Why Data Deletion Fails? A Study on Deletion Flaws and Data Remanence in Android Systems

Junliang Shu, Shanghai Jiao Tong University
Yuanyuan Zhang*, Shanghai Jiao Tong University
Juanru Li, Shanghai Jiao Tong University
Bodong Li, Shanghai Jiao Tong University
Dawu Gu, Shanghai Jiao Tong University

Smart mobile devices are becoming the main vessel of personal privacy information. While it carries valuable information, the data erasure is somehow much more vulnerable than one has foreseen. The security mechanisms provided by Android system are not flexible enough to thoroughly delete those sensitive data. Besides the weakness among the several provided data erasing and file deleting mechanisms, we also target the Android OS design flaws in data erasure, and unveil that the design of Android OS contradicts some secure data erasure demands. We present the data erasure flaws in three typical scenarios on mainstream Android devices, such as *data clearing flaw*, *application uninstallation flaw* and *factory reset flaw*. Some of these flaws are the inherited data deleting security issues from Linux kernel, and some are new vulnerabilities in Android system. Those scenarios reveal the data leak points in Android systems. Moreover, we reveal that the data remanence on the disk is rarely affected by the user's daily operation, such as file deletion, app installation and uninstallation, by a real-world data deletion latency experiment. After one volunteer has used the Android phone for 2 months, the data remanence amount is still considerable. Then, we proposed *DataRaider* for file recovering from disk fragments. It adopts file carving technique and is implemented as an automated sensitive information recovering framework. *DataRaider* is able to extract private data in raw disk image without any file system information, and the recovery rate is considerably high in the four test Android phones. In the end, we propose some mitigation for data remanence issues, and give the users some suggestions on data protection in Android systems.

CCS Concepts: •**Security and privacy** → **Mobile platform security;**

Additional Key Words and Phrases: Data recovering, secure deletion, file carving, mobile security

ACM Reference Format:
ACM Trans. Embedd. Comput. Syst. 1, 1, Article 1 (January 2015), 22 pages.
DOI: 0000001.0000001

This work is partially funded by the Major program of Shanghai Science and Technology Commission (Grant No: 15511103002): Research on Mobile Smart Device Application Security Testing and Evaluating,2015.6.30-2017.6.30.
Authors' addresses: Junliang Shu, Shanghai Jiao Tong University,800 Dongchuan RD. Minhang District, Shanghai, China;email: s.junliang@gmail.com;Yuanyuan Zhang, Shanghai Jiao Tong University,800 Dongchuan RD. Minhang District, Shanghai, China;email: yyjess@sjtu.edu.cn;Juanru Li, Shanghai Jiao Tong University,800 Dongchuan RD. Minhang District, Shanghai, China;email: romangoliard@gmail.com;Bodong Li, Shanghai Jiao Tong University,800 Dongchuan RD. Minhang District, Shanghai, China;email: uchihal@sjtu.edu.cn;Dawu Gu, Shanghai Jiao Tong University,800 Dongchuan RD. Minhang District, Shanghai, China;email: dwgu@sjtu.edu.cn.
This work is submitting to the Special issue on "Embedded Device Forensics and Security: State of the Art Advances"

DOI: 0000001.0000001

1. INTRODUCTION

The popularity of smart mobile computing platforms such as Android device has changed the way people process the information. Developers have built myriads attractive and innovative applications to add convenience and fun to people's lives. They also store quite an amount of sensitive data from those service, such as camera, telephony and GPS, on the device. With malicious intention, the attackers might retrieve valuable information. For instance, mobile social network apps help to get connected with friends, online banking service help to keep track of the financial status, mobile health apps help to manage private information about diet, medication adherence, stress, smoking cessation, parental and infant care, photo shooting anywhere anytime and get synchronous on Cloud. All of these build up a digital figure of a people in the new age of Internet. Therefore, plenty of privacy information falls in the hand of the smart mobile devices [Azfar et al. 2016a; Azfar et al. 2016b; Azfar et al. 2015], which are not considered as the perfect security vessel of sensitive information.

Being the most popular mobile operating system, Android has long been a prime source of criticism on privacy leak. The major issue is how Android system and applications manipulate the data, such as when and where the data has been read, modified or transmitted. Studies in [Heuser et al. 2014], [Bugiel et al. 2013], [Xu et al. 2012], [Jeon et al. 2012], [Enck et al. 2014], [Arzt et al. 2014], [Backes et al. 2013], [Wu et al. 2014] have discussed different aspects, including hardware characteristics and operating system features, that would directly cause the failure in secure data operations that leads to the privacy leak. Yet seldom do they notice that data remanence after insecure deletion could also be an inconvenient threat since Android does not provide enough clearness of how third-party applications process user data stored on the mobile device.

For example, secure deletion on flash memory is well studied [Wei et al. 2011], [Reardon et al. 2012], [Reardon et al. 2013], The safety of the data is not well protected by the underlying processing mechanism of the operating system especially Android. Data remanence caused by improper but stealthy data deletion behavior of the Android system is worth studying to guide a more secure data erasing.

The major work of this paper is twofold. The first part of our work discusses the data remanence caused by some design flaws of Android system. One of the main reasons that leads to data remanence is the ignorance of Android on data deletion. Although data remanence and its recovery have been studied in many previous researches [Kung 1993], [Bauer and Priyantha 2001], [Quick and Choo 2013a], [Quick and Choo 2013b], Android security mechanism still does not consider this issue even in the latest version. We found that Android 6.0 and *ext4* filesystem are still not able to provide secure deletion. Another issue is the fragmentation of the Android OS. Android devices come in vastly different performance levels and screen sizes. Further, there are various different versions of Android currently running on those devices that intensifies the fragmentation. A myriad versions of Android OS introduce various implementations, e.g. the way to manipulate embedded SD card, unlock the bootloader, modify the *recovery* subsystem, etc. Under certain circumstances, the proprietary implementations contradict the privacy protection requirements, and further initial the no perceptible privacy residues.

Does it mean that using other files to overwrite the original files would guarantee the secure deletion? According to our data deletion latency experiment, the answer is 'No.' By tracking the data remanence on the Android devices of several test volunteers, after more than 2 months usage of the phone, the observed data remanence has over 40% chance of staying intact.

After thoroughly studying on exhibition of the data deletion latency and the impact of this underlying file system design flaw, we construct several attack contexts to evaluate the related impact. Previous researches mainly focus on physical layer factors and concern less about the system level issues. We argue that it is hard to acquire a really secure data deletion mechanism without considering system level issues carefully. We study the design policy and data operation interfaces related to three aspects of privacy erasure on the major version of the Android OS with five mainstream Android devices.

Moreover, for better evaluate the remanence of the deletion operation, we propose *DataRaider*, a file carving tool for recovering SQLite files from crumbs. Then, the ethical attacks have successfully recovered over 80% of the data from the data remain on the device. It allows us to obtain deleted emails, WeChat chat logs, and even allows impersonation attacks on an online payment account. We also analyze how much information could be retrieved after a vulnerable *factory reset* operation. The results indicate that most of the devices are of high privacy disclosure risk due to the deployment of third-party *Recovery* system. Approximately, 99% of the original data has remained after an improper data erasing process, which is considered as a significant failure in privacy erasure operation.

The contributions of this paper include:

(1) We first present the data operations in several system layers including the flash memory, the file systems and the *Recovery* subsystem. We analysis the vulnerabilities or design flaws which cause the data deletion deficiency, respectively. As we have discovered, the flash memory attempts to prolong its lifetime by balancing the write operations on more units. Besides, the *TRIM* command in some Android versions worsens the situation by ill implementation on deleting the data from the flash memory. Both of them guarantee longer data remaining time and higher probability of its recovery. We also discuss the partition clearing failure caused by careless implementation of file deletion in the *Recovery* subsystem.
(2) Based on the discoveries and analysis results, we construct a series of attacks of privacy retrieving. We mainly focus on three most common privacy data deletion scenarios: *app data clearing*, *app uninstallation* and *factory reset* operations. Our attacks successfully retrieve over 80% deleted privacy data from the first two attacks. The implementation flaw in factory reset leads to the success of our attack on recovery of the privacy data directly from the flash image. In 3 out of 4 experimental mobile phones, we have retrieved over 90% of the data from the cleared partitions.
(3) We also design and implement an advanced data remain evaluating framework based on the file carving technique. The implemented *DataRaider* is able to extract private data in raw disk image without any file system information, and the recovery rate is considerably high in the four test Android phones. According to our experiments, the performance of *Dataraider* is comparable with main stream file carving tools.

