2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

MobiCeal: Towards Secure and Practical Plausibly Deniable Encryption on Mobile Devices

Bing Chang*, Fengwei Zhang†, Bo Chen‡, Yingjiu Li*, Wen-Tao Zhu§, Yangguang Tian*,
Zhan Wang¶ and Albert Ching‖
*School of Information Systems, Singapore Management University, {bingchang, yjli, ygtian}@smu.edu.sg
†Department of Computer Science, Wayne State University, fengwei@wayne.edu
‡Department of Computer Science, Michigan Technological University, bchen@mtu.edu
§Data Assurance and Communications Security Research Center, Chinese Academy of Sciences, wtzhu@ieee.org
¶RealTime Invent, Inc. ‖i-Sprint Innovations

***Abstract*—We introduce *MobiCeal*, the first practical Plausibly Deniable Encryption (PDE) system for mobile devices that can defend against strong coercive multi-snapshot adversaries, who may examine the storage medium of a user's mobile device at different points of time and force the user to decrypt data. MobiCeal relies on "dummy write" to obfuscate the differences between multiple snapshots of storage medium due to existence of hidden data. By incorporating PDE in block layer, MobiCeal supports a broad deployment of any block-based file systems on mobile devices. More importantly, MobiCeal is secure against side channel attacks which pose a serious threat to existing PDE schemes. A proof of concept implementation of MobiCeal is provided on an LG Nexus 4 Android phone using Android 4.2.2. It is shown that the performance of MobiCeal is significantly better than prior PDE systems against multi-snapshot adversaries.**

***Index Terms*—Plausibly Deniable Encryption, Mobile Security, Multi-snapshot Adversary, Side Channel Attack, Fast Switching**

I. Introduction

Mobile devices play an increasingly important role in our daily life and are prevalently used for processing sensitive information (e.g., by professional journalists or human rights workers). However, traditional encryption does not work in certain situations where the device owner is captured by an adversary and is coerced to disclose the key for decrypting the sensitive information on the device. To defend against such adversaries, various plausibly deniable encryption (PDE) systems have been proposed recently [2], [15], [21], [32], [33], [34], [27]. The existing PDE systems for mobile devices (e.g., [21], [34], [35], [43], [27], [20]) work correctly under the assumption that an adversary examines the storage medium once only on a user's device. However, they may not work if an adversary can take multiple snapshots of the storage medium at different points of time. In practice, such multi-snapshot attacks have been reported and thus posed realistic threats to users. For example, the Guardian [37] and the NBC News [30] have reported that US border agents not only demand travelers that they hand over their phones and their passwords, but also make full copies of all of the data on the phones, without any warrant or even suspicion. Another example is that an independent journalist was reported to have all of his computers, mobile phones and camera flash drives searched and copied when he was crossing a border, and he was inspected for seven times during five years [26].

The existing PDE systems on mobile devices [21], [34], [35], [43], [27], [20] are not resilient against such multi-snapshot attacks since they hide sensitive data in the randomness initially filled across the entire disk. By comparing storage snapshots at different points of time, a multi-snapshot adversary may detect any unaccountable changes to the randomness. Another drawback of these PDE systems is that users are required to reboot their mobile devices before using PDE functions. In emergency, users may miss the best moments since the rebooting process is usually time consuming.

It is challenging to design a secure and practical PDE scheme for mobile devices. All existing PDE systems that can defend against multi-snapshot adversaries [15], [19], [32], [33] are not suitable for mainstream mobile devices due to the following challenges.

1) The PDE scheme should be resistant to strong multi-snapshot adversaries on *resources-limited* mobile devices, making it unsuitable to transplant existing approaches available for desktop computers, e.g., HIVE [15] and DataLair [19], to mobile devices. Both HIVE and DataLair rely on a special "write-only oblivious RAM" to obfuscate all write access to the storage medium, such that no multi-snapshot adversary can identify any unaccountable changes to the storage medium at different points of time. Unfortunately, oblivious RAM is known for its poor I/O performance which is not suitable for the resources-limited mobile devices. HIVE/DataLair is designed with the assumption that an adversary can obtain snapshots after every single write operation is performed on the disk, so it needs complicated mechanisms to defend against such a strong adversary. We consider a more practical "on-event" adversary who can obtain multiple snapshots after the user is prepared (e.g., at border checkpoint). This more realistic adversarial model enables us to design a lightweight PDE scheme that is suitable for mobile devices.

