Feng, W, Liu, C, Guo, Z, Baker, T, Wang, G, Wang, M, Cheng, B and Chen, J

MobiGyges: A Mobile Hidden Volume for Preventing Data Loss, Improving Storage Utilization, and Avoiding Device Reboot

MobiGyges: A Mobile Hidden Volume for Preventing Data Loss, Improving Storage Utilization, and Avoiding Device Reboot

Wendi Feng[a], Chuanchang Liu[a,*], Zehua Guo[b,c], Thar Baker[d],
Gang Wang[b,c], Meng Wang[a], Bo Cheng[a], and Junliang Chen[a]

[a] *Beijing University of Posts and Telecommunications, 10 Xitucheng RD, 100876, Beijing, China*
[b] *Beijing Institute of Technology, 5 Zhongguancun ST South, 100081, Beijing, China*
[c] *University of Minnesota Twin Cities, 117 Pleasant ST, 55455, Minneapolis, USA*
[d] *Liverpool John Moores University, James Parson Building, Liverpool, L3 3AF, UK*

Abstract

Sensitive data protection is essential for mobile users. Plausibly Deniable Encryption (PDE) systems provide an effective manner to protect sensitive data by hiding them on the device. However, existing PDE systems can lose data due to overriding the hidden volume, waste physical storage because of the “reserved area” used for avoiding data loss, and require device reboot when using the hidden volume. This paper presents MobiGyges, a hidden volume based mobile PDE system, to fill the gap. MobiGyges addresses the problem of data loss by restricting each storage block used only by one volume, and it improves storage utilization by eliminating the “reserved area”. MobiGyges can also avoid device reboot by mounting the hidden volume dynamically on-demand with the *Dynamic Mounting* service. Moreover, we identify two novel PDE oriented attacks, the *capacity comparison attack* and the *fill-to-full attack*. MobiGyges can defend them by jointly leveraging the *Shrunk U-disk method* and *multi-level deniability*. We implement the MobiGyges proof-of-concept system on a real mobile phone Google Nexus 6P with LineageOS 13. Experimental results show that MobiGyges prevents data loss, avoids device reboot, improves storage utilization by over 30% with acceptable performance overhead compared with current works.

Keywords: data loss preventing, hidden volume, improving storage utilization, sensitive data protection, avoiding reboot

1. Introduction

Mobile devices (e.g., smartphones) have become prevalent in recent years, especially in the era of 5G and the Internet of Things (IoTs) [1]. Hence, protecting private and sensitive data on mobile devices are very important to users [2, 3]. One existing solution is to use Full Disk Encryption (FDE) [4]. FDE uses an encrypting key to encrypt user data before storing it on a device and decrypt the data before applications using it [5]. Nonetheless, FDE is not secure because sensitive data can be compromised when the encryption key is exposed, since the encrypted data can be easily decrypted with the only encryption key.

Recent works [6–10] propose Plausibly Deniable Encryption (PDE) to enhance the security. PDE is a data protection paradigm that protects sensitive data on both stationary systems and mobile systems by providing *deniability* for the sensitive data. Deniability means that sensitive data owner can deny the existence of the data. Modern PDE systems use the *hidden volume mechanism* to implement the deniability. The hidden volume based PDE system stores the sensitive data on the hidden volume, yet the hidden volume itself is concealed inside the device. Logically, the storage space on the hidden volume based PDE systems can be divided into *hidden volumes* and an *outer volume*. The outer volume is visible to all users for daily purposes and will be used automatically as the system starts up, while hidden volumes are concealed in the device and store the sensitive data.

[*] Corresponding author.
Email address: lcc3265@bupt.edu.cn (Chuanchang Liu)

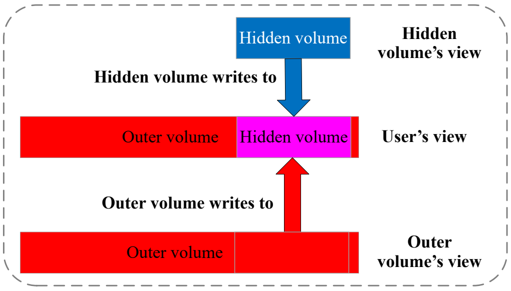

Figure 1: Data loss caused by data override. The outer volume writes data on hidden volume occupied blocks.

Such hidden volume based solutions have the following limitations:

- **Data loss.** Hidden volumes are concealed inside the outer volume [7–11], but the outer volume does not know the existence of hidden volumes. Therefore, it is likely to write data on the storage blocks that are occupied by hidden volumes. Thus, sensitive data stored on the hidden volume will be lost. Figure 1 shows an example of data overriding between the outer volume and the hidden volume. In the figure, the outer volume considers all the storage space (red space) to be usable. When the outer volume writes data on the space of the hidden volume (purple space), the data on the hidden volume will be lost.

- **Storage waste.** Studies [7–10, 12] attempt to solve the data loss problem by placing the hidden volume into a "reserved area", and the outer volume will not write data to that area. The size of the "reserved area" is bigger than the capacity of the hidden volume. As depicted in Figure 2a, in previous works, the right part (green blocks plus the blue block) of the physical volume is reserved for the hidden volume. Since the capacity of the hidden volume (the blue block) is much smaller than that of the "reserved area", the hidden volume can "float" inside the "reserved area". Hence, the exact starting position of the hidden volume can be arbitrary, hidden volumes are thus protected. However, this mechanism could waste a large amount of storage space (green blocks). We find in these solutions [7–9], up to 45%[1] of the total storage space is wasted, which is huge for the resource-limited mobile devices.

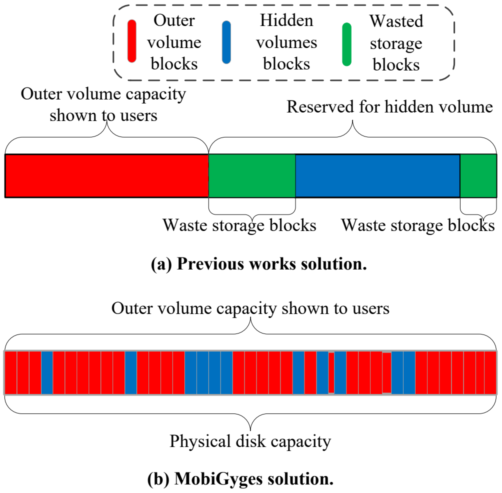

Figure 2: Hidden volume is placed into a reserved area to avoid data override. Large amount of storage space is wasted. MobiGyges can fully utilize almost all the storage space.

- **Device reboot.** State-of-the-art works use two modes in their system, namely, the *normal mode* and the *PDE mode* [7–11]. The normal mode uses the outer volume while the PDE mode uses the hidden volume, respectively. When users want to use the hidden volume, they have to use the PDE mode. However, device reboot is required to switch modes. Rebooting the device to use the PDE mode wastes time and is not convenient for users especially those who want to use the hidden volume urgently.

Apart from the drawbacks, we identify two possible PDE oriented attacks (detailed in Section 4) that might compromise the sensitive data, and current solutions fail to defend.

- **The capacity comparison attack.** The attacker may discover the hidden volume by comparing the capacity of the outer volume and the hidden volume. If their capacities are different, the attacker can doubt the device is particularly designed, which is prone to expose the hidden volume and hence compromises the sensitive data. For example, a 32GB device uses 5GB for the hidden volume, so the capacity of the outer volume is 27GB. The attacker can

[1] Including storage space taken up by the structure of a file system. The calculation of the utilization is detailed in Section 7.

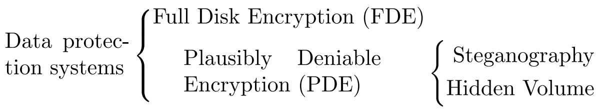
Data protection systems
Full Disk Encryption (FDE)
Plausibly Deniable Encryption (PDE)
Steganography
Hidden Volume

Figure 3: Data protection systems classification.

doubt about the 5GB capacity difference, and conduct further investigation.

- **The fill-to-full attack.** If the attacker identifies the potential existence of the hidden volume, he/she may conduct the fill-to-full attack to explore the real capacity of the outer volume by writing arbitrary data to the outer volume and filling it until full. After filling the outer volume, the attacker gets the audited information and conducts the capacity comparison attack. If the real capacity is different from that of the physical disk, the hidden volume will be compromised.

Existing solutions cannot solve the three problems simultaneously, and they cannot defend the two attacks. To this end, we present MobiGyges in this paper. MobiGyges is a hidden volume based PDE system. It introduces the Volume Management module and FDE module, which prevents sensitive data loss by restricting each storage block usable by solely one volume, and improves the storage utilization by eliminating the "reserved area" (as shown in Figure 2b), and avoids rebooting the device to use the hidden volume by introducing the Dynamic Mounting service that mounts the hidden volume on-demand. MobiGyges also uses the *Shrunk U-disk* method (detailed in Section 5.3.1**(3)**) and *multi-level deniability* (detailed in Section 5.3.2**(3)**) to jointly defend the aforementioned attacks.

Our main contribution in this paper is threefold, summarized as follows.

- We propose MobiGyges to solve the data overriding problem, improve storage utilization with the aid of Thin Provisioning and Device Mapper, and avoid device reboot to use the hidden volume on-demand by introducing the Dynamic Mounting service.