The rest of this work is organized as follows: Section 2 introduces essential background related to the data storage and the file operations; Section 3 exhibits three aspects of the system design defects which cause the insecurity of data; Section 4 we examine the current state of deletion flaws on Android and the techniques we use to retrieve sensitive information out of the data remain. Section 5 proposes a mitigation method along with suggest use and performance analysis; Section 6 describes the related work; Section 7 concludes our work.

2. STORAGE SYSTEM BACKGROUND

2.1. Linux File System

2.1.1. Ext4 File System. The ext4 file system is the most common default file system in Linux distributions like Mint, Mageia, Ubuntu, and Android. It is a *journaling* file system for Linux, developed as the successor to ext3[ext4 filesystem 2015].

The ext4 file system manages the file storage as a series of block groups and the files allocate storage space in units of *blocks*. A block is a group of sectors whose size varies between 1KiB and 64KiB. Blocks are in turn grouped into larger units called block groups. Block size is specified at mkfs time and typically is 4KiB.

To decrease performance loss, the block allocator tries to keep each file's blocks within the same group, thereby reducing seek times. With the default block size of 4KiB for instance, each group will contain 32,768 blocks, for a length of 128MiB. The number of block groups is the size of the device divided by the size of a block group[ext4 2010].

The layout of a standard block group is depicted in Table I.

Table I. Layout of a Standard Block Group of ext4

Group 0 Padding	*ext4 Super Block*	*Group Descriptors*	*Reserved GDT Blocks*
1024 bytes	1 block	many blocks	many blocks
Data Block Bitmap	*inode Bitmap*	*inode Table*	*Data Blocks*
1 block	1 block	many blocks	many more blocks

The ext4 file system has a journal that records updates to the file system metadata before update. In case of system crash, the OS reads the journal, then either reprocesses or rolls back the transactions in the journal. Example metadata structures in Table I include the directory entries that store file names and *inodes* that store file metadata. The journal contains the full block that is updated, not just the value being changed. When a new file is created, the journal should contain the updated version of the blocks containing the directory entry and the *inode*. Due to its property, the journal is very useful for recovering files even after a format operation. On file deletion, the filesystem only deletes the journal information, however, the block content still remains in the storage system. It leaves the hackers chance to recover the file.

2.1.2. Ext4_utils Security Issues. A root cause to the failure in wiping the metadata is caused by the ineffective implementation of ext4_utils. The earlier version of *ext4_utils* by default wipes the partition when performing the format operation. It only wipes the index of the files but leaves the metadata. However, it adds the explicit wipe option(on 2011-01-28) and this change is also merged into the Android Open Source Project. In detail, this changing adds a *-w* option to explicitly inform the *make_ext4fs* tool to wipe the partition before formatting. The original intention is to use the *BLKSECDISCARD* ioctl to erase the partition, so that it avoids leaving any data content. As a result, if the wipe option -w of *make_ext4fs* is ignored, the formatting will not wipe the old data on partition, as it will be deliberated in Android *Recovery* subsystem.

2.2. Android Storage System

2.2.1. Flash Memory Issue. Unlike commodity personal computer that uses magnetic storage devices like hard drives, Android smartphones and tablets mainly store data on the SoC with eMMC(embedded Multi-Media Controller) and the Solid State Driver as the internal memory [Memory 2014a]. The type of the internal memory is primarily the NAND type flash memory [Memory 2014b]. One characteristic of NAND flash is that its I/O interface does not provide a random-access external address bus. Instead,

data must be read on a block-wise basis, with typical block sizes of hundreds to thousands of bits. Even if the erased data is part of a block, the entire block should be overwritten. As a result, for a NAND flash memory, the deleted data is not thoroughly erased right away but labeled as *unused* by system. This characteristic, however, leads to the data remain obviously. In the only scenario where the whole block has been overwritten, the goal of thorough data erasure can be fulfilled.

The flash memory has a finite number of program-erase cycles, as most commercially available flash products are guaranteed to withstand around 100,000 P/E cycles before the wear begins to deteriorate the integrity of the storage. The corresponding data processing mechanism of the operating system must consider its physical characteristic and the security-performance tradeoff. Experiment results in [Reardon et al. 2012] revealed that, the deleted data will remain in the flash memory until it's overwritten by other operations. It implies a relatively longer residence time of the privacy data. The residence time is increasing with the size of the memory and decreasing with the frequency of usage of the memory.

Another issue is that even if the data filling or rewriting is fulfilled, it should be carefully deployed to ensure the effectiveness. Unlike traditional hard disk, flash memory has a unique data writing management characteristic, which prevents file system from employing an in-place overwriting operation. If the file system commands the storage medium to overwrite an existing data block, the on-board controller of the memory may write the data into a new place while labeling the old data block as an *unused* one. It implies that, the data remain issue is transparent to the upper layers in Android.

2.2.2. Android File System Partition Specification. Like a normal hard disk drive, the internal storage medium of an Android device can be partitioned. The internal memory usually consists of the following partitions:

— The */system* partition contains the entire Android OS, other than the kernel and the ramdisk. It includes the Android GUI and all the system applications that come pre-installed on the device. This partition is by default mounted as read-only for no data modification is required during the runtime.
— The */recovery* partition holds a second bootable Linux system known as *Recovery* system. The *Recovery* subsystem can be seen as a rescue system allowing basic operations on the device without booting into full Android. If the device enters the recovery mode, the *Recovery* system is activated for performing advanced recovery and maintenance operations such as OTA update [Wiki 2015].
— The */data* partition, also known as */userdata* partition, contains the user's data such as contacts, SMS, settings and all android applications installed. If it is erased, the Android system will return back to factory settings. The most valuable privacy information of the users usually resides in this partition, which makes it the preferred attack object.
— In addition, the */sdcard* partition is special for Android device. Oftentimes it is not on the internal memory of the device but rather the external SD card (However currently manufacturer integrates *sdcard* into the main internal memory for the performance). It is used to store non-sensitive files such as media, documents, downloaded files etc. On devices with both an internal and an external SD card, the */sdcard* partition is always used to refer to the internal SD card. For the external SD card, if present, an alternative partition is used, which differs from device to device.

Among those partitions, the */data* partition is the most valuable one for the data recovery attacking.

2.2.3. Android Data Deletion. Generally, there are two operations related to data deletion in Android.

The first one is the common file deletion operation via system interface. In this case, even when an explicit deleted file retention facility is not provided or when the user does not use it, operating systems do not actually remove the contents of a file when it is deleted unless they are aware that explicit erasure commands are required, like on a solid-state drive. Instead, they simply remove the file's entry from the file system directory, because this requires less work on a NAND flash memory and is therefore faster, and the contents of the file-the actual data-remain on the storage medium. The data will remain there until the operating system reuses the space for new data.

Since Android 4.3, the operating system issues the *TRIM* command to let the drive know no longer maintain the deleted data [Bell and Boddington 2010]. *TRIM* is a garbage collection feature that helps negate write performance reduction in flash-based storage devices, *TRIM* negates this by marking deleted data for the SSD controller to erase while the device is idling, making sure the cells holding deleted data are erased and ready to be written on. Figure 1 shows the process of data deletion with *TRIM* feature activated.

With *TRIM* feature the Android OS informs the device which sectors are no longer used (do not contain valid data anymore) and that device does not need to keep data content. Thus the TRIM command helps to erase data more thoroughly due to its low-level property. However, most of the current devices are not updated to support this new property and thus the data remains may still exist. Moreover, a side effect of the *TRIM* feature is that it may introduce observable data deletion latency: the hardware controller may not immediately recycle the data block.

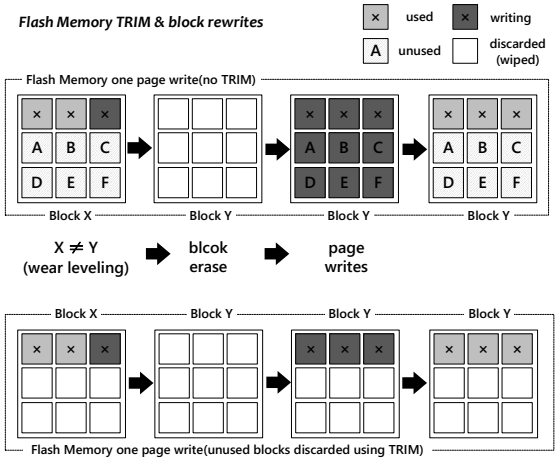

Fig. 1. The TRIM feature

The second operation is the *factory reset* operation for Android. Before an Android device is re-used, the owner would like to clear the storage to prevent accidental disclosure of confidential data to the succeeding user. According to our study on the *Recovery* subsystem, see Section 4, doing a *factory reset* is to reset the device to its initial state including removing all the data from the */data* and */cache* partition. As mentioned above, the */data* partition stores almost all the sensitive information related to user's

privacy. This operation triggers a data wiping against the entire partition that contains most of the sensitive information of the user. Specifically, the *factory reset* generally asks the storage medium to use its low-level data wiping function(built-in ATA command) to guarantee a secure data deletion.