2) The PDE scheme should be free from *side channel attacks* [23] which pose a serious threat to security of existing PDE schemes. Both HIVE [15] and DEFY [33] are subject to side channel attacks [23]. The major reason is that they do not isolate hidden data from public data sufficiently, so the

2158-3927/18/$31.00 Â©2018 IEEE
DOI 10.1109/DSN.2018.00054

information of the hidden data may be recorded in the public data. As a result, a multi-snapshot adversary may easily learn the existence of hidden data by analyzing the public data.

3) The PDE scheme should be fit for mainstream mobile devices, benefiting the deniability and *large-scale deployments*. DEFY [33] is specifically designed for mobile devices against multi-snapshot adversaries. However, DEFY heavily relies on the special properties provided by flash file system YAFFS [40]. DEFY is not immediately applicable to other flash file systems such as JFFS, UBIFS, and F2FS due to its strong coupling with YAFFS. In addition, a flash file system usually requires direct access to raw NAND flash, which is rarely supported in mainstream mobile devices since they usually use NAND flash as block devices through flash translation layer (FTL). Steganographic file system [32] is originally designed for desktop computers to defend against multi-snapshot adversaries. However, it heavily relies on the legacy Linux kernel and specific APIs for handling hidden files, and is thus not applicable to modern mobile devices.

4) The *usability* of the PDE scheme should be well treated, so that users can easily deal with sensitive data. Some existing designs [15], [33] do not provide details about how to use the system appropriately, but a wrong operation may lead to severe information leakage. Other designs [34], [21] require users to reboot their devices so as to switch between public mode (i.e., a mode in which the user can process public non-sensitive data) and hidden mode (i.e., a mode in which the user can process hidden sensitive data). The rebooting process is usually time-consuming and may thus lead to missing the best timing of collecting sensitive data.

These challenges motivate us to design *MobiCeal*, the first secure and practical PDE system on mainstream mobile devices that can defend against coercive multi-snapshot adversaries. MobiCeal relies on several key insights. First, we devise a "dummy write" mechanism to defend against multi-snapshot attacks. With dummy writes, any changes to the hidden data become accountable for the denial of the existence of hidden data in the presence of multi-snapshot adversaries. Second, MobiCeal is designed to be secure against side channel attacks [23]. The public data and the hidden data are strictly isolated in the system, eliminating the possibility of information leakage. Third, we decouple our design from both upper layers (e.g., file systems) and lower layers (e.g., storage media) to make it file system friendly and fit for mainstream mobile devices. Last, to improve usability of MobiCeal, we add a support for fast switching to help users switch from public mode to hidden mode. Prior PDE systems [21], [34] require users to reboot their devices to switch modes, which may take more than one minute in practice. The switching time in MobiCeal is less than 10 seconds, which is made possible by restarting Android framework instead of rebooting the device.

Contributions. The major contributions of this paper are two-fold. First, we design the first secure PDE system for mobile devices against multiple snapshot adversaries. A formal proof shows that MobiCeal provides reliable deniability against multi-snapshot adversaries. MobiCeal is also shown to be free from side channel attacks, which pose a serious threat to many other PDE systems.

Second, MobiCeal is practical to be implemented on mainstream mobile devices. MobiCeal is built into the block layer of Linux kernel such that any block file systems can be deployed on top of it. MobiCeal relies on a lightweight "dummy write" mechanism to defend against the multi-snapshot adversary, which introduces an acceptable performance overhead, making it suitable for resources-limited mobile devices. In addition, MobiCeal is easy to use, and supports fast switching from its public mode to hidden mode.

A proof-of-concept implementation of MobiCeal is provided on an LG Nexus 4 Android phone using Android 4.2.2, and an availability test is conducted on a Huawei Nexus 6P phone using Android 7.1.2. Compared to the default Android full disk encryption, MobiCeal introduces approximately 18% overhead which is much smaller than that of typical prior PDE systems secure against multi-snapshot adversaries.