- We identify the capacity comparison attack and the fill-to-full attack, and propose the Shrunk U-disk method and multi-level deniability to jointly defend them.

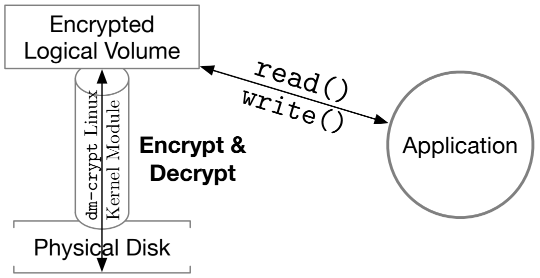

Figure 4: Encrypted logical volume and physical storage. Applications use the normal system call to read or write data to the encrypted logical volume. The `dm-crypt` Linux module automatically encrypts and decrypts the data between the encrypted logical volume and the physical disk.

- We implement the MobiGyges proof-of-concept system on Google Nexus 6P with the LineageOS [13] 13 by porting Thin-Provisioning (pdata_tools) and Logical Volume Management (LVM) into the Android system and implementing a TriggerApp to use hidden volume on-demand secretly. We conduct experiments to evaluate MobiGyges's storage utilization, performance overhead, and experimental results show that MobiGyes reaches all our design goal and improves storage utilization by over 30% compared with current solutions.

The rest of the paper is organized as follows. Section 2 introduces related works, and Section 3 presents the threat model and assumptions. In Section 4, we introduce our newly identified PDE oriented attacks. Section 5 presents the design of MobiGyges. In Section 6, we present the implementation of MobiGyges with LineageOS 13 on Google Nexus 6P. In Section 7, we conduct rigorous experiments and analyze the experimental results. Section 8 discusses common attacks defended by MobiGyges, the drawback, and possible future works. Finally, we conclude the paper in Section 9.

2. Related Works

Data protection is of paramount importance, and there have already been various systems proposed for security. In this section, we categorize current data protection systems (as shown in Figure 3) and introduce related works based on the taxonomy. Other types of file systems like versioning file system [14] that are useful in post-intrusion file system analysis applications, or reliable file retention and retrievability required by legal regulations for sensitive data management. However, they are not suitable for personal sensitive data protection. Hence, they are out of the scope of this paper and, therefore, not discussed here.

2.1. Full Disk Encryption

FDE is an elementary way to prevent sensitive data from attacks by encrypting all data on a volume. As shown in Figure 4, on Linux based systems, FDE creates an encrypted logical volume with `dm-crypt` Linux kernel module, and it encrypts all the data that saved on the logical volume before committing to the physical storage. Similarly, when an application needs to read data, it automatically decrypts the data from the physical storage and redirects the data to the logical volume. Thus, FDE is transparent to applications. FDE has now been a standard configuration for any security system. There are many mature FDE solutions, e.g., TrueCrypt [15, 16], BitLocker [17, 18], LUKS (Linux Unified Key Setup) [19], FileVault [20].

TrueCrypt [15, 16] is a cross-platform encryption software that supports multiple cipher encryption scheme. It underpins various ways for both software architecturally and hardware chip performance improvement to speed up the full disk encryption process. TrueCrypt supports PDE, which will be introduced in Section 2.2.2.

BitLocker [17, 18] is a built-in encryption utility software on Windows developed by Microsoft to conduct FDE. It has been a system-level component since Windows Vista. The default encryption scheme is AES with cipher block chaining (CBC) or XTS with a 128-bit or 256-bit key. Note that CBC is not used over the whole disk but applied to each individual sector.

LUKS [19] (Linux Unified Key Setup) is an encryption specification for Linux used for employing FDE. Unlike most of the encryption software creates their own encryption functionality. LUKS creates a unified encryption format that can be used for various tools like `cryptsetup`.

FileVault [20] is another FDE solution introduced with Mac OS X Panther by Apple Inc. As recommended by NIST [21], it uses the AES-XTS mode of AES with 128-bit blocks and a 256-bit key to encrypt the disk.

However, since FDE uses only one encryption key, all the FDE-only solutions fail to provide users the deniability of sensitive data stored on their devices, which is not enough for protecting the sensitive data.

2.2. Plausibly Deniable Encryption

PDE is the kind of encryption paradigm that provides the user with the ability to deny the existence of sensitive data on the device. There are primarily two ways of implementing PDE: Steganography and the hidden volume.

2.2.1. Steganography

An example of Steganography is hiding sensitive data into a multimedia file like a photo, a video or an audio file as noise points. Since people tend to ignore those noise points, the sensitive data is thus protected. However, this technique requires a large amount of computation, which is not appropriate for mobile devices, because mobile devices lack of computation and storage resources. StegFS [22] is a steganography-based file system, and its key idea is to hide data in a bunch of cover files. Another work [23] uses external entropy sources and erasure codes to deniably and reliably store data within the unallocated space of an existing file system. However, these solutions has the following shortcomings. i) It wastes storage space. ii) The performance is low especially when writing. iii) The possibility of data loss is high. iv) The modification of Ext2 may lead to compromise of deniability. All these shortcomings make it not suitable for mobile devices.

2.2.2. Hidden volume

Hidden volumes achieve PDE by concealing themselves into a device. It is light weighted and has minimum computational and storage burdens. However, current hidden volume solutions have mainly three drawbacks that we have pointed out in Section 1, and these drawbacks cannot be addressed at the same time. In this section, we classify works based on the drawbacks they solve as follows.

(a) Data loss reducing solutions

TrueCrypt [15] and FreeOTFE [24] are PC PDE solutions. They can create hidden volume(s) as files or physical volumes. However, both TrueCrypt and FreeOTFE can only create PDE hidden volume on their resource files. Therefore, if these files are broken or lost, all the data stored on the hidden volumes will be lost. Thus, the solution is prone to compromise the sensitive data.

Table 1: Feature comparison between MobiGyges and current works. (✓ means the functionality is provided, ✗ means the functionality is not available, and — means not applicable.)

Features	Mobiflage [7]	MobiHydra [8]	MobiPluto [9]	MobiMimosa [11]	MobiCeal [12]	MobiGyges
Data loss prevention	✓	✓	✓	✗	✓	✓
Reserved area elimination	✗	✗	✗	—	✗	✓
Device Reboot avoidance	✗	✗	✗	✗	✓	✓
Capacity comparison attack defense	✗	✗	✗	✗	✗	✓
Fill-to-full attack defense	—	—	—	—	—	✓

Mobiflage [7] is the first implementation of mobile hidden volume based PDE prototype system on Android. To avoid sensitive data loss, Mobiflage reserves a block of storage space and places the hidden volume at an arbitrary position in the area. We call it the “reserved area” technique. However, up to 45% of the total physical storage is wasted. Based on Mobiflage, MobiHydra [8] implements multi-level deniability. However, similar to Mobiflage, MobiHydra also fails to solve the problem of low storage utilization.

MobiPluto [9, 10] is the first file system friendly PDE solution built on the Android operating system. It leverages virtual logical volumes techniques, and all block-based file system can run on it without modifications. It also addresses the data loss problem by placing the hidden volume into a “reserved area”, therefore, the storage utilization is low.

MobiCeal [12] is another recent hidden volume PDE solution, and its key contribution is to defend against strong coercive multi-snapshot adversaries. It does not consider the storage waste and capacity inconsistency problems in previous solutions.

(b) High storage utilization solutions

MobiMimosa [11] is our former work of the hidden volume based PDE solution for the Android. Its key idea is manually choosing storage blocks on a physical device and converting the blocks into a hidden volume. However, the location information of the storage blocks is stored on a `dm_table` file. Once getting the file, the attacker can easily get the sensitive data, and if the `dm_table` file is lost, all the sensitive data on the hidden volume will be lost.

TrustGyges [25] addresses the above problem by storing the `dm_table` file on to a cloud server. In order to avoid run-time attack, it fetches the `dm_table` file inside the Trust Execution Environment (TEE). However, it fails to propose a proper method to use the hidden volume without the network connection.

(c) Reboot avoiding solutions

MobiCeal [12] is the latest PDE solution and is the only mobile PDE solution that does not require device reboot. Similar to all other solutions, it still has the concept of normal mode and PDE mode. Its main concern is how to fastly switch from the public mode to the hidden mode (PDE mode). It achieves the device reboot avoidance by restarting the Android framework. Although, it reduces the switch time, but the time of restarting the Android framework is still too long (≈10s for entering the PDE mode and ≈70s for returning to normal mode). Furthermore, applications will suffer response lags after restarting the Android framework that results from CPU cache misses, and hence influences the user experience.

2.2.3. Functionality Comparison

Although mobile PDE has been explored by [7–9, 11, 12], we differentiate MobiGyges with them in Table 1 on key features desired for PDE oriented data protections.

First, most of the current solutions use the “reserved area” to prevent data loss resulting from data override on the hidden volume. Therefore, they fail to utilize the storage space efficiently. Second, all current works cannot defend our newly identified two novel PDE oriented attacks. Consequently, the possibility of exposing the hidden volume and compromising the sensitive data is high. Our proposed MobiGyges system can address all of these issues at the same time.