3. ANDROID DATA ERASING FLAWS

We notice that the protection of privacy should cover the lifetime of the data, which begins from data generation, and extends to data transformation, calculation, transmission, storing, till its deletion. The failure on any stage would cause unexpected privacy leak. To avoid this, protection mechanism should be imposed throughout these stages. It should be mandatory for the end users to assure that the private data on device is under protection, and it is thoroughly and securely erased when a file delete operation takes place, even if the attackers have the capability of recovering information from metadata.

During its lifetime, the privacy data could be erased by system's data deletion operation in four scenarios: 1) generic file deletion, 2) app data clearing, 3) app uninstallation, 4) partition clearing. Except generic file deletion which is easily observed, the other three operations is out of app's control. We have found failures in such data erasure demand happen whenever the users intend to clear confidential files, uninstall high-assurance apps and *factory reset* their devices. And in the following, corresponding design flaws of these operations are thoroughly discussed.

3.1. Obscure File Deletion Flaw

The file deletion issue of flash memory is quite simple and well studied: if file deleting operation of the system has not thoroughly erased the data from the flash memory, the content can be partially or fully retrieved. However, people often focus on generic data deletion and recovery (e.g., recovering data on sdcard of the device). While this kind of insecure data deletion is well observed and can be enhanced, on Android platform, privileged data's deletion is less discussed, which is more obscure and may lead to even more sensitive private data's leak.

According to the design specification, Android app stores its privileged data in */data* partition and the data is protected by Android's sandbox mechanism. The mutual access to this part of data between applications is restricted thus it gives the users an incorrect impression that the applications and the involved data are under protection. However, the deletion operation to this part of data is often obscure. While the sandbox protects the data at runtime, it does not consider the secure deletion when a temporary file or the application reaches its completion. The ways applications delete files and system uninstall applications both lead to insecure file deletion. Yet the application itself is seldom aware of this data disposing process and could not control the deletion. In this situation, the obscurity of this process determines that the privileged data is neither securely disposed nor properly encrypted. Thus data remanence happens in a stealthy way.

More seriously, the prerequisite of a secure sandbox is that the root privilege has not been overwhelmed, or else the sandbox cannot withstand any data theft intents from outside the application, and the attacker is able to visit all the data in the memory. The fact is, for many Android devices, the root permission is very easy to be obtained [Rooting 2015], [Android 2015]. If a device has been rooted, the */data* partition can be dumped and attacker could restore the data from the remanence without even implement a physical access to the storage medium.

Erasing the privacy data is expected to securely erase the files from file system. We argue that the obscurity in app data deletion process controlled by Android may cause the leak of sensitive data. Attacker can easily recover all the data remaining

on the disk by applying the common disk recovery approaches [Forensics 2014]. With the help of these recovered data, attackers can implement various attack like grab sensitive information, obtaining decrypt secret data, etc. In the following, two ethic attacks are performed to verify our analysis.

We have studied two typical obscure file deletion scenarios in which the privacy leak usually happens: 1) app data clearing, and 2) app uninstallation.

3.1.1. Insecure App Data Clearing. The first obscure data deletion operation is Android's "Clear data" function for each app. This function is used to reset an app to its initial state and clear an app's privileged data that contains credential information such as browser history, cache, application login token and data encryption keys heavily related to user's privacy. In detail, Android system uses the `File.delete()` API for app data clearing. This API does not consider the secure data deletion requirement and is insecure in data clearing. We locate the corresponding source code in Android Open Source Project. After checking Android AOSP source code from 4.0.4 to 6.0.0, we figure out the invoking chain of the `File.delete()` function:

```
initiateClearUserData()->
    ... ->
        clearUserData()->
            do_rm_user_data()->
                    delete_user_data()->
                        delete_dir_contents()->
                            unlinkat()
```

The `File.delete()` interface will invoke `remove()` function in Linux C Library, which then invokes `unlink()` and `rmdir()` syscalls for file deletion operation. Neither of the Linux syscall does additional thorough file erasing operation. Therefore, this data clearing process does not meet the demand of secure erasure. While the misunderstanding of security in `File.delete()` misleads users to believe that the privilege data has been thoroughly erased, it results in an insecure data deletion operations on Android system.

3.1.2. Insecure App Uninstallation. Another security concern in obscure file deletion is app uninstallation. The most common security concern for an application on Android is whether the data it saves on the device is accessible to illegal access. By default, files created on the internal storage(more specifically, */data/data/pacakgeName* directory) are accessible only to the owner's app. Because Android implements this isolation, most applications tend to store sensitive data such as database using internal storage. These sensitive data, if retrieved, can be used to construct a similar context and forge the identity of app's user.

During the app uninstallation, Android system will not only removes the app's executable, but also delete all of the corresponding data of the app. Similar to data clearing function, the default app uninstallation may also perform insecure data deletion and the privacy may still be leaked.

The Android OS performs app uninstallation with a relatively complex operation sequence. We check the latest Android AOSP source code to confirm if the uninstallation is vulnerable when the system still adopts the insecure data deletion. We found that the app uninstallation triggers `unlinkat()` syscall, which operates in exactly the same way as either `unlink()` or `rmdir()`:

```
deletePackage()->
    deletePackageAsUser()->
        deletePackageX()->
```

```
deletePackageLI()->
    removePackageDataLI()->
        removeDataDirsLI()->
            remove()->
                uninstall()->
                    _delete_dir_contents()
```

Again, this system-level app privacy erasure operation does not consider the risk of insecure data deletion and is generally vulnerable to any common data recovering based attack.

3.2. Factory Reset Flaw

Factory reset is a functionality of the Android OS to clean all the metadata from the */data* and */cache* partitions and thus reset the device to its initial state. It is designed to be the last line of defense for data cleansing, taking account of the existing file deletion method in the underlying file system being quite incapable. However, due to its capability of flash memory manipulation before booting the Android OS, the *factory reset* operation is not implemented by the Android OS. Instead, it is performed by the device's *Recovery* subsystem. The *Recovery* subsystem is an Android-based lightweight runtime environment parallel to the Android OS. The main functionalities of *Recovery* subsystem include OTA system updating, factory reset, etc.

For better user experience or installing other Android distribution versions, end users would like to modify the original *Recovery* subsystem. Currently, the mostly popular alternative *Recovery* subsystems are CWM [CWM 2015] and TWRP [TWRP 2015]. We have intensively studied both of these *Recovery* subsystems, and found neither of them provide reliable privacy erasure mechanism, which means the *factory reset* cannot thoroughly erase all the data on the flash memory and the SD card (if there is any). Official *Recovery* subsystem makes use of the interface provided by *ext4_utils* library to wipe the */data* partition, while in third-party *Recovery*, such as CWM and TWRP, *factory reset* command is implemented in their own way. They first parse the */sdcard* partition to check whether it is a virtualized partition part of */data* partition. If so, the *Recovery* subsystem executes `rm -rf` command on each sub-directory in */data* partition except the sub-directory linked to virtual */sdcard* partition. Notice that the file deletion on NAND flash memory is not a reliable data deletion operation. The `rm` only delete the file index but not the metadata. Therefore, after the *factory reset* operation, the metadata are still remained on */data* partition.

After analyzed the source code of the third-party *Recovery* subsystems CWM and TWRP on five devices, we found that this flaw exists in all the released versions. We further tested devices with CWM and TWRP *Recovery*, and found out that if a device equips with an external removable SD card, the vulnerability can be eliminated.

Unfortunately, not only the third-party *Recovery* subsystems are facing the unsafe partition deletion. As we have studied, the original *Recovery* subsystems in Android 4.3 and earlier versions are also suffering from the same vulnerability. The applying of *TRIM* or some other type of partition clearing implementations are not able to erase the data securely so as to leave the chance to recover the data from the remanence.

4. EXPERIMENT

4.1. Data Remanence Experiment

To validate our analysis and reveal the potential hazards may caused by data erasing flaws, we perform a series of experiments on mainstream Android devices and observe the data erasure situation.