II. Background

A. Full Disk Encryption

A full disk encryption (FDE) system encrypts the entire disk with a key to prevent unauthorized access to the data. FDE is usually transparent to the upper layer as the data are automatically encrypted or decrypted upon being written or read. BitLocker [1] and FileVault [12] are two popular FDE tools. FDE has been available on Android to encrypt userdata partition since version 3.0 [4]. Android FDE is based on *dm-crypt* [16], a Linux kernel module working in the block device layer. In Android, the block devices (e.g., an eMMC card [9] that is presented to the kernel as a block device) can be encrypted by *dm-crypt* which creates an additional layer of "encrypted block device" over the original block device.

To enable Android FDE, a user should choose a secret password first. Android uses a randomly generated master key to encrypt the entire disk using *dm-crypt*, and the master key is encrypted with a key derived from the password using `PBKDF2` [14]. Note that `PBKDF2` also needs a salt which is randomly generated. The encrypted master key and the salt are stored in the encryption footer that is located in the last 16KB of the userdata partition. When Android boots and detects that the userdata partition is encrypted, it asks for a password. After having obtained the password, it derives a key from the password using `PBKDF2` with the salt read from the encryption footer. It then decrypts the master key and passes the master key to *dm-crypt*, who can then decrypt the entire disk.

B. Plausibly Deniable Encryption

Canetti et al. [17] initially explored plausibly deniable encryption (PDE) to protect the confidentiality of messages transmitted over networks against a coercive attacker. When being applied to storage domain, there are two main types of PDE techniques: *hidden volumes* [2], [34], [21], [15] and *steganographic file systems* [13], [29], [32].

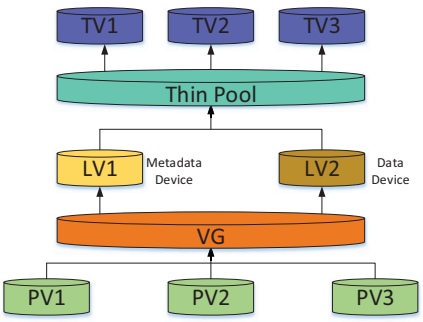

Fig. 1. LVM and thin provisioning architecture.

The hidden volume technique works as follows: There are two encrypted volumes on the disk, a public volume and a hidden volume. The public volume is encrypted by a decoy key and the hidden volume is encrypted by a hidden key, both using full disk encryption. The public volume is placed on the entire disk and the hidden volume is usually placed from a secret offset towards the end of the disk. Note that initially the entire disk is filled with random data and the data written to the public volume should be placed sequentially from the beginning of the disk so as to avoid over-writing the hidden volume. When a user is coerced to reveal the encryption key, he/she can disclose the decoy key. If the attacker is unable to confirm the existence of the encrypted hidden volume, the existence of the hidden data can be denied. This mechanism works when the attacker can only access the disk once, e.g., obtaining the disk after having seized the user. However, it is problematic if the attacker periodically obtains snapshots of the disk, because the attacker can detect changes in the "free" space of the public volume by comparing snapshots taken at different time, and suspect existence of hidden data [23].

The other type of PDE technique is based on **steganographic file systems** whose idea is to hide sensitive data among regular file data. This can be achieved by introducing a large number of cover files [13] or hiding the data into abandoned/dummy file blocks [29], [32]. The main concern of the steganographic file system is to avoid over-writing the hidden sensitive data, which requires creating a large amount of redundancy, leading to inefficient use of disk space.

C. LVM and Thin Provisioning

Logical volume manager (LVM) [28] is a userspace toolset that provides logical volume management capabilities on Linux. LVM is a device mapper [11] target which becomes a component of the Linux kernel since version 2.6. LVM creates a layer of abstraction over physical storage, allowing users to create logical storage volumes. LVM introduces three concepts: physical volumes (PV), volume groups (VG), and logical volumes (LV). The underlying physical storage, such as a partition or the entire disk, can be initialized as a physical volume. Physical volumes are combined into volume groups. A volume group can be divided into logical volumes. In Android, LVM has gained popularity for flexibly handling internal and external storage [38], [24].

Thin provisioning [39] has become a feature in the Linux kernel since version 3.2. Thin provisioning is different from the conventional provisioning known as "thick provisioning". In thick provisioning, storage administrators usually need to plan ahead, and install more storage capacity than required to avoid any potential failures caused by inadequate storage. In thin provisioning, only the logical storage space is allocated to a volume and the physical storage capacity is not released until it is actually required. This "on-demand" storage eliminates the need of installing unnecessary storage media.