3. Threat Model and System Assumptions

The key to protect sensitive data on a hidden volume based PDE system is concealing the hidden volume [26]. We propose the following threat model and put the beneath assumptions for MobiGyges based on works presented in [7, 8].

1. The attacker knows that the `userdata` partition is encrypted by FDE and also the key of this encryption. But the attacker lacks the knowledge of MobiGyges's design. Thus, the attackers still cannot retrieve the sensitive data because they do not know where and how to do that. Alternatively, the attacker knows about the design of MobiGyges; but, not sure how many hidden volumes there are on the phone.

2. The attacker knows the design of MobiGyges but is uncertain if the key provided by users is the key to the hidden volume he sought.

3. The attacker can get root privilege and the physical storage of the phone or dump the raw data from the storage medium.

It is notoriously hard to achieve security on a device with backdoor hardware or software. Therefore, assumptions involving the backdoors of the device are necessary for us to design MobiGyges. We make the following assumptions accordingly.

1. The hardware of users' phone is backdoor-free.

2. The system level software (e.g., bootloader and the mobile operating system) are backdoor-free.

Without these assumptions, user's operational behavior can be monitored, which is impossible to conceal the hidden volume and protect the sensitive data stored on it.

4. Novel PDE Oriented Attacks

In this section, we introduce our identified two novel PDE oriented attacks and assume attacker $\mathcal{A}$ conducts the following attacks. Since these two attacks are correlated with each other, we propose joint solutions to defend the attacks.

Table 2: Notation definitions used throughout the paper

.

Notation	Description
$\mathcal{A}$	An attacker.
D	A mobile device.
C_p	The physical capacity of the device D.
C_o	The capacity of the device D's outer volume.
O_l	The sector offset of the mapping logical volume.
S_l	The number of sectors of the original volume.
T	The mapping type.
D_p	The mapped device.
O_p	The offset of the mapped physical volume.
S_m	The size of metadata volume.
S_p	The size of pool volume.
S_c	The chunk size of pool volume.
N	The name of the hidden volume.
$T(s,b)$	The trim function that trims string s into a b-length string.
$h(x)$	The hash function that hashes variable x into a hash value.
η	The storage utilization.
$Offset$	The hidden volume offset in Mobiflage [7].
$vlen$	The capacity of the device in Mobiflage [7].
$H(x)$	The PBKDF2 iterated hash function in Mobiflage [7].
pwd	The password of the hidden volume in Mobiflage [7].
$salt$	The random salt value for PBKDF2 in Mobiflage [7].

4.1. Capacity Comparison Attack

When PDE systems are created using hidden volumes, the total capacity of the outer volume and hidden volumes should be equal to the physical capacity. Hence, if untreated, the capacity of the out volume will be smaller than the physical capacity. By comparing the capacities of the outer volume and the physical disk, the attacker can get the capacity inconsistency, and the capacity inconsistency offers attacker $\mathcal{A}$ an indication of the potential existence of hidden volumes, which may lead to the attacker $\mathcal{A}$ to conduct a further investigation. Thus, the attack makes hidden volume based PDE system highly prone to compromise the sensitive data. We formalize the attack as shown in Equation 1, and all notation definitions can be found in Table 2.

$$\mathcal{A} \leftarrow D \leftarrow \begin{cases} 1, & C_p > C_o \\ 0, & C_p = C_o \end{cases}, \qquad (1)$$

where, D denotes the device, C_p is the physical capacity of the device, and C_o is the capacity of the outer volume. 1 represents device D has special design of its storage system (PDE is compromised), and 0 otherwise.

Existing literature fails to defend the capacity comparison attack. According to Equation 1, we can leverage the second condition in the equation to defend the attack by eliminating the capacity difference between the outer volume and the physical disk. To this end, we have to find a method to modify the capacity of the outer volume C_o to be the same as the physical capacity C_p. We detail the method used of defending the attack in Section 5.

4.2. Fill-to-Full Attack

The fill-to-full attack is a complementary attack of the capacity comparison attack. Suppose now we can defend the capacity comparison attack by setting the capacity of the outer volume to be the same as that of the physical disk. Attacker $\mathcal{A}$ may still doubt that the device might have hidden volumes, but he/she is uncertain about it. To this end, attacker $\mathcal{A}$ may write arbitrary data on the outer volume by filling data to the outer volume and auditing the total size of the data that have been written. After getting the audited information, attacker $\mathcal{A}$ can further conduct the capacity comparison attack by comparing the physical capacity with the audited data size plus used capacity on the outer volume before the fill-to-full attack.

We demonstrate the fill-to-full attack with a real-world example. Suppose Alice has two containers: one is a standard 1-liter container, and another is marked as 3 liters. Alice wants to know if the second container is a 3-liter container precisely as it marked. She can fill up the 1-liter container with water and pour the water to the 3-liter container for 3 times. If the total amount of water cannot fill up the 3-liter container, or the 3-liter container overflows, Alice can confirm that the 3-liter container does not have a 3-liter capacity. The fill-to-full attack uses the same strategy.

Since the real size of the outer volume is smaller than the physical capacity. The audited size should be smaller than the physical capacity, which again reduces to the capacity comparison attack as we introduced in Section 4.1. In the next section, we introduce MobiGyges design, including how we defend the two attacks.

5. MobiGyges Design

In this section, we introduce the design of MobiGyges. MobiGyges is designed for protecting sensitive user data rather than system data that include program executables and data. We first present our design considerations, and we then propose our design overview. As the Volume Management module is the key component of MobiGyges, we finally make a detailed anatomy of the design of the Volume Management module.

5.1. Design Considerations

The current hidden volume based PDE solutions cannot address the aforementioned three problems at the same time, and they also fail to defend the capacity comparison and the fill-to-full attacks. MobiGyges is designed to conquer them all.

(1) Eliminating the data override issue. The root of data loss is due to the data override phenomenon between the outer volume and the hidden volume(s), which two or more volumes access the same storage block. Therefore, if we restrict one physical storage block that can only be used by one volume, the issue is addressed. Thus we can split the total storage blocks into fine-grained storage blocks instead of coarsely divide the whole physical storage into two parts, and the outer volume and the hidden volume(s) allocate storage space from the fine-grained storage blocks (as shown in Figure 2b).

(2) Improving the storage space utilization. As we have introduced, the low storage utilization results from the “reserved area”. Therefore, eliminating the “reserved area” can improve the storage utilization.

(3) Avoiding device reboot. Rebooting the device costs time, which is not convenient for users, especially when users have to capture some important data and want to store it immediately into the hidden volume. We intend to avoid device reboot

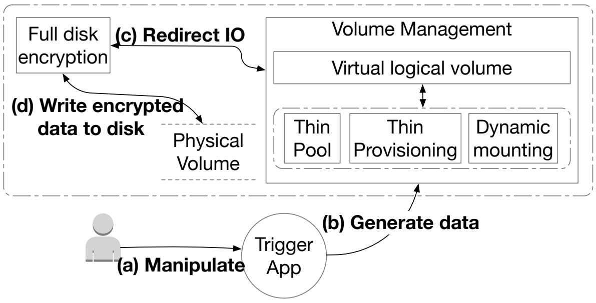

Figure 5: MobiGyges key components and data-flow.

by mounting the hidden volume on-demand without rebooting the device.

(4) Defending the novel PDE oriented attacks. The capacity comparison and the fill-to-full attacks result from the capacity inconsistency issue. The challenge is how we defend the fill-to-full and capacity comparison attacks.

5.2. Design Overview

MobiGyges is a mobile hidden volume based PDE system. In this subsection, an overview of MobiGyges design, starting with a general description of its components, is presented. Then, we propose a so-called *Shrunk U-Disk method* to defend the capacity comparison attack, and we also leverage the *multi-level deniability* to defend the fill-to-full attack. Next, we introduce our design solution associated with the design consideration introduced in Section 5.1. Finally, we describe the user steps.

5.2.1. Solutions

With the design considerations discussed in Section 5.1, we propose our PDE system design solutions. i) We introduce an independent component called the Volume Management module to manage the outer volume and hidden volume(s) rather than having the volumes themselves handle the hidden volume concealing. It flexibly allocates separate storage blocks for volumes and avoids the override of the same storage space. The module also eliminates the "reserved area" and thus mitigates data loss and improves storage utilization. We use a FDE component to encrypt the Volume Management module and the file system structure, which protects the system. ii) We design a userspace application called TriggerApp and the dynamic mounting service to mitigate device

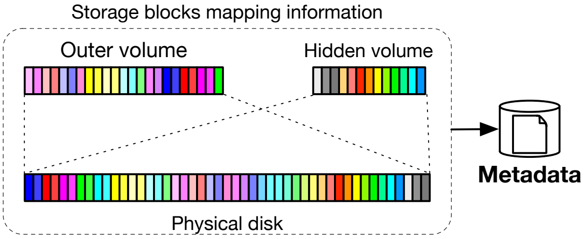

Figure 6: Splitting the physical disk into fine-grained storage blocks (each block is represented by different colors), and each fine-grained block is used only for one volume (blocks of the outer volume and the hidden volume are allocated from the split physical disk). Thus, the "reserved area" is not required in the design. The occupancy information is stored into the metadata datastore.

reboot. The dynamic mounting service can mount the hidden volume without rebooting the device, and TriggerApp can secretly trigger the use of hidden volumes and mount the hidden volume on-demand using the dynamic mounting service without rebooting the device. iii) We use the *Shrunk U-disk method* and *multi-level deniability* jointly in the Volume Management module to defend the capacity comparison and the fill-to-full attacks. In terms of fill-to-full attack, we leverage multi-level deniability by recording the size of data that has already been written and redirecting the extra attack IO requests to other places that will not take up the physical storage space and finally making the size of the attack IO plus the used capacity before the attack equals the physical capacity. Moreover, we employ the Shrunk U-disk method to defend the capacity comparison attack by intentionally labeling the capacity of the outer volume capacity equivalent to the physical capacity. We detail the Shrunk U-disk method and multi-deniability in Section 5.3.1**(3)** and Section 5.3.2**(3)**.