Table II. Data remained after Clear data operation on Sony Lt28H

File Set Size	Data Remanence Rate
85.64MB	100%
150.63MB	100%
400.30MB	100%

4.1.1. Data Clearing. First we evaluate the data remanence rate after normal "clear data" operation on Android. The experiment sets up on a Sony Lt28h mobile phone with Android 4.1.2. We first select 100 files of various types in some app's privileged sandbox environment, and then perform "clear data" operation to erase those files on the internal storage of Sony Lt28h. After the clearing, the raw memory image of the internal storage is dumped to perform common file recovering for the 100 chosen files, and the file recovering rate is calculated. We repeat this deletion operation three times with different files of different apps, the results are shown in Table II. As the results show, if the privileged data is only cleared by the "clear data" operation, it is actually not effectively erased at all.

The second experiment we performed uses a Galaxy Nexus 4 mobile phone with Android 4.4.2, which belongs to one of our colleagues for daily use. As this device is usually used as a mobile terminal to browse web pages with Chrome, the owner has already cleared the relevant data to protect his privacy before contributing this device to our experiment. We first dump the raw disk image of this device's internal storage, then using state-of-the-art data recovering tools such like *Recuva*,*Extundelete* and *UFS explorer* to extract as many files as we can. With the help of a normal Chrome application on another Android device as template, we could pick up those files belong to Chrome from the recovered files. Finally, 275 files of Chrome are picked up and some manual analyses are employed to extract a large number of privacy data such as browsed images, videos, browsing history and cookies.

In conclusion, due to the obscurity of data clearing function of Android, the insecure data deletion operation does not protect the owner of device from avoiding privacy leakage, which may jeopardize the device owner seriously in real life.

4.1.2. App Uninstallation. To evaluate the app uninstallation vulnerability, we perform common app uninstallation procedures on Android device and observe the data erasure situation. We evaluate the app uninstallation vulnerability using two devices. The first device we choose is a Sony Lt28h mobile phone with Android 4.1.2 (without TRIM feature). We would like to quantitatively evaluate the data remanence rate of app uninstallation when normal app uninstallation operation is performed. We choose five typical Android apps include *WeChat*, *QQ*, *MicroBlog*, *FaceBook* and *SnapChat* to test. For each app, we first install it and use it for a time. Then, we record every file in the app's internal storage directory. Finally, we uninstall the app, reboot the operating system, and then retrieves the raw disk image of the internal */data* partition. After acquiring the raw disk image, we conduct common file recovering approach to extract as many files as possible, and calculate the file recovering rate. For each app we repeat the experiment for three times. We also perform one test with operating five apps concurrently. The results are shown in Table III. As the results show, almost 80% of the data are still remained.

Table III. Data remanence rate after app uninstallation on Sony Lt28h

Exp	Wechat	QQ	Microblog	Fackbook	Snapchat	All
#1	86/110	184/190	64/79	65/70	23/25	313/399
#2	79/79	160/160	73/78	66/70	13/15	358/426
#3	80/80	149/156	40/69	66/70	15/15	334/396

The second device we choose is a Samsung Galaxy S5 mobile phone with Android 4.4.2 with TRIM feature enabled. The tested device is from a common user and is in use for a long time. Before testing, all of the apps on this device are uninstalled. We perform the same attack on this mobile phone and found that even with the TRIM feature activated, the dumped raw disk image of */data* partition still contains a huge amount of sensitive data. In particular, we aim at attacking the *WeChat* app, which is a very popular app in mainland China with more than four hundred million users and supports not only online communication but also online payment service, on this device. From the dumped raw disk image we recover 1214 files from */data/data/-com.tencent.mm* directory. These data are crucial as both identify authentication token and message database. We transfer these data to another device with a *WeChat* app just installed and successful forge the identify of the original user. As Figure 2 shows, not only can attacker retrieve the chat record and friends list information, but he can also observe the online transaction information of the app.

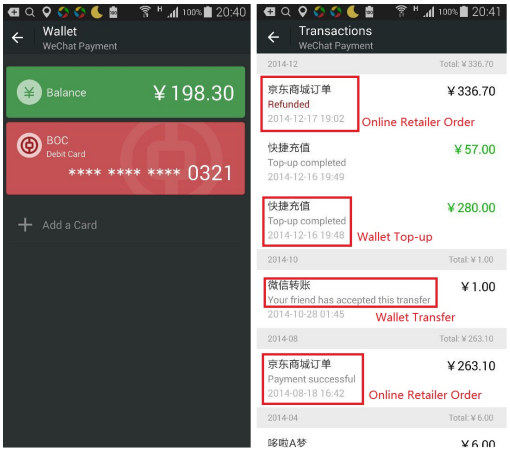

Fig. 2. Attack on WeChat app

4.1.3. Factory Reset. Data remaining after *factory reset* is a widely existing threat for two most prevailing third-party *Recovery* subsystems, and once attackers try to retrieve data from any secondhand device, over 90% of the data blocks will be recovered for the next step data content understanding. Given the metadata erasing flaw, it is not difficult to construct a privacy retrieving attack. We test the metadata recovery attack on five mainstream Android devices including Nexus4, Nexus7, Galaxy S3, Motorola MT917 and Sony Lt28h. Our experiment is conducted with CWM and TWRP *Recovery* subsystems respectively. For each device-*Recovery* combination three flash memory images are dumped:

— the entire */data* partition's image, denoted as $Image_o$,
— the after third-party-*Recovery factory reset* image, denoted as $Image_w$, and
— after *Reset*, OS initiates the partition, the third image is obtained as $Image_i$.

We developed a diff tool to observe the change of the data blocks in $Image_w$ and $Image_i$ by comparing with $Image_o$. The results are listed in Table IV.

Table IV. Data remains after data wiping(in $Image_w$) and after reboot(in $Image_w$) for various devices(Notice that TWRP does not support MT917)

Device	Recovery version	Partition size(Number of Blocks)	Sqlite block count	Image$_w$	Image$_i$
ASUS Nexus 7 (Andriod 4.4.2)	TWRP 2.6.3.1	6.01GB(1575680)	3714	92.55%	97.10%
	CWM 6.0.4.3		2524	99.95%	99.14%
Samsung Galaxy S3 (Android 4.2.2)	TWRP 2.7.0.0	11.5GB(3022848)	8915	99.83%	99.13%
	CWM 6.0.4.6		973	99.97%	99.73%
Sony Lt28H (Android 4.4.2)	TWRP 2.6.3.0	2.00GB(524288)	2904	0.48%	8.69%
	CWM 6.0.3.0		5168	0%	0.49%
Motorola MT917(Android 4.1.2)	CWM 5.0.2.5	3.84GB(1007616)	2195	13.7%	14.88%
LG Nexus 4 (Android 4.4.2)	TWRP 2.6.3.3	13.1GB(3449600)	11227	99.74%	97.49%
	CWM 6.0.4.7		11476	99.9%	98.34%

4.2. Deletion latency on Android

Although we have discussed the flaws and root causes of file delete operations on Android, we also want to investigate how long the data remanence will exist. To further determine the remanence period of improper deleted data, we conduct a long-term experiment on deletion latency, which reflects the period between data deletion and its actual removal from the storage medium. If the deletion latency can be ignored (e.g., several hours), deleted privacy data can be expected to be secure for the lack of attack surface. However, if the deletion latency is long enough, the chance for attacker to retrieve the data is not negligible.

To measure the deletion latency of deleted data exposed to attackers, our experiment records the actual data removal period of several apps' private data on different Android devices. We choose three mainstream Android devices (Huawei Honor 4A, LG Nexus4 and LG Nexus5) as our experimental targets, and these experimental devices are sent to different testers for daily use. The information of devices and target files is listed in Table V. All three devices contain a monitoring program that records the occurrence of exact data removal event. Through a more than two months testing, the situation of how data remanence changes is summarized.

The first step of our experiment is to label the deleted files' location so that the later monitoring could be employed. Before deleting all the target files through common Android file deletion API, we directly read and record the block id and block data of each file. Then, we execute delete operation to lead all the target block marked as unused by the filesystem. Among the unused blocks, about one third of them are adjacent, while others are individually distributed on the disk. This distribution is an important feature because the size of unused disk fragments will influence the filesystem's recycling, which is demonstrated in our following testing.