The volumes provided by thin provisioning are called "thin volumes" (TV). Two logical volumes are needed when using thin provisioning: one for data device and the other for metadata device. The data device contains data blocks of the thin volumes while the metadata device contains the free space bitmap and the block mappings for the thin volumes. Two device mapper targets are provided by the *dm-thin-pool* module, thin-pool and thin. The metadata device and the data device are mapped to a pool device by the thin-pool target, while the thin target maps this pool device to multiple thin volumes (See Fig. 1). On top of a thin volume, a block-based file system can be deployed or an encrypted block device can be created using *dm-crypt*.

III. Model and Assumptions

A. Adversarial Model and Assumptions

We consider a computationally bounded adversary who can take snapshot of the block device storage (e.g., eMMC card, SD card, which are usually exposed as block devices using flash translation layer) of a mobile device at different points of time [23], [15], [32], [33]. For example, when the device owner enters/exits a guarded facility or crosses border, the observer takes a snapshot of the mobile device storage and tries to compromise deniability by analyzing the snapshots. The adversary can have full knowledge of MobiCeal's design, but should not know the hidden password or the encryption key of the hidden volume. The adversary can obtain root privilege of the device and can access both the internal and external storage each time when capturing it. In addition, it can coerce the device owner to reveal the passwords/encryption keys, in order to decrypt the device to obtain sensitive data. Furthermore, the adversary can use any password cracking programs, perform advanced computer forensics on the disk image, or correlate different snapshots to compromise deniability.

Similar to all the prior PDE systems for mobile devices [21], [33], [43], our design also relies on the following assumptions: The adversary is assumed to be not able to capture the device owner when he/she is working with the hidden volume. Otherwise, the sensitive hidden data will be trivially disclosed. We also assume that the adversary will not continue coercing the device owner once being convinced that the passwords or encryption keys have been revealed. The user should refrain from revealing the hidden passwords/hidden keys as disclosing hidden sensitive data will create life threat to him/her. Furthermore, MobiCeal needs to be merged with Android code stream, so that its availability itself is not a

red flag [21], [34], [35], [43], [20]. The mobile OS, kernel, bootloader, firmware, and the baseband OS are malware-free, and the user does not use any malicious apps that collect information of the hidden volume or the hidden password.

B. Notations

We assume that there exists a sequence of independent volumes $\{\mathcal{V}_i\}, i \in [1, max]$ on a hard disk, where max denotes the maximal number of volumes. To store information into public or hidden volumes, each user needs to choose a set of passwords $\{\mathcal{P}_i\}$ which in turn serves as encryption keys. Each volume $\mathcal{V}_i$ has a unique password $\mathcal{P}_i$, and each volume has at most $n_i \in \mathbb{N}$ blocks of data, where $\mathbb{N}$ denotes the number of blocks. Note that we allow users to choose a *secret* number of volumes $l \in max$. A volume encryption scheme consists of the following Setup, Read and Write operations on the disk.

- Setup. It takes $(\lambda, t, \mathcal{P}, \mathcal{B}, [n_1, \cdots, n_l])$ as input and outputs a sequence of volumes $\{\mathcal{V}_1, \cdots, V_l, \cdots, \mathcal{V}_{max}\}$. Note that λ denotes the security parameter, t denotes the number of available blocks, and $\mathcal{B}$ denotes the block size.
- Read. It takes $(b, i, \mathcal{P})$ as input, returns data d in block b of volume $\mathcal{V}_i$ if $i \leq l$. Note that $\mathcal{V}_i$ is the output of Setup$(\lambda, \mathcal{P})$.
- Write. It takes $(b, d, i, \mathcal{P})$ as input, stores data d in block b of volume $\mathcal{V}_i$ if $i \leq l$. Note that $\mathcal{V}_i$ is the output of Setup$(\lambda, \mathcal{P})$.

C. Security Model

Informally, a coercive multi-snapshot adversary $\mathcal{A}$ attempts to obtain any stored data from a hard disk that employs a hybrid [1] volume system. We then formally define a multi-snapshot security game between a Probabilistic Polynomial-Time (PPT) adversary $\mathcal{A}$ and a simulator $\mathcal{S}$ as follows.

