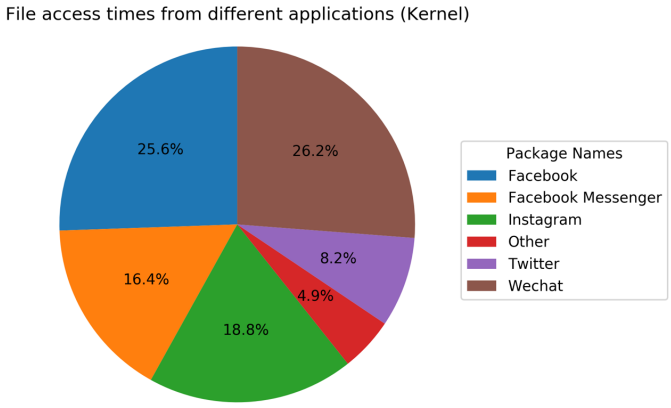


Fig. 8. Number of file accesses in kernel layer

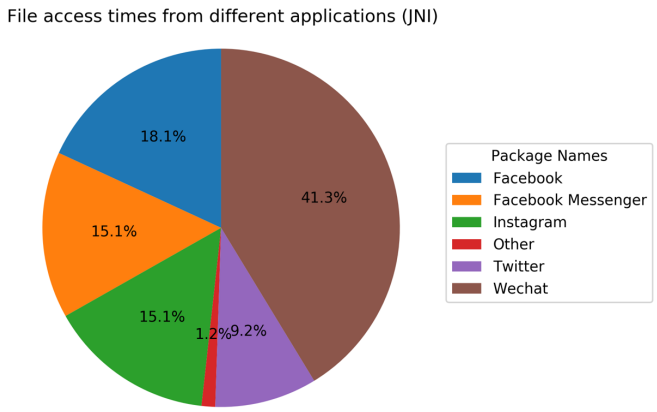


Fig. 9. Number of file accesses in JNI layer

C. Distribution of file access targets

To mitigate the bias introduced by unimportant files, this experiment is based on the file accesses in JNI layer. In this experiment, file access locations are divided into 3 categories: internal storage, external storage, and system storage. As mentioned in II-A, internal storage is for application-related information and it's only accessible by the application, external storage is accessible by all applications granted the permission.

In order to find out whether external and internal storage is more accessed, we did an analysis of accessed file distribution on device storage for each application. As shown in Fig. 10, Facebook, Messenger and Instagram only have a small ratio on external storage access, which may indicate that they have a higher protection on privacy; On the other side, WeChat and Twitter both have a considerable part of external file accessed, this means they have a higher probability of invading user's privacy.

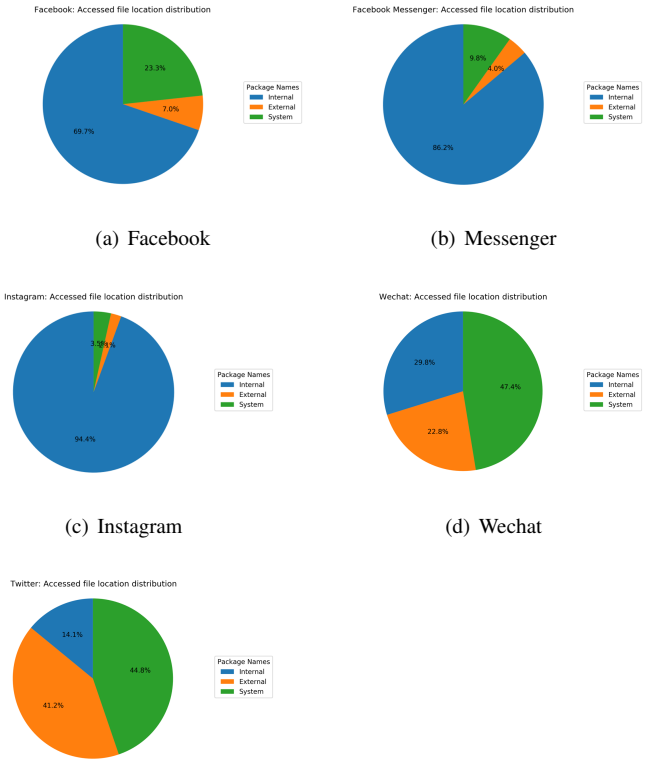
Potential security glitch: Because the high load of external file accesses by WeChat and Twitter, we look into the target paths, and find that WeChat saves message histories on external storage while Twitter stores picture and video caches on external storage. This could be a potential security glitch and may introduce privacy leaks.

VII. Limitations and Future Work

This paper proves that file accesses could be monitored and controlled. However there are still some limitations existing in this system.

1) The file type classification for now is inaccuracy, because it's decided by file extensions. Files could be video, audio, database, resources, but without any file extension.
2) Each file access restriction rule for now only supports blocking a file. Users cannot apply one rule to a folder.
3) File access restriction policy can only applied to Java programs, not capable of blocking native programs

For future work, (1) the file type classification algorithm should be improved by using a more sophisticated method, such as machine learning. (2) The file access restriction policy should support blocking folders.



(a) Facebook (b) Messenger

(c) Instagram (d) Wechat

(e) Twitter

Fig. 10. file location distribution on each application

VIII. Conclusion

In this paper, one fine-grained file access monitoring and control system is proposed, designed and implemented. Users could examine file accesses of installed applications in both JNI layer and kernel layer. On the one hand, for beginning users, an intuitive GUI android application is provided to help examining file accesses. On the other hand, for advanced users, we provide an interface to define a set of rules, to block files from applications. As a part of this research, we measured the file accesses of 5 mainstream Android applications: Facebook, Messenger, Twitter, Instagram and WeChat, and found that WeChat and Twitter store sensitive information on external storage, which could be a potential security glitch.

References

[1] Statista, "Number of mobile phone users worldwide from 2015 to 2020 (in billions)," 2019.

[2] Statcounter, "Mobile Operating System Market Share Worldwide," 2019.

[3] Kaspersky, "Kaspersky Security Bulletin 2013," 2013. [Online]. Available: https://media.kaspersky.com/pdf/KSB_2013_EN.pdf

[4] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for android malware detection," in *2011 Seventh International Conference on Computational Intelligence and Security*, Dec 2011, pp. 1011–1015.

[5] P. Wijesekera, A. Baokar, L. Tsai, J. Reardon, S. Egelman, D. Wagner, and K. Beznosov, "Dynamically regulating mobile application permissions," Feb 2018.

[6] S. Jeong, K. Lee, J. Hwang, S. Lee, and Y. Won, "Androstep: Android storage performance analysis tool," in *Software Engineering 2013 - Workshopband*, S. Wagner and H. Lichter, Eds. Bonn: Gesellschaft für Informatik e.V., 2013, pp. 327–340.