We install a monitoring program with root privilege on each device to record the changing of data remanence. After the deletion operations the monitoring program start to continuously scan all target blocks per hour, checking whether a block has been actually overwritten and recording the remanence rate of all deleted data. Once the monitoring program start running, three devices are delivered to different volunteers for daily use. All volunteers are not aware of the details of experiment so as to keep their cellphone usage habits over the entire experiment time. After two months, we reclaim the experiment devices and analyze the log of monitoring program to acquire the final conclusion.

Table V. Device and file set information of deletion latency experiment

Device(Android Version)	Disk Size(Free/All)	File set size	Amount of files(blocks)	Maximal/Minimum file size
Huawei honor 4A (5.1.0)	2.5GB/8GB	42.4MB	222(10710)	25140KB/4KB
Nexus4 (4.4.2)	10.5GB/16GB	36.2MB	743(9119)	6370KB/4KB
Nexus5 (5.0.0)	4.1GB/16GB	57.5MB	336(14705)	8346KB/4KB

Figure 3 shows our experimental results that reflect the relationship between data remanence rate and time. We can see at a glance that during the experiment, the data remanence rate is continuously decreasing. However, even after two months of daily operations, **none** of the three experiment devices has entirely wiped the target deleted data. At least 40% target deleted data is still remained on each of the three devices. Particularly, the tested Nexus 4 smartphone reveals an approximate 90 % data remanence, which leads to a great possibility of data leakage. After a thorough analysis of the log data, many other observations are discovered. First, individual unused blocks are reused prior to adjacent unused blocks. According to the block allocation algorithm of *ext4* filesystem, which follows a greedy strategy, the most suitable unused area is always chosen for writing new data. For Android system, small files such as web browser logs, temporary records and configuration files are more likely to be created or modified during daily use. The operations on small files will cause the reallocation of unused fragments of small size and thus the individual unused blocks are more likely to be recycled. Second, we notice that there are some significant declines of data remanence rate during the experiment. These declines imply the allocation of some large unused areas with adjacent unused blocks. We find that it is unusual for daily use to trigger the allocation of a large file written on Android as we only observe a few such events. Thus, data in adjacent unused blocks are expected to stay longer than those in individual unused blocks.

Another important conclusion drawn from our experiment is that there is no remarkable relationship between deletion latency and disk size. Although prior research [Reardon et al. 2012] shows that the deletion latency of log-structured filesystem has positive correlation with disk size in simulation experiment. But according to Fig 3, the disk size of Nexus5 is larger than Huawei honor 4A while the deletion latency of Nexus5 is obviously shorter than Huawei honor 4A. In our experiment, we find that the actual deletion latency mainly depends on the user behavior and disk block distribution.

4.3. Recovering from incomplete data fragments

In the previous sections, we use normal file/data recovery utilities such as *dd*, *recuva* or *extundelete* to recover deleted files. The integrity of deleted data and file system metadata (e.g journal) is mandatory for normal data recovery utilities. But the result of deletion latency experiment shows that such integrity may be broken after being used for a long time. In such context, a significant amount of valuable data still remains in the incomplete data fragments.

To assess the data remain issue in such situation, we implemented *DataRaider* that adopts *file carving* technique to recovery files from crumbs. It's implemented as an automated sensitive information recovering framework for evaluating data remains when the integrity of remain data is broken.

File carving is a useful technique for recovering specific type of files from data fragments, here we choose SQLite Database file as our target because Android suggests that app uses SQLite database to store data that are frequently queried. Most of the confidential data in Android are managed with SQLite3 DBMS and are stored as *.db*

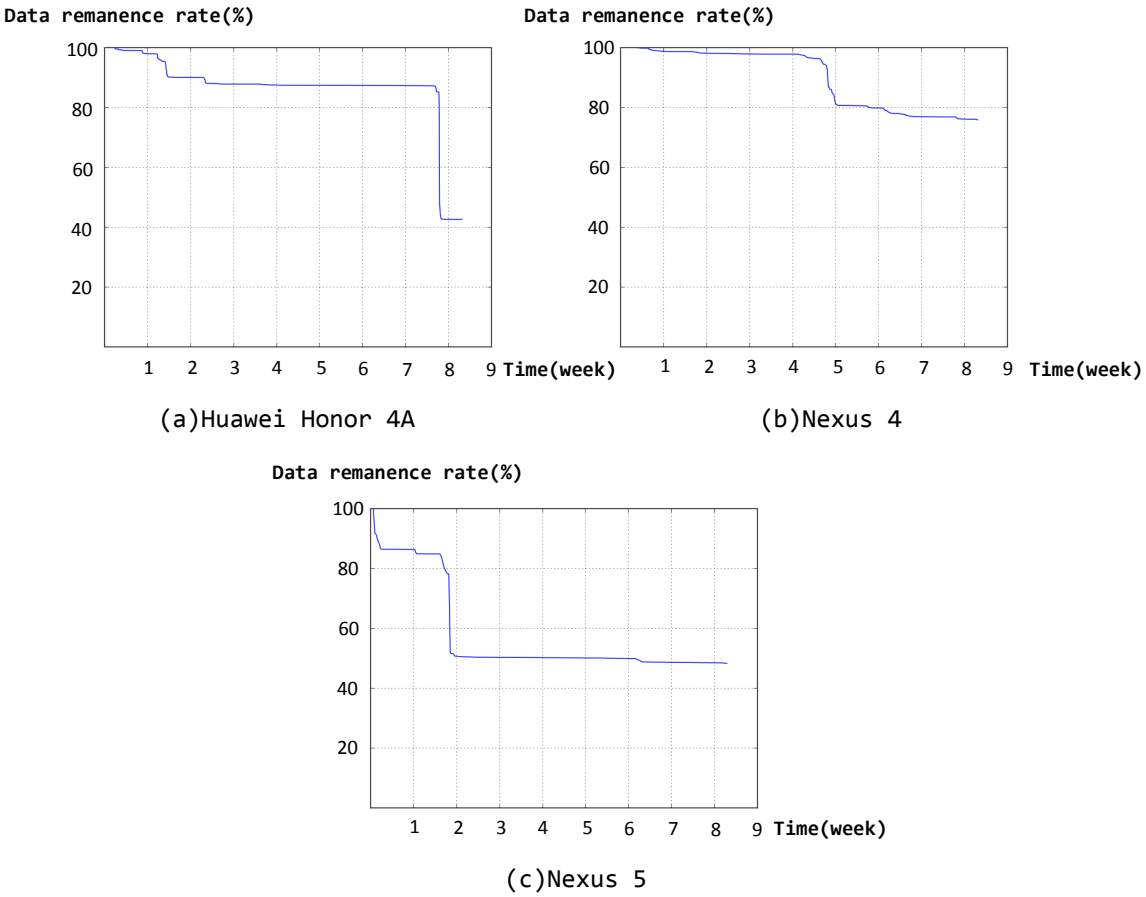

Fig. 3. Data remanence rate of deleted files

file in userdata partition, according to Android development document's recommendation. Typical databases for system applications include short message(mmssms.db), contacts information(contacts2.db), system setting(settings.db), keychain(grants.db), schedule(calendar.db), etc. Except for the standard system apps, a huge amount of third-party apps also store their data using SQLite interface.

As the target of *DataRaider* is to recover deleted files the wiped devices, its recovering process focuses on reconstructing SQLite database on the *userdata* partition. The entire recovering process is shown in Figure 4.

While common ext4 data recovering techniques often rely on the journaling feature to boost the data recovering[Kim et al. 2012]. We aim at recovering remaining data without any support from the file system.

DataRaider starts with an disk image file. It is split into data blocks first and analyzes the data blocks directly.