- Setup. Upon receiving a required volume number $l \in max$ from $\mathcal{A}$, $\mathcal{S}$ first chooses a set of passwords $\{\mathcal{P}_i\} \in \{0,1\}^{|\lambda|}$, where $i \in [1, \cdots, l]$. Second, $\mathcal{S}$ initializes a set of public volume $\{\mathcal{V}_w\}$ that associates with a set of passwords $\{\mathcal{P}_w\}$ $(w \in [1, \cdots, k], k \leq max - l)$. Third, $\mathcal{S}$ generates two independent hybrid volume encryption schemes Σ_0 and Σ_1 with respect to two sets of passwords with size l and $l-1$ respectively. Eventually, $\mathcal{S}$ sends two sets of passwords $\{\mathcal{P}_w\}$ and $\{\mathcal{P}_1, \cdots \mathcal{P}_{l-1}\}$, and an initial snapshot $\mathcal{D}_0$ of the disk to $\mathcal{A}$. $\mathcal{S}$ also tosses a random coin b which will be used later in the game. Note that the initial snapshot $\mathcal{D}_0$ is taking on two sets of volumes $\{\mathcal{V}_1, \cdots, \mathcal{V}_w\}, \{\mathcal{V}_1, \cdots, \mathcal{V}_l\}$.
- Training. $\mathcal{A}$ chooses two access patterns $(\mathcal{O}_{0,i}, \mathcal{O}_{1,i})$ and a value d_i, and sends them to $\mathcal{S}$. Note that the value d_i specifies whether $\mathcal{A}$ would like a snapshot of the disk after execution i. $\mathcal{S}$ "executes" one of access patterns based on bit b, and sends a snapshot $\mathcal{D}_i$ of the disk to $\mathcal{A}$ if $d_i = 1$; Otherwise, proceed to next execution.

[1] "Hybrid" means that a disk may consist of public and hidden volumes.

$$\mathcal{A} \leftarrow \mathcal{D}_i \leftarrow \begin{cases} Execute_{\Sigma_1}(\mathcal{O}_{1,i}) & b = 1 \\ Execute_{\Sigma_0}(\mathcal{O}_{0,i}) & b = 0 \end{cases}$$

Note that $\mathcal{O}_{j,i}$ denotes the access pattern $j \in [0,1]$ of execution $i \in [1, poly(\lambda)]$. We allow $\mathcal{A}$ to obtain snapshots with on-event frequency[2], and we specify the restrictions of access patterns (e.g., $\mathcal{O}_0 = [o_{0,1}, \cdots, o_{i,n}]$) as follows.

  - If access $o_{0,i}$ is a Read/Write in volume $\mathcal{V}_j, j \in [0, l-1]$, then access $o_{1,i}$ in pattern $\mathcal{O}_1$ must be *equal* to $o_{0,i}$;
  - If access $o_{0,i}$ contains a Read/Write in volume $\mathcal{V}_j, j \in [0, l]$, then *at least* one public volume $\mathcal{V}_w$ is randomly *refreshed* after each execution i;
  - If access $o_{0,i}$ indeed contains a Read/Write in volume $\mathcal{V}_l$, then operations can be *plausibly* applied to one of public volumes $\{\mathcal{V}_1, \cdots \mathcal{V}_w\}$.
- Guess. $\mathcal{A}$ outputs bit b'. If $b' = b$, then $\mathcal{S}$ outputs 1; Otherwise, $\mathcal{S}$ outputs 0.

We define the advantage of $\mathcal{A}$ in the above game as

$$\mathtt{Adv}_{\mathcal{A}}(\lambda) = |\Pr[\mathcal{S} \rightarrow 1] - 1/2|.$$

Definition III.1. *We say a hybrid volume encryption scheme* Σ *has* multi-snapshot security *if for any PPT* $\mathcal{A}$, $\mathtt{Adv}_{\mathcal{A}}(\lambda)$ *is a* negligible *function of the security parameter* λ.

IV. MobiCeal Design

A. Design Overview

The existing hidden volume-based PDE systems for mobile devices [21], [34], [35], [43], [20], [27] cannot defend against a multi-snapshot adversary. This is because, they hide sensitive data among randomness being filled initially across the disk. However, by comparing snapshots being captured at different points of time, the multi-snapshot adversary can easily detect changes over randomness which were not supposed to happen, and may suspect existence of hidden data. A fundamental limitation of the hidden volume-based approach is that, the randomness is filled one time initially (i.e., *static*), which is definitely not able to defend against the multi-snapshot adversary, who is "*dynamic*".