5.2.2. Key Components

MobiGyges consists of four main components. *Volume Management module, FDE module, physical volume module, and TriggerApps module.* i) The Volume Management module is the most important module of MobiGyges, and it manages the life-cycle of all the outer volume and hidden volumes. ii) FDE module is the encryption layer on top of the physical storage to protect the Volume Management module, and iii) physical volume module is the physical storage of the device provided by the device vendor. iv) TriggerApp module is an application, which is designed for the user to utilize the

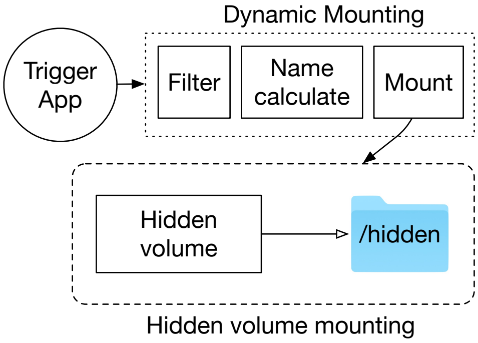

Figure 7: The Dynamic Mounting service structure. Authorized App can mount the hidden volume on-demand.

hidden volume securely and conveniently without requiring entering the detailed system commands.

Figure 5 depicts the data-flow in the entire MobiGyges workflow and the inner data-flow between MobiGyges components. As per the mentioned figure, **(a)** users first manipulate TriggerApps, and **(b)** TiggerApps then generates data and exchanges data between the Volume Management. **(c)** Next, the Volume Management module processes the data with *Device Mapper* and *Thin Provisioning*, and sends Input/Output (IO) redirections to the FDE. **(d)** PDE finally encrypts or decrypts data with the `dm-crypt` kernel module and communicates with the physical volume and commits data on the physical volume or reads data from it.

5.2.3. User steps

Steps for using MobiGyges:

1. Boot the device with MobiGyges. The outer volume is mounted automatically. After booting up the device, the user can start using it as an ordinary device for daily purposes.

2. When using hidden volumes, open TriggerApp and enter a level (level of deniability is detailed in Section 5.3.2) of the password to mount the corresponding hidden volume without rebooting the device.

3. If an attacker suspects that PDE exists on the device and conducts the fill-to-full attack, the user can use level 0 before then (detailed in Section 5.3.1**(4)**) to defend the attack.

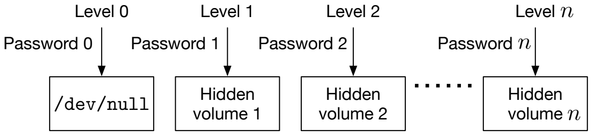

Figure 8: Multi-level deniability. Level 0 is used to defend the fill-to-full attack.

5.3. The Volume Management Module

This subsection details the design of the key part of MobiGyges, the Volume Management module. The Volume Management module leverages *Thin Provisioning* and *Device Mapper* by converting the physical storage into a Thin Pool and using the virtual logical volume to manage the physical storage in a fine-grained manner. In the rest of this subsection, we show how fine-grained storage blocks used by the Volume Management module solves the aforementioned problems and defends the attacks, and we then anatomize each services in the module.

5.3.1. Volume Management Module Circumvention

(1) Data Loss and Low Storage Utilization Elimination

MobiGyges avoids data loss and improves the low storage utilization using the Volume Management module with the following steps as shown in Figure 6.

i) The Volume Management module splits the whole physical disk into fine-grained storage blocks (i.e., 64KB), and then, ii) the module allocates each fine-grained storage block only for one volume, so the storage space of each volume is independent, and the data on each fine-grained storage block cannot be override by other volumes. Hence, the data loss problem is avoided. iii) The module tracks the usage of each fine-grained block, and stores volumes and the storage block mapping information (metadata) in a special format and encrypts the metadata with FDE on the physical disk. The metadata is stored in the metadata logical volume as shown in Figure 9. When allocating storage space for the metadata logical volume, the available storage blocks are first sorted from big to small, and the least storage block that is larger than the needed storage size is used. In the figure, we can see that iv) no “reserved area” is used because of the exclusive access of the same storage block from volumes, and hence the storage utilization is improved.

(2) Device Reboot Avoidance

MobiGyges avoids device reboot when using the hidden volume by leveraging the Dynamic Mounting service in the Volume Management module. The Dynamic Mounting service first filters the mounting request by identifying the owner of the request, and it only allows authorized applications (e.g., the TriggerApp) to mount the hidden volume. The mounting request is sent associated with a token generated by using OAuth [27], and thus if the token is not valid, the Dynamic Mounting service rejects the request. Then the Dynamic Mounting service calculates the name of the hidden volume based on the given PDE password and mounts the hidden volume on the mounting point. The procedure is shown in Figure 7.

(3) Capacity Comparison Attack Defense

The attacker may conduct the capacity comparison attack and the fill-to-full attack to expose the hidden volume and compromise the sensitive data.

MobiGyges leverages the *Shrunk U-disk method* to defend the capacity comparison attack. Shrunk U-disk is a kind of disk labeled with a bigger capacity, and operating systems also display the labeled capacity to users. However, the real capacity of the disk is smaller than the labeled capacity. For example, a memory stick is labeled to have 32GB storage, and the operating system also shows the capacity is approximately[2] equal to 32GB, but its actual capacity is only 8GB. Only when exhausting more than 8GB storage space on it, can the user find out the trick. However, users usually would not do that and cannot discover the "trick". We can exploit this "trick" to defend the capacity comparison attack by simply modifying the capacity of the outer volume to be the same as the physical capacity, which protects the hidden volume. The implementation choice of Shrunk U-disk method is detailed in Section 5.3.2.

The challenge is how we can modify the volume capacity. We leverage *Thin Provisioning* to fill the gap. Thin Provisioning is a novel technology designed to address the storage waste problem in the Cloud data center (DC). Because the servers are usually installed with larger physical storage capacity than the actual size of data usage for future demands. This is called *Thick Provisioning*, but the physical storage cannot reach its capacity upper limit before replacing the disk with a larger one. Hence, a large amount of storage space is wasted. Thin Provision allows operators to flexibly preconfigure a bigger capacity of virtual volumes than the physical capacity, and physical storage spaces are not be used until data commit on them. When the physical storage exhausts, the operator can install new physical disks without modifying any previous settings. We employ the flexibility of Thin Provisioning and set the capacity of the outer volume to be the physical capacity to mitigate the capacity comparison attack.

(4) Fill-to-Full Attack Defense

It is natural to conduct the fill-to-full attack when the attacker suspects the existence of the hidden volume(s). We leverage *multi-level deniability* to defend the attack. Multi-level deniability allows users to define different levels of importance of sensitive data, and each level of sensitive data is associated with one hidden volume. To defend the fill-to-full attack, we introduce a special level (level 0) of deniability, which is associated with `/dev/null`. As shown in Figure 8, when the attack discovers the hidden volume and tries to conduct the fill-to-full attack, the user uses the level 0 deniability and gives the device to the attacker to perform the fill-to-full attack. Attack data will first be written to the outer volume and recorded by the Volume Management module. When the physical disk exhausts, new attack data will be directed to `/dev/null`, and the Volume Management module will occur a full storage error when the recorded size of attack data plus the used capacity of the outer volume before the attack equals physical capacity.

5.3.2. Volume Management Module Composition

In this subsection, we bring the technical design details of the Volume Management module and present how the Volume Management module manages the physical disk.

The Volume Management module leverages *Thin Provisioning* and *Device Mapper* to convert the physical disk into a *Thin Pool*, which is a resource pool consists of fine-grained storage blocks. The Volume Management module then create *virtual logical volumes* as the outer volume and hidden volumes. The storage spaces of the virtual logical volumes are allocated from the Thin Pool, and each

[2] It is a convention that storage hardware vendors using 1k=1000 rather than 1k=1024 to calculate the capacity.