First, it extracts data blocks from the /userdata partition. As we investigate more than 20 devices, we found that all of them adopt a 4KiB block size for the ext4 file

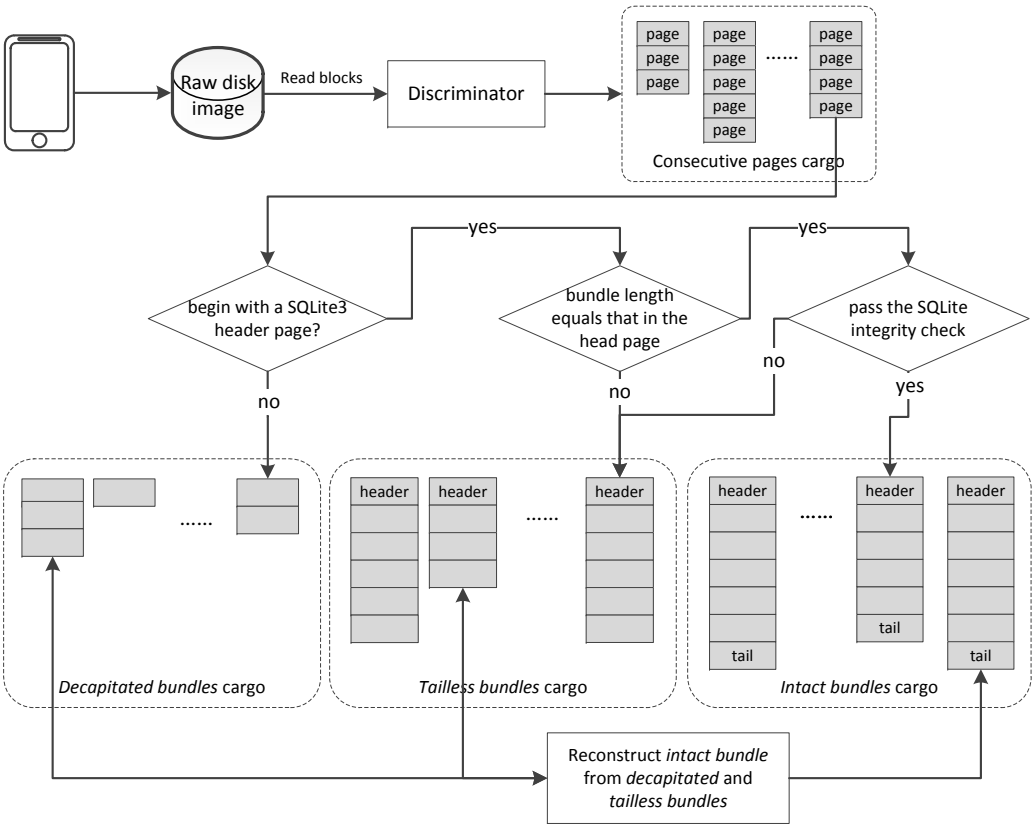

Fig. 4. Sensitive Information Extraction Process

system's choice. Moreover, we also find that more than 95% of the SQLite databases adopt the page size as same as the ext4 block size. Thus the splitting of the raw disk image separates the entire data into units of 4KiB.

Second, *DataRaider* discriminates every block to filter out potential SQLite pages. A typical SQLite3 database file consists of multiple pages. The size of one page is defined in database file's header. Generally, SQLite on Android uses 4KB size page corresponding to the ext4 file system's default block size. The page number starts from 1 and continues in one database file. The first page is the SQLite3 header page, which is crucial for a database file. Because it contains the header and the master table *sqlite_master* of this database. Except for the header page, a SQLite database can also contain other types of pages, among which the one we are interested is *b-tree* page. SQLite DBMS uses *b-tree* page to store the meta structure and the data content. A SQLite3 database can be approximately seen as a tree whose root is the header page, and most of its child nodes and leaf nodes are *b-tree* page[Format 2015]. The identification of SQLite3 data page relies on the internal structure of the SQLite3 b-tree page. A SQLite3 b-tree page stores one or multiple data records of the database. Ideally, the file will be allocated consecutive blocks[Kim and Kim 2012]. This causes data blocks of the same file to be close together. We'll use this fact to restrict where we search for deleted data. Then our discriminator will group one or multiple consecutive pages as a **bundle**. It will continually analyze the block until meeting a block that does not meet the rule. After this process, *DataRaider* stores the bundles in a *cargo* for the further file reconstruction.

The last step is to perform a SQLite database reconstruction procedure to recover sensitive information as much as possible. For bundles in the *bundle cargo*, *DataRaider* categorize them as three types: intact bundle, tailless bundle and headless bundle Intact bundle denotes to a group of consecutive blocks that is already a complete database file. Tailless bundle is a group of consecutive blocks that contains a SQLite3 header page but is not an intact database. And headless bundle denotes to a group of consecutive data pages. In the recovering process, now that an intact bundle is already a recovered database, the reconstruction of a database mainly tries to fix tailless bundle with a header page to an intact database. According to the header page contained in a tailless bundle, *DataRaider* could determine the original size of the database and calculate the the size of lacked part(we denote the size as L). Then it tries to supplement one tailless bundle with other headless bundles. *DataRaider* searches for a set of headless bundles with the total size equals to L, and tries to combine those bundles with the tailless bundle, and then verifies whether it is a valid database file. This process is iteratively employed until every headless bundle is tested.

We evaluate the functionality of *DataRaider* on three Android devices(Nexus4, Galaxy S3 and Nexus 7). We try to restore the database of SMS, contact list, keychain, system setting, etc. After a *factory reset* operation, we have used the devices for several days. Then, we put those devices into the recovery test. The results are listed in Table VI. Except for very few entries, almost all private information have been successfully restored by performing sophisticated data retrieval to a certain extent. This experiment reveals that a lot of extractable sensitive data has been left on the disk while some data blocks were missing and the integrity of data was compromised.

We evaluate the performance of *DataRaider* by comparing the time cost with a well known stream file carving tool *Foremost*[of Special Investigations et al. 2015].

Foremost is one of the first file carvers that implement sequential header to footer file carving by the file's header and its size. Carving SQLite database files is difficult for common file carving tools because of the missing footer and the size of the file. To provide a better solution, *DataRaider* adopts a customized file reconstruction stage right after discriminating related blocks from the disk image. Despite more time cost, this stage makes *DataRaider* competent to recover the SQLite database files.

We divide the whole recovering process into two stages for *DataRaider*, discrimination and reconstruction. In our experiment, *Foremost* has been applied to carve the common application file types (e.g. pdf, jpg, png, bmp, mp3, etc.) from the given disk images. It does not write any detected files back to the disk. *DataRaider* is used to carving SQLite database files without writing reconstructed files back to the disk. The experiment is implemented on a PC with Intel i3-2120 and 16GB RAM. Both *DataRaider* and *Foremost* are used to analysis a 2GB disk image for 10 times.

On average, the results show that the discrimination stage of *DataRaider* cost 65.06s and the reconstruction stage cost 42.54s respectively, and it has successfully recovered 243 SQLite database files. *Foremost* cannot recover SQLite database files. Instead, it costs 60.37s to recover 444 common application files.

The recovery process for *Foremost* and the discrimination stage of *DataRaider* have cost equivalent time. The mandatory need for both of them to parse the whole disk image cannot be neglected. Next stage for *DataRaider* brings some extra time consuming on reconstructing the SQLite database files which is in an acceptable range considering the extra functionality it provides than *Foremost*.

Our experiment shows that the overall performance of *DataRaider* is comparable with main stream file carving tools.

Table VI. Recovered data proportion of sensitive information related database on various devices

Device	Database	Origin Size	Data Recovered
Nexus4	SMS	428KB	2.8%
	Contact	1184KB	84%
	KeyChain	20KB	100%
	System Setting	88KB	100%
	Calendar	164KB	7.3%
	3rd APP (WeChat)	1112KB	100%
Galaxy S3	SMS	292KB	62%
	Contact	1024KB	72%
	KeyChain	20KB	100%
	System Setting	140KB	100%
	Calendar	388KB	47%
Nexus 7	Contact	904KB	39.8%
	KeyChain	20KB	100%
	System Setting	76KB	15.8%
	Calendar	160KB	77%

5. MITIGATION

As an open platform, Android system has various implementations inevitably. The fragmentation of Android systems have ignited discussions on many security topics. One of them is the vulnerable data wiping in Android system. As its various causes among myriads of versions, the secure data clearing has difficulty in finding an all-in-one solution. The battle between data clearing and recovering would continue.

Various techniques have been developed to resist recovery attacks. The widely known methods include overwriting based secure deletion, data encryption and physical destruction of the device. In this paper we only focus on software-assistant data deletion.

Research claim that With *TRIM* feature the Android OS can issue *TRIM* commands to wipe the entire drive in seconds [King and Vidas 2011]. Another aspect of study focuses on secure deletion, which has been studied in a variety of contexts and has also been extensively surveyed [Wei et al. 2011], [Reardon et al. 2012], [Reardon et al. 2013]. Secure deletion is the base of privacy erasure but unfortunately is very hard to be implemented correctly on most mobile devices. In [Wei et al. 2011], the authors empirically evaluate the effectiveness of erasing data from flash-based SSDs. Their conclusion is that none of the available software techniques for sanitizing individual files were effective, although the proposed attack requires laboratory data recovery techniques and equipments. Moreover, in [Reardon et al. 2012] the authors thoroughly discussed secure deletion at user-level. They propose two user-level solutions that have achieved secure deletion. In [Leom et al. 2016], the authors conducted a comparative

summary of existing approaches to realize secure flash storage deletion and idetified existing limitations with experiments.