To defend against such a dynamic attacker, the intuition is to also make the defense dynamic. A few existing PDE schemes followed this idea by incorporating ORAM [15], [19], in which each single write is turned to be oblivious to the adversary. All those attempts, however, are problematic, due to the following reasons. First, ORAM is prohibitively expensive in terms of both computation and I/O [15], making it unsuitable for mobile devices that are equipped with limited resources. Second, we found *all those ORAM-based PDEs rely on an assumption that protecting every access pattern is*

[2] Adversary is allowed to have plausible hidden access pattern choice with on-event frequency snapshots in our proposed multi-snapshot security model for hybrid volume encryption schemes. Please refer to [15] for detailed description of these settings.

necessary for mitigating a multi-snapshot adversary. ORAM was originally designed to hide access pattern over data being outsourced to an untrusted third party (e.g., a cloud provider). In this setting, the cloud provider is able to constantly monitor access of the data (i.e., *highly dynamic*) due to its full control over the data during the lifetime. In a mobile device setting however, the adversary does not have a full control over the victim's mobile device during its lifetime, and is thus not able to constantly monitor each access (i.e., *less dynamic*). Therefore, we believe that hiding every access is unnecessary for mobile devices and the ORAM-based approach is overkill for the less dynamic attacker in the mobile device setting. Another PDE system for mobile devices, DEFY [33], was designed for a less dynamic attacker, but it strongly relies on the system properties provided by a specific flash file system, and is shown to be vulnerable to deniability compromise [27].

To achieve deniability against a less dynamic attacker without relying on the expensive ORAM [15], [19] or specific system properties [33], we propose a dummy-write approach. Specifically, each time when writing public non-sensitive data, the system will perform a few additional artificial writes of randomness. In this way, although the adversary can obtain multiple snapshots, uncountable changes (i.e., caused by storing the hidden data) observed by the adversary through comparing snapshots can be denied as being caused by the dummy writes. Note that the hidden sensitive data should be encrypted using a secret key, such that without having access to the secret key, the encrypted hidden data should be indistinguishable from the randomness created by the dummy writes. A few questions still need to be answered.

1) *How many dummy writes should be performed for each public write?*

For a good obfuscation, the number of dummy writes being performed each time should vary. In our design, the number of dummy writes follows exponential distribution. We choose exponential distribution, since it can ensure that the number of dummy writes varies in a wide range and, meanwhile, the probability of generating a large number of dummy writes each time can be controlled as small to avoid inefficient I/O performance and disk utilization.

2) *How to generate the data for each dummy write?*

The dummy data are used to deny the existence of encrypted hidden sensitive data. Therefore, without having access to the decryption key, the adversary should not be able to differentiate the encrypted hidden data from the dummy data. To achieve this, the dummy data can be created using the same encryption algorithm (as the hidden data) with random input and random keys, and the corresponding key should be discarded after each encryption.

3) *How can the system prevent the public data from overwriting the hidden data?*

As the public mode has no knowledge on the existence of the hidden data, it may easily cause overwrites to them. We need a technique to ensure that newly written public data will not over-write the existing hidden sensitive data. The hidden-volume technique (Sec. II-B) addresses the over-write issue by placing the hidden volume to the end of the disk. However, such a technique is only suitable for file systems that perform writes sequentially on the storage media (e.g., FAT32) and over-writes are still possible when the disk is heavily used. Steganographic file systems (Sec. II-B) address this issue by utilizing the global bitmap in the file system to separate the hidden data from the public data. This, however, requires extensive modifications of the large code base of the file system being used, which contradicts our "file system friendly" design principle.

To resolve the over-write issue, we borrow the "global bitmap" idea of the steganographic file system, but move it to the block layer. The global bitmap will keep track of blocks being used by all the public, dummy, and hidden data. Therefore, when hidden data are written, the corresponding blocks in the bitmap will be marked as "allocated", and will not be used by public/dummy data. This will not lead to deniability compromise, since the bitmap information for the hidden data can be denied as for the dummy data.

4) *What other attacks the design is still vulnerable to?*

The current design is fine if the system always writes a small amount of hidden data occasionally. However, if the system writes a large hidden file, the adversary may observe from the snapshot that the public data are followed by a large amount of randomness[3], and may suspect existence of hidden sensitive information, compromising deniability. To avoid this deniability compromise, all the data (including public, dummy, and hidden data) should be written to random locations across the disk. Following the aforementioned ideas, we design a basic MobiCeal scheme which can defend against a multi-snapshot adversary (Sec. IV-B). We also extend the basic MobiCeal to support multiple levels of deniability (Sec. IV-C). In addition, we describe additional design considerations of MobiCeal (Sec. IV-D).