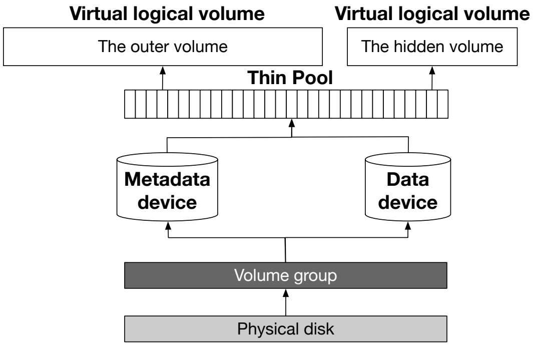

Figure 9: The creation of the outer volume and hidden volume. The outer volume and the hidden volume are virtual logical volumes.

virtual logical volumes allocates storage blocks distinctly from others.As depicted in Figure 9, i) the Volume Management module converts physical disk into a volume group with Device Mapper. Each volume group consists of one or more physical disks. ii) Then, the Volume Management module creates two logical devices using the created volume group with Device Mapper, namely, the data device and the metadata device. The data device is the storage space of Thin Pool, and the metadata device records the usage of each fine-grained storage block. Device mapper is used to map existing block devices into another logical block device. Device mapper redirects or filters IO requests from logical block devices to the mapped device (physical disk in our case). The process can be formalized by a 5-tuple, $\langle O_l, S_l, T, D_p, O_p \rangle$, where O_l denotes the sector offset of mapping logical volume block, and S_l is the number of sectors of the original volume block. T denotes the type used to describe the way of mapping. D_p denotes the mapped device, and O_p denotes the offset of mapped physical volume block. iii) Next, the Volume Management module converts the the two devices into a Thin Pool. iv) Finally, the Volume Management module creates two virtual logical volume by using fine-grained storage blocks in Thin Pool with Device Mapper.

We anatomize each components in the Volume Management module in the rest of this subsection.

(1) Thin Pool is the resource pool used for virtual logical volumes and created atop the encrypted logical volume. As shown in Figure 11, it virtualizes the encrypt logical volume as a resource pool by carefully splitting the whole storage space into fine-grained storage blocks, and the outer volume and the hidden volumes allocate storage space from the pool on demand. When one storage block in the Thin Pool is allocated by one virtual logical volume, the storage block will be marked as used, and it will not be allocated for other virtual logical volumes. Hence the overriding problem is addressed. The mapping of fine-grained storage blocks and virtual logical volumes is stored in the metadata device in Figure 9. In our case, each block is 64KB in size. Since each block is exclusively used by the hidden volume or the outer volume separately, the "reserved area" is needless. Thus, the storage utilization also improves.

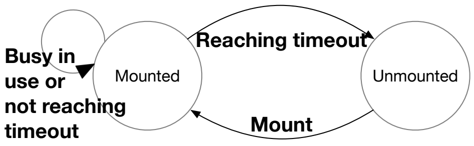

Figure 10: Dynamic mounting state management.

When creating the data and metadata logical volumes, the relationship between the size of the metadata volume and data volume can be formalized as follows.

$$S_m = \frac{S_p}{S_c} \times 64, \tag{2}$$

where S_m denotes the size of metadata volume. S_p denotes the size of pool volume, and S_c denotes the chunk size of pool volume.

(2) Virtual logical volume is the final logical volume used as an outer volume or a hidden volume. The reason it is called "virtual" is when creating the virtual logical volume, its storage resources is not allocated until data commit on it. The capacity assigned at the creation time is only a label, and it does not indicate the actual capacity it possesses. The topmost part of Figure 11 shows the architectural level in the MobiGyges system. In MobiGyges, the capacity of the outer volume is configured as big as the physical volume to defend the capacity comparison attack. Device mapper is used to create the virtual logical volume from Thin Pool. For applications, the usage of the virtual logical volume has no differences between a physical volume.

(3) Multi-deniability allows users to define different levels of importance for their sensitive data and put them into different hidden volumes corresponding to different levels of deniability. Mobi-

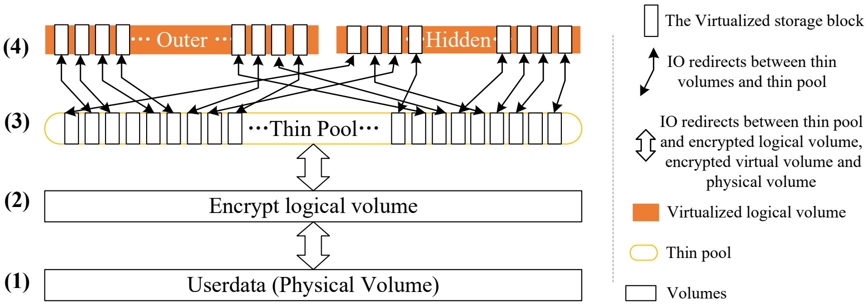
(4)
··· Outer ···
Hidden
(3)
···Thin Pool···
(2)
Encrypt logical volume
(1)
Userdata (Physical Volume)
The Virtualized storage block
IO redirects between thin volumes and thin pool
IO redirects between thin pool and encrypted logical volume, encrypted virtual volume and physical volume
Virtualized logical volume
Thin pool
Volumes

Figure 11: Disk layout of MobiGyges. **(1)** The `userdata` is the physical volume that used for storing user-generated data. **(2)** FDE is applied to encrypt both data and the structure on top of it. **(3)** all the `userdata` storage is virtualized as a Thin Pool. **(4)** virtual logical volumes are created as outer volume and hidden volume, respectively.

Gyges provides multi-deniability by creating multiple hidden volumes. Each hidden volume is reserved for providing a specific level of deniability. To this end, even if the hidden volume is exposed, users may still deny the existence of sensitive data because the attacker fails to know the number of deniability the system provides, and which hidden volume is used to store the sensitive data. Each hidden volume has its name and password, and the name is calculated from the password. Without the password, the system cannot find the correct name of the hidden volume and cannot mount the hidden volume and thus fails to fetch the sensitive data stored on it. The name of each hidden volume is calculated with Equation 3.

$$N = T(h(passwd + salt), b), \qquad (3)$$

where, N is the final name. $T(s, b)$ is a trim function that trims s into a b-length string. $h(x)$ is a hash function that hashes x into a hash value. $passwd$ is the user password towards a hidden volume, and $salt$ is used for counter rainbow table attack [28].

(4) Dynamic Mounting mounts the hidden volumes on demand. Unlike previous solutions, which requires rebooting the device and logging into the PDE mode to use the hidden volumes. MobiGyges can use hidden volume on demand needless device rebooting to switch to the PDE mode. Therefore, authenticated applications can use dynamic mounting to mount the needed hidden volume instantly. Due to the confidentiality of the hidden volume, operations of authenticated applications should be cautious. When requesting for mounting the hidden volume, an access token is needed to validate the application, and if the token is not valid, the mounting request is rejected.

Android is a UNIX-like system, and in a UNIX-like system, devices are presented as block device files. Each block device file has its own name. Therefore, when using a hidden volume, users first enter the password of the wanted hidden volume at authenticated application such as TriggerApp and MobiGyges will mount the correct hidden volume based on Equation 3. As depicted in Figure 10, when the hidden volume is mounted, a timer is started, and the hidden volume will be automatically unmounted as the timer times out. This mechanism protects the hidden volume from being discovered by examining the currently mounted devices.

5.3.3. TriggerApp

TriggerApp is the interface among the users and MobiGyges's hidden volumes. Users have to use TriggerApp to trigger the special operations and dynamically mount the hidden volume for storing sensitive data. TriggerApp functionality should be secret to prevent MobiGyges from exposing. Therefore, TriggerApp is recommended to be implemented inside a system built-in application.

6. Implementation

We implement our MobiGyges prototype system on LineageOS 13 for Google Nexus 6P. We choose LineageOS rather than original Android Open Source Project because LineageOS provides necessary vendor specific hardware adaptation like camera and baseband driver supports. For the Volume Management component of MobiGyges, we add about 500 lines of C code to LineageOS, and we port Logical Volume Management (LVM) and Thin Provisioning tools (pdata_tools) [29] and some system building scripts to Android. For TriggerApp, we add approximately 500 lines of Java code. In this section, we present the implementation challenges and considerations of MobiGyges.

(1) Manipulating the `userdata` partition is hard on Android. MobiGyges initialization requires mounting and unmounting the `userdata` partition. However, the `userdata` partition cannot be unmounted while the system is running because some system files are stored on the `userdata` partition, and these files are busy in use [30]. For system stability and data integrity, the operating system does not allow the `userdata` partition to be unmounted while busy. However, the creation of Thin Pool and virtual logical volume requires a free (not in used) partition. To this end, we put the Thin Pool and virtual logical volume creation procedure executing at the system booting stage before mounting the `userdata` partition. We put the code right after the FDE procedure in Android Volume Daemon (VOLD) located at `cryptfs.c`. The code executes `lvm` toolset by forking a child process.

(2) Running toolsets on Android is another challenge for us because the toolsets are usually for desktop and server that are x86 platforms rather than the ARM-based mobile platforms. Luckily, Android is based on Linux kernel and has the kernel modules needed by Thin Provisioning and Device Mapper, and the system calls are the same as desktop and server versions. Therefore, porting LVM and pdata_tools that are used to build logical volumes, Thin Pool, and virtual logical volumes does not require much code and building system modification. We use `gcc-arm-linux-androideabi` to conduct the cross-compile. We first intended to compile the source directly, but these tools require libraries that should be cross compiled in advance and made them Android runnable. Apart from that, `-enable-static_link` and `LDFLAGS=-static` flags should be set to ensure the compilation target is statically executable. Otherwise, the toolsets cannot run when pushing to Android because Android lacks of these libraries. We also modified the code in the Android build system to compile the toolsets into the final installation package.