We support the idea of data encryption as an efficient protection strategy. When the encrypted data is about to be deleted, securely disposing the encrypted key rather than wiping the data is easier to achieve an equal secure deletion. However, to fulfil robust and efficient encryption scheme may involve many issues including performance overhead and key management, for example, the Android's Full Disk Encryption scheme. Although it keeps evolving since Android 3.0, it still has been restricted by many user-level applicability and compatibility factors, and the security bound is thus reduced. We will detail the flaws of FDE in Section 6.

Lee et al. [Lee et al. 2008] has designed a NAND flash file system with a secure deletion functionality. This method uses the idea of encryption.It requests that all the keys of a specific file to be stored on the same block. Therefore, only one erasing operation on the key files is required to securely delete the data files. However, their work only support YAFFS file system yet current Android devices generally adopt ext4 file system. In [Wang et al. 2012], the authors present a FUSE (Filesystem in USErspace) encryption file system for Android. The proposed encrypted file system introduces about 20 times performance overhead than normal system.

The trusted SQLite database storage [Luo et al. 2013] is another option when performance is a major concern. Although, we have to remember that the lightweight SQLite database encryption solution is less secure than full disk encryption. At the system booting phase, it is faster to load an encrypted database rather than an encrypted partition. And these sensitive files are decrypted on-the-fly only when legitimate users are unlocking and using the phone. Besides, to reload the encrypted database is of little cost thus each time the screen is locked, system could erase decrypted data in memory directly and no sensitive data is stored. That means even if the attacker directly access to the remaining data, he still cannot get any useful data. Thus the sensitive files are well protected.

Another possible mitigation can be an application level security enhancement. One of the implementation can be hooking the key libC functions related to disk I/O. Wrapped by some cryptographic operations, the data writes to the disk in cipher text form. It would be satisfying in both security and efficiency purpose.

Given the present situation of Android OS fragmentation, this method provides transparent encryption that adapts to almost all the Android versions without interrupting the device user. Comparing to system modification scheme or upgrading the OS to a higher version such as Android Jelly Bean or Lollipop, a more realistic solution users would prefer is a simple way to deploy security enhancement on their applications.

6. RELATED WORK

6.1. Data Recovering

Many data recovering methods designed for PCs have been applied on mobile devices, such as file recovery, file carving[Kim et al. 2012], [extundelete 2013], [Recuva 2015]. While the ext4 file system provide many improved features, one of its features, the journaling feature, can be leveraged to boost the data recovering. Journaling is a technique employed by many file systems in crash recovery situations. Tools such as *extundelete* can recover files immediately after parsing the ext4 filesystem's journal, usually within a few minutes. The contents of an inode can be recovered by searching the file system's journal for an old copy of that inode. Then, that information can be used to determine the file's location within the file system [Fairbanks et al. 2010].

JDForensic [Kim et al. 2012] is a tool aiming to analyze journal log area in ext4 file system and extract journal log data to recover deleted data and analyze user actions. Extundelete [extundelete 2013] is another popular open-source tool focusing on recovering file from disk with ext3 and ext4 file system. It is able to recover the contents of an inode by searching the file system's journal for an old copy of that inode. However, if the supporting journal information is broken or missing, both tools are not able to recover any files.

Close to our research, there are many studies focusing on directly data recovery using both forensics and physical access. Our work is generally based on these studies but we focus on the data erasure failure and related concrete threats rather than data remanence and extraction. A direct approach of data extraction is to read the internal memory through the boundary-scan (JTAG) test pins. [Breeuwsma et al. 2006] introduces the details of using a JTAG test access port to access memory chips directly. This approach is feasible for the situation that the phone is locked and cannot be accessed via USB cable. The disadvantage is that it needs to disassemble the mobile device and it requires specific analyzing equipment and knowledge to fulfil.

Another acquisition of Android memory images proposed by FROST [Müller and Spreitzenbarth 2013] is against full disk encryption. It focuses on breaking disk encryption of Galaxy Nexus devices. Again, it requires complicated experimental environment to fulfil the attack.

There are also researches aim at specific resources on Android. Do and Choo [Do et al. 2015] propose an adversary model to facilitate forensic investigations of mobile devices in 2015. An evidence collection and analysis methodology for Android devices is provided in this paper. With the help of this methodology, Do and Choo extract various information of forensic interest in both the external and internal storage of six chosen mobile devices.

Choo et al. [Leom et al. 2015] provide a study about the thumbnails recovery in Android. They examine and describe the various thumbnail sources in Android devices and propose a methodology for thumbnail collection and analysis from Android devices.

Immanuel and Choo [Immanuel et al. 2015] study the diversity of cache formats on Android and give out a taxonomy of them based on app usage. Using this taxonomy as a base, they propose a systematic process, known as the Android Cache Forensic Process, to forensically classify, extract and analyze Android caches.

6.2. FDE Flaw

Since Android 3.0, Google has provided the *full-disk encryption(FDE)* as an optional security service. FDE derives from the *dm-crypt* feature in Linux. The user can choose to activate FDE service from the system settings.

Since the Android OS version 5.0 (Lollipop), full-disk encryption mechanism has been improved to counter brute-force attack and has enabled full crypto features by default. Being similar to iOS, it adopts hardware-assist encryption and implements per-app encryption as well. Google has claimed that Lollipop would introduce hardware support to enhance the security strength and boost cryptographic process speed.

However, a weaker full-disk encryption is still prevalent in Android app markets, which would lead to easy recovery of the data. The early versions such as Jelly Bean(version 4.1 - 4.3.1) and Kitkat(version 4.4 - 4.4.4) are still in use. Till May 2014, Android Jelly Bean is still the dominant OS version in the Android ecosystem. Android Jelly Bean provides a weaker full-disk encryption that would make brutal force attack on the encryption keys much easier [disk encryption 2014].

First, due to the lack of hardware encryption chip's support, the master key for FDE is generated from Android lockscreen passcode. The encryption parameters for calculating the secret keys are stored on a special structure in a disk partition called *crypto*

footer. Before Android 5.0, the attacker is capable of launching a brute-force attack against the PIN by analyzing the specific information in the structure to obtain the encryption key. If the encryption key is acquired, data remanence is again the source of privacy leaking. According to [disk encryption 2014], for FDE scheme before Android 4.4, the popular brute-force attacking tool *hashcat* can achieve more than 20,000 PBKDF2 hashes per second on a notebook with NVIDIA GeForce 730M, and recovers a 6 digit PIN in than 10 seconds. On the same hardware, a 6-letter (lowercase only) password takes about 4 hours. Considering the custom of users (less people would like to set a complex lockscreen passcode and repeatedly input it every time), it is realistic to recover the master key of FDE in a reasonable time. For Android 4.4.x, the complexity of brute-force attack is increased but still feasible. And this only restricts the attacker to recovering passcode within a relative longer time.

Second, postulating the using of a third-party *Recovery* subsystem, FDE would face an even worse situation. As we have deliberated in the previous sections, the ineffective implementation `rm` command in some *Recovery* system have caused the incapable of wiping the data content on the disk. In this case, the encrypted metadata has been kept intact. With the help of the encryption key retrieving attack, the complete content on the disk can be revealed.

In a word, FDE scheme before Android 5.0 is not so effective as it is expected. A more robust data protection requires better encryption scheme to thoroughly eliminate the threat of data remanence.

7. CONCLUSIONS

In this work, we analysis several defects in Android system such as data remain after data clearing, app uninstallation and even *factory reset*. To prevent privacy leak caused by recovering the data remain, the data must be erased in three aspects: generic files deletion, app uninstallation, and metadata erasure. Due to various reasons, privacy erasure fails in many ways. We perform a systematical study on the impact of the imperfect data erasure operations, and demonstrate how the attackers manage to recover sensitive information in three scenarios: generic file deletion, app uninstallation and factory reset. Essentially, the primary cause of erasure failure is the inefficient implementation on file operations and the third-party *Recovery* system implementation flaws.

As other successful operating systems, Android provide agility and convenience to the mobile smartphone users, and it either consider enough security enhancement at the design phase. Confidential information exist in Android system as generic files, information residing in an app, or metadata on flash memory. In this work, we exhibit how it has been ignored and caused serious data remanence in Android systems. Data wiping is more important under mobile environment than traditional computing context. Besides hardware-assist security enhancement such as TrustZone, we hope to propose some OS level data wiping or protection mechanism for the Android ecosystem in future.