B. A Basic MobiCeal Scheme

We first introduce three types of virtual volumes:

(a) *Public volume.* A public volume is used for daily operations which provides storage encryption without deniability. The user can enter the decoy password during booting in order to use the public volume. The public volume is encrypted using a decoy key via FDE (Sec. II-A). The decoy key can be computed using the decoy password. When the user is coerced, he/she can simply disclose the decoy password, protecting the hidden sensitive data.

(b) *Hidden volume.* A hidden volume is used when the user needs to store sensitive data, whose existence needs to be denied when the user is coerced. The hidden volume is encrypted using a hidden key via FDE. The hidden key can be computed using the hidden password. The user can enter the hidden password during booting to use the hidden volume.

(c) *Dummy volume.* A dummy volume only stores data created by dummy writes. The purpose of the dummy volume is

[3]Writes performed by a file system (e.g., FAT and Ext4) usually exhibit a certain level of spatial locality.

to obfuscate the existence of the hidden volume. Without having access to the hidden key, the adversary is not able to differentiate whether a volume (which is not a public volume) is a hidden volume or a dummy volume. In this way, the user can deny the existence of the hidden volume by interpreting it as a dummy volume.

To ensure that a multi-snapshot adversary cannot distinguish the hidden volume from the dummy volume, we introduce the dummy write mechanism and the random allocation strategy. Note that the system keeps the metadata (e.g., the global bitmap, the mappings of each virtual volume and the corresponding blocks) in a known location and the adversary can have access to them. This will not compromise deniability, since the metadata for the hidden volume can be interpreted as that for the dummy volume.

Dummy Write. We use a dummy write mechanism to obfuscate writes to the hidden volume. When a data block is allocated to the public volume to store data (i.e., a public write is issued), a dummy write will be performed with a certain probability. To prevent the adversary from learning the pattern of dummy writes, the dummy write will be performed if and only if the following condition satisfies:

$$rand \leq stored\_rand \bmod x.$$

Here, x is a positive integer constant (e.g., we can fix x as 50 when initializing the system). $stored\_rand$ is a random number which is periodically updated (e.g., daily). To obtain a new value of $stored\_rand$, we can utilize pseudorandom number generator, or a more secure way is to extract it from the random noise present in mobile device hardware [41]. $rand$ is an integer chosen uniformly at random from 1 to $2 \cdot x$ upon each dummy write, to ensure that the probability of performing dummy write will be always under 50%.

When a dummy write is performed, m free blocks will be allocated and the corresponding blocks should be marked as "allocated" in the global bitmap. These blocks will be filled with random noise, which should be indistinguishable from the encrypted data (Sec. IV-A). m is determined as follows:

$$m = \lfloor m' \rfloor, m' = -(ln(1-f))/\lambda.$$

Here, f is a random number in the range of $(0, 1)$ and λ is the rate parameter, making m' follow exponential distribution. The mean value of m' is $1/\lambda$, e.g., if we choose λ as 1, each dummy write will be allocated one free block on average. The exponential distribution is advantageous since it can ensure that the value of m can have a large variance which is good for deniability.

Block Allocation Strategy in Block Layer. A common block allocation strategy in the block layer is sequential allocation, by which when data blocks are allocated to virtual volumes, they will be allocated sequentially from the disk (e.g., thin provisioning [21]). A concrete example for the sequential allocation is shown in the following:

$$D_{v_2}||D_{v_1}||D_{v_2}||D_{v_2}||D_{v_2}||D_{v_2}||D_{v_2}||D_{v_2}||D_{v_2}||D_{v_1}$$

Here, D_{v_1} means the data block allocated to the public volume (identified by v_1) and D_{v_2} means the data block allocated to the hidden volume (identified by v_2). From the aforementioned block layout, an adversary can observe that seven data blocks are allocated between D_{v_1}. To deny the existence of the hidden data, the user will claim that the seven data blocks have been allocated to dummy volumes. However, since the number of dummy writes associated with each public write will be limited, the adversary may observe that the number of dummy blocks being claimed by the user exceeds this limit (this is highly possible if a large file has been written to the hidden volume), and suspects the existence of hidden volume.