(3) TriggerApp implementation is an important part of the work of implementing MobiGyges. We implement our TriggerApp into the Android system built-in Calculator. Apart from the basic calculation functionality, our tailored Calculator has the extra functionality like secret recording, filming, and picturing. The calculator could still make calculations, and otherwise, the system is prone to expose the special design and thus compromise sensitive data. As we all know, dividing any number by 0 is an illegal operation. When users attempt to do such an illegal operation, the system will throw an exception and prompt an alert to users to indicate the calculation is not allowed. Thus, we change the exception processing behavior of the calculator by modifying it to display a special operation Use Interface (UI) according to what the user enters. Identifying the divided by 0 operation can be done by parsing the input `String` every time when the user presses the '=' button. However, users usually just conducts normal calculations operations, and the input parsing is worthless under such condition. Android applications are written in Java[3]. Java has an `ArithmeticException` that whenever an illegal operation occurs, it throws an `ArithmeticException`. We put our parsing code into the `catch {}` code block. When an `ArithmeticException` is triggered, the code will analyze the input and process the operations accordingly.

(4) Dynamic mounting is implemented as SystemService. Normally, all kinds of Android supporting hardware has the SystemServices and Hardware Abstract Layer (HAL) definition pair. Apart from that, lightweight Service and lightweight HAL are also provided as pairs in Android. For example, the WiFi module has both Service and HAL definitions. But the HAL definition is needless in our system because no new physical hardware devices are added to the system. We implement the MobiGyges SystemService by using the method provided by [31]. Inside the dynamic mounting SystemService, it i) calculates the trimmed hash value with the input password and salt by using Equation 3. The salt, uses the crypto footer of the `userdata` partition in our case, and ii) it mounts the corresponding hidden volume with the calculated device name and the access token. If either the password or the access token is incorrect, the mounting request is rejected.

[3] C/C++ can also be applied to Android application to improve the running efficiency

(5) Full Disk Encryption (FDE) is used to protect the Volume Management module. MobiGyges first performs FDE by creating an encryption layer on the `userdata` partition, which makes it easy to encrypt all data on the `userdata` partition. MobiGyges uses 128-bit Advanced Encryption Standard (AES) [32] with cipher-block chaining (CBC) and ESSIV:SHA256 to perform the encryption. The master key is encrypted with 128-bit AES via invocations to the OpenSSL library. It is recommended that users use 128 bits or more for the key (with 256 being optional) to improve the security. Figure 11**(2)** represents the encrypted logical volume created atop the physical volume. It also shows the logical position in MobiGyges. FDE is created using the Device Mapper technology.

7. Evaluation

In this section, we describe experiments on storage utilization and performance overheads by comparing MobiGyges with state-of-the-art works. We conduct experiments on Google Nexus 6P mobile phone with 3GB LPDDR4 DRAM and an octa-core CPU with 4 Cortex-A53@1.55GHz cores and 4 Cortex-A57@2GHz cores. We use `dd` [33], bonnie++ [34], and AndroBench [35] to conduct the experiments.

7.1. Performance Evaluation Tools

`dd` copies blocks of data from one file to another and is provided by most UNIX platforms. It allows parameters like r/w buffer size that can be easily used for IO performance study. We also use the `fsync` parameter in `dd` because it does not bypass the kernel disk caches, and when it writes data to the device, the data may still not be committed on the device upon `dd` completion.

We use Bonnie++ in our experiments: an IO benchmark tool suite that aims to perform several simple tests of hard drive and file system performance. It has 2 types of tests. The first is to test the IO throughput. The second is to test creation, reading, and deleting operations on many small files. We use the second type in our test.

AndroBench is a popular benchmark tool for Android. It is an Android application, which provides sequential/random r/w tests and SQLite benchmarks. Since Android uses SQLite as its built-in database for applications, the performance of SQLite is trivial for Android Apps user experience and is closely correlated with the storage performance.

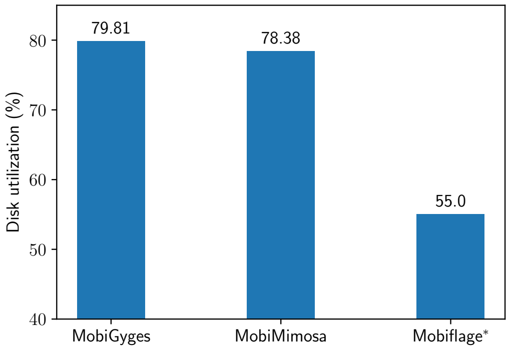

Figure 12: Disk utilization result of MobiGyges (1 hidden volume) and Mobiflage. Mobiflage* represents all the systems uses the same mechanism as Mobiflage.

7.2. Storage Utilization Evaluation

MobiGyges creates the Thin Pool using Equation 2 to calculate the size of metadata volume. Hence, the capacity of Thin Pool equals to the size of the data volume, so the size of the data volume is the actual capacity that allows users to store their data on the device. To this end, physical storage utilization η can be calculated with Equation 4. The definitions of the notations used in Equation 4 are the same as that of Equation 2 that can be found in Table 2.

$$\begin{aligned}\eta &= \frac{S_p}{S_p + S_m} \\ &= \frac{S_p}{\frac{S_p}{S_c} \times 64 + S_p} \times 100\% = 99.9024\%.\end{aligned} \tag{4}$$

We compare disk utilization of Mobiflage and MobiMimosa with MobiGyges because most of the related state-of-the-art works [8–10, 12] use the same method as Mobiflage. For all experiments, we use `dd` to conduct experiments, and each has 50 trials[4].

As shown in Figure 12, the average disk utilization of MobiGyges is 79.81%. The 20% loss is mainly because each Ext4 file system takes up about 10% of the storage for its metadata use [36, 37], and we have two volumes formatted with Ext4,

[4] Results are stable in the first 20 trials, and we conduct another 30 trials to decrease the errors.

one for the outer volume and the other for the hidden volume. Therefore, the storage space utilization of MobiGyges is 99.76%, which is equivalent to the theoretical result shown in Equation 4.

$$\begin{aligned} \mathit{Offset} &= \lfloor 0.75 \times \mathit{vlen} \rfloor \\ &\quad - (H(pwd||salt) \bmod \lfloor 0.25 \times \mathit{vlen} \rfloor). \end{aligned} \tag{5}$$

Mobiflage [7] proposes Equation 5 to calculate the offset of placing the hidden volume, and this equation indicates the capacity of the hidden volume is between 25-50% of the total disk capacity. Therefore, Mobiflage introduces up to 25% of the storage waste. Since each file system takes another 10% of the capacity for metadata, and there are two file system instances used (one used for the outer volume and the other used for the hidden volume), there is 20% of inevitable waste on each device. Thus, the total storage waste for Mobiflage is 45%. We want to get the improvement purely benefitting from our MobiGyges design, so we add 20% for each of the experimental results, and hence, MobiGyges increases the storage utilization by over 30%[5].

MobiMimosa [11] achieves storage utilization improvement by using the `dm_table` to record the storage block used by the hidden volume for the outer volume. Therefore, the only waste is the `dm_table`. MobiMimosa uses the Ext4 file system, and the default block size of the Ext4 file system on Android is 4KB. Consequently, if a hidden volume has 5GB capacity, the size of the `dm_table` file can be up to 64MB[6], which means the size of `dm_table` is 1.25% of the capacity of the hidden volume. Since the maximum size of a hidden volume is usually smaller than a half of the total physical disk capacity, the size of the `dm_table` file can take up to 0.625% of the total physical storage space. We test the storage utilization on Google Nexus 6P, with the same setting as MobiGyges. The results show that the overall storage utilization of MobiMimosa that has one 5GB hidden volume is 78.375%.

7.3. Performance Overhead Evaluation

Performance overhead on the PDE system is critical because the attack may compromise the PDE system if the performance overhead is significant. We evaluate the IO performance overhead of MobiGyges (Outer and Hidden in the figures) by comparing the IO performances of MobiGyges with that of Android FDE (Baseline in the figures) because Android FDE is enabled by default, and MobiGyges uses Android FDE as a foundation. In the rest of this subsection, we use different IO benchmark tools to evaluate the performance of the system and analyze the overhead.

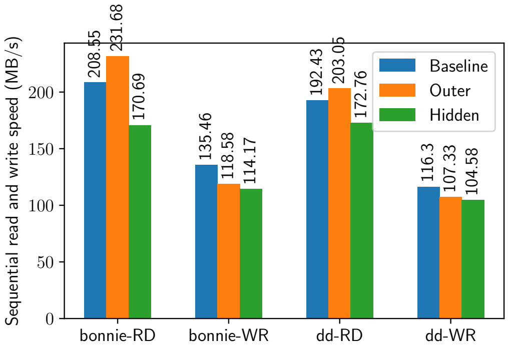

Figure 13: Bonnie++ and `dd` IO performance test. **Baseline** is the original Android FDE. **Outer** is the MobiGyges's outer volume, and **Hidden** is the MobiGyges's hidden volume.