REFERENCES

Rooting Android. 2015. Rooting (Android OS). http://en.wikipedia.org/wiki/Rooting_(Android_OS). (2015).

Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 259–269. DOI:http://dx.doi.org/10.1145/2594291.2594299

Abdullah Azfar, Kim-Kwang Raymond Choo, and Lin Liu. 2015. Forensic taxonomy of popular Android mHealth apps. *arXiv preprint arXiv:1505.02905* (2015).

Abdullah Azfar, Kim-Kwang Raymond Choo, and Lin Liu. 2016a. An android communication app forensic taxonomy. *Journal of Forensic Sciences* 61, 5 (2016), 1337–1350.

Abdullah Azfar, Kim-Kwang Raymond Choo, and Lin Liu. 2016b. Android mobile VoIP apps: A survey and examination of their security and privacy. *Electronic Commerce Research* 16, 1 (2016), 73–111.

Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. 2013. AppGuard: Enforcing User Requirements on Android Apps. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'13)*. Springer-Verlag, Berlin, Heidelberg, 543–548. DOI:http://dx.doi.org/10.1007/978-3-642-36742-7_39

Steven Bauer and Nissanka Bodhi Priyantha. 2001. Secure Data Deletion for Linux File Systems.. In *Usenix Security Symposium*, Vol. 174.

Graeme B Bell and Richard Boddington. 2010. Solid state drives: the beginning of the end for current practice in digital forensic recovery? *Journal of Digital Forensics, Security and Law* 5, 3 (2010), 1–20.

Ing Breeuwsma and others. 2006. Forensic imaging of embedded systems using JTAG (boundary-scan). *digital investigation* 3, 1 (2006), 32–42.

Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. 2013. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies.. In *Usenix security*. 131–146.

CWM. 2015. ClockworkMod Recovery. https://www.clockworkmod.com/rommanager. (2015).

Ming Di Leom, Christian Javier DOrazio, Gaye Deegan, and Kim-Kwang Raymond Choo. 2015. Forensic collection and analysis of thumbnails in android. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, Vol. 1. IEEE, 1059–1066.

Revisiting Android disk encryption. 2014. Revisiting Android disk encryption. http://nelenkov.blogspot.com/2014/10/revisiting-android-disk-encryption.html. (2014).

Quang Do, Ben Martini, and Kim-Kwang Raymond Choo. 2015. A Forensically Sound Adversary Model for Mobile Devices. *PloS one* 10, 9 (2015), e0138449.

William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *ACM Trans. Comput. Syst.* 32, 2, Article 5 (June 2014), 29 pages. DOI:http://dx.doi.org/10.1145/2619091

Understanding ext4. 2010. Understanding ext4. http://digital-forensics.sans.org/blog/2010/12/20/digital-forensics-understanding-ext4-part-1-extents. (2010).

ext4 filesystem. 2015. ext4 filesystem. https://ext4.wiki.kernel.org/index.php/Main_Page. (2015).

extundelete. 2013. extundelete. http://extundelete.sourceforge.net/. (2013).

Kevin D Fairbanks, Christopher P Lee, and Henry L Owen III. 2010. Forensic implications of EXT4. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*. ACM, 22.

Solid State Driver Forensics. 2014. Solid State Driver Forensics. http://www.forensicswiki.org/wiki/Solid_State_Drive_(SSD)_Forensics. (2014). Online; accessed Oct-2014.

SQLite3 File Format. 2015. SQLite3 File Format. https://www.sqlite.org/fileformat.html. (2015).

Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. 2014. ASM: A Programmable Interface for Extending Android Security. In *Proceedings of the 23rd USENIX Conference on Security Symposium (SEC'14)*. USENIX Association, Berkeley, CA, USA, 1005–1019. http://dl.acm.org/citation.cfm?id=2671225.2671289

Felix Immanuel, Ben Martini, and Kim-Kwang Raymond Choo. 2015. Android Cache Taxonomy and Forensic Process. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, Vol. 1. IEEE, 1094–1101.

Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. 2012. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 3–14.

Dohyun Kim, Jungheum Park, Keun-gi Lee, and Sangjin Lee. 2012. Forensic analysis of Android phone using Ext4 file system journal Log. In *Future Information Technology, Application, and Service*. Springer, 435–446.

Hyeong-Jun Kim and Jin-Soo Kim. 2012. Tuning the Ext4 Filesystem Performance for Android-Based Smartphones. In *Frontiers in Computer Education*. Springer, 745–752.

Christopher King and Timothy Vidas. 2011. Empirical analysis of solid state disk data retention when used with contemporary operating systems. *digital investigation* 8 (2011), S111–S117.

Kenneth C Kung. 1993. Secure file erasure. (Nov. 23 1993). US Patent 5,265,159.

Jaeheung Lee, Junyoung Heo, Yookun Cho, Jiman Hong, and Sung Y Shin. 2008. Secure deletion for NAND flash file system. In *Proceedings of the 2008 ACM symposium on Applied computing*. ACM, 1710–1714.

Ming Di Leom, Kim-Kwang Raymond Choo, and Ray Hunt. 2016. Remote wiping and secure deletion on mobile devices: a review. *Journal of Forensic Sciences* (2016).

Ming Di Leom, Christian Javier DOrazio, Gaye Deegan, and Kim-Kwang Raymond Choo. 2015. Forensic Collection and Analysis of Thumbnails in Android. In *Trustcom/BigDataSE/ISPA, 2015 IEEE*, Vol. 1. IEEE, 1059–1066.

Yuhao Luo, Dawu Gu, and Juanru Li. 2013. Toward active and efficient privacy protection for Android. In *Information Science and Technology (ICIST), 2013 International Conference on*. IEEE, 924–929.

Flash Memory. 2014a. Flash Memory. http://www.jedec.org/category/technology-focus-area/flash-memory-ssds-ufs-emmc. (2014). Online; accessed Oct-2014.

Flash Memory. 2014b. Flash Memory: SSDs, UFS, e.MMC. http://en.wikipedia.org/w/index.php?title=Flash_memory. (2014). Online; accessed Oct-2014.

Tilo Müller and Michael Spreitzenbarth. 2013. FROST. In *Applied Cryptography and Network Security*. Springer, 373–388.

United States Air Force Office of Special Investigations, The Center for Information Systems Security Studies, and Research. 2015. Foremost. http://foremost.sourceforge.net/. (2015).

Darren Quick and Kim-Kwang Raymond Choo. 2013a. Digital droplets: Microsoft SkyDrive forensic data remnants. *Future Generation Computer Systems* 29, 6 (2013), 1378–1394.

Darren Quick and Kim-Kwang Raymond Choo. 2013b. Forensic collection of cloud storage data: Does the act of collection result in changes to the data or its metadata? *Digital Investigation* 10, 3 (2013), 266–277.

Joel Reardon, David Basin, and Srdjan Capkun. 2013. Sok: Secure data deletion. In *Security and Privacy (SP), 2013 IEEE Symposium on*. IEEE, 301–315.

Joel Reardon, Claudio Marforio, Srdjan Capkun, and David Basin. 2012. User-level secure deletion on log-structured file systems. In *Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*. ACM, 63–64.

Recuva. 2015. Recuva. https://www.piriform.com/recuva. (2015).

Rooting. 2015. Rooting. http://forum.xda-developers.com/wiki/Root. (2015).

TWRP. 2015. Team Win Recovery Project. http://teamw.in/project/twrp2. (2015).

Zhaohui Wang, Rahul Murmuria, and Angelos Stavrou. 2012. Implementing and optimizing an encryption filesystem on android. In *Mobile Data Management (MDM), 2012 IEEE 13th International Conference on*. IEEE, 52–62.

Michael Wei, Laura M. Grupp, Frederick E. Spada, and Steven Swanson. 2011. Reliably Erasing Data from Flash-based Solid State Drives. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST'11)*. USENIX Association, Berkeley, CA, USA, 8–8. http://dl.acm.org/citation.cfm?id=1960475.1960483

Android Recovery Wiki. 2015. Android Recovery Wiki. http://forum.xda-developers.com/wiki/Recovery. (2015).

Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang. 2014. AirBag: Boosting Smartphone Resistance to Malware Infection. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2013*. http://www.internetsociety.org/doc/airbag-boosting-smartphone-resistance-malware-infection

R. Xu, H. Saıdi, and R. Anderson. 2012. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX conference on Security*.