(1) Bonnie++ test: We run 50 trials[7] of sequential block tests on a 6GB file[8] on each system. Figure 13 shows that the outer volume outperforms approximately 10% over Android FDE in terms of reading. The reason for MobiGyges's outer volume performs better is because of the IO request batching mechanism. When reading sequentially, the system can merge different IO requests, cache them, and read a bunch of adjacent storage blocks together [38]. However, the write performance is reduced by about 12%. We analyze the impact of the performance overhead by considering the user experience. With the same 1GB file, we calculate the time difference between MobiGyges and the baseline (Android FDE). Therefore, in terms of reading, MobiGyges reduces about 500ms compared with the original Android FDE. Similarly, MobiGyges takes 600ms longer than the original Android FDE. We believe the performance penalty is acceptable. Comparing the hidden volume and outer volume of MobiGyges, the write performance is only reduced by 3%. The reduction results from the extra encryption over the hidden volume.

[5] $((79.81\% + 20\%) - (55\% + 20\%))/55\% + 20\%) \approx 30\%$

[6] Each `dm_table` item is 80 byte.

[7] The results are very stable, we run 50 trials to minimize the error.

[8] Bonnie++ requires to test on a file whose size is twice as big as the device RAM to decrease the influence of system cache.

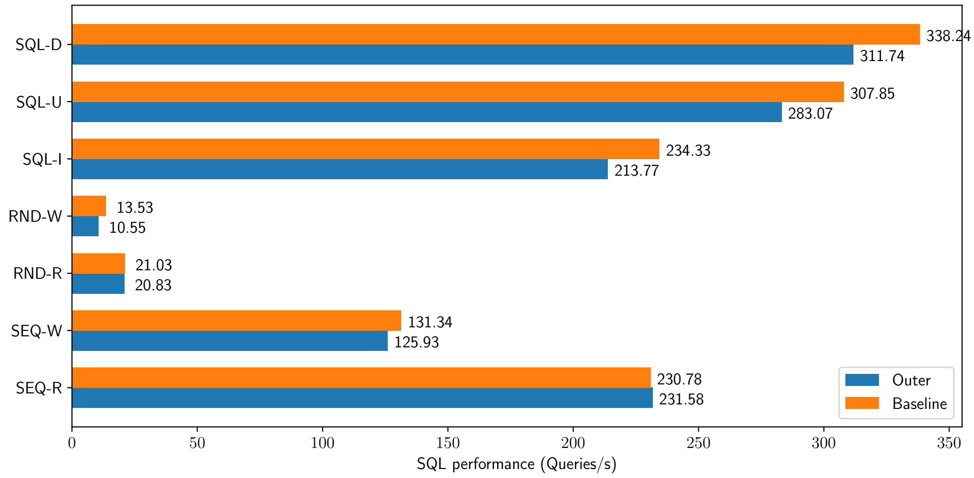

Figure 14: Sequential read and write speed and random read and write speed comparing between outer volume and original Android FDE. Tested with AndroBench. The unit of SQL operations is Q/s (query per seconds). **Outer** is the MobiGyges's outer volume and **Baseline** is the original Android FDE.

(2) `dd` test: We use the `dd` command to generate data from `/dev/zero` and write data on the volume with buffer size 600MB for once[9] to test writing performance of MobiGyges and the original Android system. Then, we use `dd` to read data from the volume and write the output to the `/dev/null` "blackhole"[10], to test the reading performance of MobiGyges and the original Android system. Normal `dd` version prints the statistical data after each command finishes. However, Android system uses the busybox [39] version `dd` tool that prints nothing after completion. Thus, we use `time` command to count the execution time. Moreover, we clear the cache in RAM before every tests[11], which eliminates the error caused by operating system caching mechanism. As shown in Figure 13, `dd` shows equivalent results as that of bonnie++.

(3) AndroBench test: Due to the permission control of AndroBench, we fail to test the performance of the hidden volume, but comparing the performance differences between the Android FDE and MobiGyges's outer volume is still sufficient to show the performance overhead of MobiGyges. Figure 14 depicts the result of the AndroBench tests. In the random r/w (RND-W and RND-R) and sequential write (SEQ-W) tests, MobiGyges has a performance penalty due to the Thin Pool and device mapping. In the sequential read (SEQ-R) test, MobiGyges outperforms the baseline because of merged IO requests and batch fetching mechanism. For SQL queries, MobiGyges reduces about 8% of the performance compared with the baseline. However, in terms of modifying data, which writes data on the volume, the performance penalty can be up to 22%. SQLite uses VDBE (Virtual Database Engine) as its background engine, and VDBE uses the B-Tree data structure to store data on a file system. VDBE also adopts the concept of paging as a unit to allocate space for the value of a key, which is similar to pages in the operating system virtual memory mechanism. The size of a page is fixed, so if the record is bigger than a page, it has to be stored into several pages that are linked together using pointers. To this end, it has to first read all the pages associated with the key before VDBE can finally update the record. Consequently, the multiple writes downgrade the performance of SQL queries, but We believe the performance penalty for the SQL operations is acceptable [40].

7.4. Performance Overhead Comparison

In this subsection, we compare the performance overhead between MobiGyges and some recent related works. Note that MobiGyges is not optimized for improving the IO performance. We post our IO performance overhead comparison experimental results here for the further optimization target.

[9] `dd if=/dev/zero of=disk.img bs=600M count=1 conv=fsync`

[10] `dd if=disk.img of=/dev/null bs=600M count=1`

[11] `echo 3 > /proc/sys/vm/drop_caches`

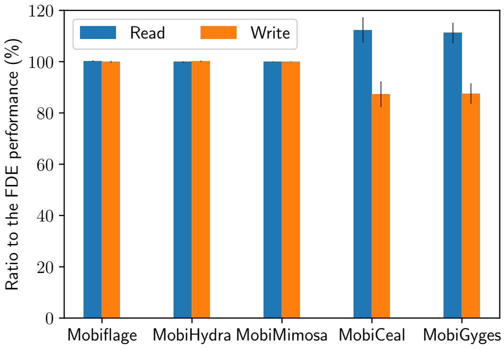

Figure 15: Performance overhead. the lower, the better. We use FDE (LUKS [19] on Linux) as the baseline.

We implement disk management part of Mobiflage [7], MobiHydra [8], MobiMimosa [11], MobiCeal [12] on Linux desktop PC and evaluate their performance with `dd`. The experimental results are shown in Figure 15, and the baseline system is the Android FDE[12]. Mobiflage, MobiHydra and MobiMimosa are native solutions, and their outer volume and the hidden volume are implemented in the same way as Android FDE. Hence, they have the equivalent performance to the Android FDE. MobiCeal and MobiGyges leverage storage virtualization mechanisms, which introduce performance overheads. MobiGyges outperforms MobiCeal on the performance overhead.

8. Discussion

In this section, we discuss counter-measurements of MobiGyges towards defending *common attacks*, and we then discuss the drawback and possible future works.

8.1. Security Discussion

Common attacks are identified by the existing literature and can be solved by existing counter-measurements. MobiGyge's mechanism of defending these attacks makes no difference to existing ones, and the discussion is to show that MobiGyges considers defending common attacks by design. Experiments of validating these mechanisms have been listed as part of our future works.

(1) Password guessing is the attack by trying all possible password characters repeatedly in a brute force way to identify the correct password. MobiGyges provides both salt and retrial timeout mechanism that can efficiently defend the password guessing attack and the rainbow table attack.

(2) Raw data parsing is conducted by parsing the physical disk raw data and try to reconstruct the files. MobiGyges uses FDE to defend the raw data parsing attack, in which all data are encrypted before committing to the physical disk.

(3) Encryption primitive leakage means the type of encryption or data protection is leaked. In the hidden volume based PDE, it refers to hidden volumes are exposed and is generally conducted from parsing the raw physical disk data. MobiGyges allocates fine-grained storage blocks from Thin Pool for the outer volume and hidden volumes on demands, which stores data in a striped way. Thus, attackers cannot tell the belonging of each data block and fail to distinguish from volume to volume. Moreover, MobiGyges applies FDE for Thin Pool, which increases the complexity and protects hidden volumes.

(4) Flash storage leakage refers to the NADN flash storage structure used by mobile devices could leak sensitive data because flash storage executes writing or wiping data in the unit of *page*, and can only change some bits in the page to 0 or change all bits in the page to 1. Hence, writing happens only on an empty page (with all 1s). Thus, data needs another *temporary* page for saving the current unchanged data in the original page. This temporary page can leak the sensitive data if not erased in time. MobiGyges addresses this problem with FDE, and data on flash pages is not the plain data but cipher data, which mitigates the flash storage leakage.

(5) Mobile carrier leakage represents the possible inconsistent record from the mobile device and the mobile carrier provider. The existing literature separate working modes into two, and PDE can only be used under the PDE mode. At the time using the PDE mode, carrier informations (e.g., phone call history, cellular traffic usage) are stored on the hidden volume, and attackers will find that these informations recorded by the carrier provider are more than that recorded by the outer volume.

[12] We implement Android FDE by using LUKS on Linux.

Thus, this is prone to expose hidden volumes. MobiGyges eliminates mobile carrier leakage by removing the design of two modes, and all the informations are stored on the outer volume. Thus, carrier information records are consistent between the carrier provider and the outer volume, which mitigates the leakage.

8.2. Drawbacks and Future Works

Although MobiGyges brings new hope to the PDE community, it falls short when the attacker can record the fill-to-full attack data and tries to retrieve the filled data from the device. One possible remedy is to employ data compression techniques to mitigate this issue. MobiGyges also opens up several interesting directions for future research. For example, theoretically evaluating the mathematically model of MobiGyges and existing works, reducing the performance overhead by replacing the Linux Thin Provisioning module with newly proposed high-performance Thin Provisioning tools (e.g., ThinStore [41]), and conducting further experiments in terms of defending common attacks (e.g., rainbow table attack).

9. Conclusion

This paper has presented MobiGyges, a PDE system that addresses the problems of data loss, storage waste, and device reboot on existing PDE systems by splitting the physical storage into fine-grained storage blocks, and each storage block is used only by one volume, and using Dynamic Mounting service to mount the hidden volume without rebooting the device. We have also identified the *capacity comparison attack* and *fill-to-full attack* targeted at PDE systems, and MobiGyges can jointly leverages the *Shrunk U-disk method* and *multi-level deniability* to defend them. We have implemented a proof-of-concept system on LineageOS 13 for real mobile devices. Experimental results show that MobiGyges achieves over 30% storage utilization improvement with an acceptable performance overhead.

Acknowledgement

This work was supported in part by the National Key Research and Development Program of China under Grant 2018YFB1003804, Natural Science Foundation of China under Grant 61921003, and the China Scholarship Council. We would also like to thank the editors and anonymous reviewers for their valuable comments and suggestions.

References

[1] Q. Zhang, G. Wang, J. Chen, G. B. Giannakis, Q. Liu, Mobile energy transfer in internet of things, IEEE Internet of Things Journal 6 (5) (2019) 9012–9019. `doi:10.1109/JIOT.2019.2926333`.

[2] F. Al-Turjman, Intelligence and security in big 5g-oriented iont: An overview, Future Generation Computer Systems 102 (2020) 357–368.

[3] M. Xiong, Q. Liu, G. Wang, G. B. Giannakis, C. Huang, Resonant beam communications: Principles and designs, IEEE Communications Magazine 57 (10) (2019) 34–39. `doi:10.1109/MCOM.001.1900419`.

[4] J. Götzfried, T. Müller, Analysing android's full disk encryption feature., JoWUA 5 (1) (2014) 84–100.

[5] Android Open Source Project, Full-disk encryption, `https://source.android.com/security/encryption/full-disk/`, [Online; accessed 28-Aug-2019] (2019).

[6] R. Canetti, C. Dwork, M. Naor, R. Ostrovsky, Deniable encryption, in: Annual International Cryptology Conference, Springer, 1997, pp. 90–104.

[7] A. Skillen, M. Mannan, Mobiflage: Deniable storage encryptionfor mobile devices, IEEE Transactions on Dependable and Secure Computing 11 (3) (2014) 224–237.

[8] X. Yu, B. Chen, Z. Wang, B. Chang, W. T. Zhu, J. Jing, Mobihydra: Pragmatic and multi-level plausibly deniable encryption storage for mobile devices, in: International conference on information security, Springer, 2014, pp. 555–567.

[9] B. Chang, Z. Wang, B. Chen, F. Zhang, Mobipluto: File system friendly deniable storage for mobile devices, in: Proceedings of the 31st annual computer security applications conference, ACM, 2015, pp. 381–390.

[10] B. Chang, Y. Cheng, B. Chen, F. Zhang, W. T. Zhu, Y. Li, Z. Wang, User-friendly deniable storage for mobile devices, Computers & Security 72 (2018) 163–174.

[11] S. Hong, C. Liu, B. Ren, Y. Huang, J. Chen, Personal privacy protection framework based on hidden technology for smartphones, IEEE Access (2017).

[12] B. Chang, F. Zhang, B. Chen, Y. Li, W. T. Zhu, Y. Tian, Z. Wang, A. Ching, Mobiceal: Towards secure and practical plausibly deniable encryption on mobile devices, in: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2018, pp. 454–465.

[13] J. John, C. Raju, Design and comparative analysis of mobile computing software framework, in: 2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT), IEEE, 2018, pp. 1639–1644.

[14] L. Catuogno, H. Löhr, M. Winandy, A.-R. Sadeghi, A trusted versioning file system for passive mobile storage devices, Journal of Network and Computer Applications 38 (2014) 65–75.

[15] Team, TrueCrypt, Truecrypt-free open-source disk encryption software for windows vista/xp, mac os x, and linux, sept 2019, `http://www.truecrypt.org/`.[accessed 20 Sept-2019].

[16] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, B. Schneier, Defeating encrypted and deni-

able file systems: Truecrypt v5. 1a and the case of the tattling os and applications., in: HotSec'08, 2008.

[17] The Windows Team, Bitlocker drive encryption overview, https://technet.microsoft.com/en-us/library/cc732774(v=ws.11).aspx (2019).

[18] N. Kumar, V. Kumar, Bitlocker and windows vista (2008).

[19] C. Fruhwirth, Luks–linux unified key setup (2009).

[20] X. OS, About filevault 2, Apple inc. Viitattu 22 (2014).

[21] M. J. Dworkin, Recommendation for block cipher modes of operation: The xts-aes mode for confidentiality on storage devices, Tech. rep. (2010).

[22] A. D. McDonald, M. G. Kuhn, Stegfs: A steganographic file system for linux, in: International Workshop on Information Hiding, Springer, 1999, pp. 463–477.

[23] A. Barker, S. Sample, Y. Gupta, A. McTaggart, E. L. Miller, D. D. E. Long, Artifice: A deniable steganographic file system, in: 9th USENIX Workshop on Free and Open Communications on the Internet (FOCI 19), USENIX Association, Santa Clara, CA, 2019.
URL https://www.usenix.org/conference/foci19/presentation/barker

[24] S. Dean, Freeotfe, https://en.wikipedia.org/wiki/FreeOTFE/.[accessed 18 Sept-2019].

[25] W. Feng, C. Liu, B. Ren, B. Cheng, J. Chen, Trustgyges: A hidden volume solution with cloud safe storage and TEE, in: Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'18), 2018, p. 511.

[26] S. Jia, L. Xia, B. Chen, P. Liu, Deftl: Implementing plausibly deniable encryption in flash translation layer, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, ACM, 2017, pp. 2217–2229.

[27] B. Leiba, Oauth web authorization protocol, IEEE Internet Computing 16 (1) (2012) 74–77. doi:10.1109/MIC.2012.11.

[28] A. Narayanan, V. Shmatikov, Fast dictionary attacks on passwords using time-space tradeoff, in: Proceedings of the 12th ACM conference on Computer and communications security, ACM, 2005, pp. 364–372.

[29] jthornber, Thin provisioning tools, https://github.com/jthornber/thin-provisioning-tools (2019).

[30] Y. Shao, X. Luo, C. Qian, Rootguard: Protecting rooted android phones, Computer 47 (6) (2014) 32–40.

[31] K. Yaghmour, Embedded Android: Porting, Extending, and Customizing, " O'Reilly Media, Inc.", 2013.

[32] E. Miles, E. Viola, The advanced encryption standard, candidate pseudorandom functions, and natural proofs, in: Electronic Colloquium on Computational Complexity (ECCC), 2011, p. 226.

[33] Wikipedia, dd (unix), https://en.wikipedia.org/wiki/Dd_(Unix). [accessed 20 Sept-2019].

[34] Coker, Russell, Bonnie++ file-system benchmark, http://www.coker.com.au/bonnie+ (2019).

[35] J.-M. Kim, J.-S. Kim, Androbench: Benchmarking the storage performance of android-based mobile devices, in: Frontiers in Computer Education, Springer, 2012, pp. 667–674.

[36] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, L. Vivier, The new ext4 filesystem: current status and future plans, in: Proceedings of the Linux symposium, Vol. 2, 2007, pp. 21–33.

[37] T.-Y. Chen, Y.-H. Chang, S.-H. Chen, N.-I. Hsu, H.-W. Wei, W.-K. Shih, On space utilization enhancement of file systems for embedded storage systems, ACM Transactions on Embedded Computing Systems (TECS) 16 (3) (2017) 83:1–83:28.

[38] Linux Kernel Organization, The kernel documentaion of thin provisioning, https://www.kernel.org/doc/Documentation/device-mapper/thin-provisioning.txt, [Online; accessed 7-Aug-2019] (2019).

[39] Erik Andersen, Busybox, https://busybox.net. [accessed 20 Dec-2019].

[40] N. Tolia, D. G. Andersen, M. Satyanarayanan, Quantifying interactive user experience on thin clients, Computer 39 (3) (2006) 46–52.

[41] K. Qian, L. Yi, J. Shu, Thinstore: Out-of-band virtualization with thin provisioning, in: 2011 IEEE Sixth International Conference on Networking, Architecture, and Storage, 2011, pp. 1–10. doi:10.1109/NAS.2011.39.