To avoid this deniability compromise, we use random allocation in the block layer. Specifically, each write from the upper layer (performed by the public or the hidden/dummy volume), should be allocated with an unused block at a random location. In this manner, the adversary will not be able to observe such a layout that a block, which has been allocated to the public volume, is followed by a large number of blocks being allocated to the hidden volume.

A potential deniability compromise remaining is that the adversary can calculate the total number of blocks for the public volume, and estimate the maximal number of blocks for the dummy volume. If the total number of blocks being allocated for non-public data exceeds this maximal number, the adversary may suspect existence of hidden data. This would happen if the user stores a very large file in the hidden volume and does not store enough data in the public volume. To mitigate this issue, we recommend that the user should store a file with approximately equal size in the public volume after storing a large file in the hidden volume. In practice, the sensitive data (e.g., secret documents, photos, short audio/video files) are usually small in size.

User Steps. If the user needs data encryption without deniability, he/she needs to enable device encryption with one password (e.g., through settings GUI). Note that before initializing the device encryption, the user should backup the data on the device since the initialization erases existing data. The system then creates a public volume (encrypted with a key derived from the password) and a dummy volume, and reboots when complete. The user can enter the password during pre-boot authentication to decrypt the device.

If the user requires the deniability feature, he/she needs to initialize the device with a decoy password and a hidden password. The system then creates a public volume and a hidden volume, encrypted with keys derived from the decoy password and the hidden password, respectively. For daily use, the user enters the decoy password during pre-boot authentication to activate the public volume. Note that we assume the user enables the screen lock when they are using the public volume, and the screen lock password is different from the hidden password. When the user wants to activate the hidden volume, he/she enters the hidden password in the screen lock (Sec. IV-D). The system closes the public volume, decrypts the hidden volume and enables the hidden volume.

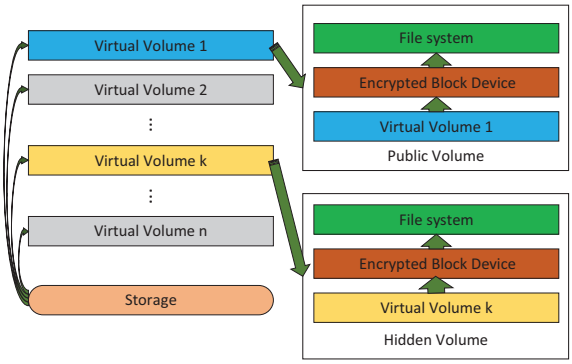

Fig. 2. System architecture of the extended MobiCeal scheme.

The user then can collect and store sensitive data. After that, the user should immediately reboot to use the public volume (Sec. IV-D). When the user is coerced by the adversary, he/she can supply the decoy password and claim that the other volume is a dummy volume. The adversary can examine the device but cannot distinguish a hidden volume from a dummy volume. If the user does not reveal the hidden password, the adversary will find no evidence of the hidden data.

C. An Extended MobiCeal Scheme Supporting Multi-level Deniability

To support multi-level deniability, the system creates n virtual volumes (by utilizing thin provisioning as introduced in Sec. II-C) initially, among which, there are n' hidden volumes ($n' < n$). Note that n' should be kept secret. After n virtual volumes (labeled as V_1, V_2, ..., V_n) are created, MobiCeal simply uses the first virtual volume V_1 as the public volume. The user can provide different hidden passwords to protect different hidden volumes and the number of hidden volumes is controlled by the number of hidden passwords.

For example, if virtual volume V_k ($2 \leq k \leq n$) is used as the hidden volume, k can be derived using the hidden password:

$$k = (H(pwd||salt) \bmod (n-1)) + 2.$$

Here, H is a `PBKDF2` [14] iterated hash function, n is the total number of virtual volumes, pwd is the hidden password and $salt$ is a random salt value for `PBKDF2`. The salt value will be stored in the encryption footer. If different hidden volumes result in the same k, another random $salt$ will be chosen. All the remaining virtual volumes are dummy volumes. Figure 2 shows the system architecture of MobiCeal.

When generating dummy writes, the system will assign them to a random virtual volume. The dummy write is assigned to V_j and j is generated as follows:

$$j = (stored\_rand \bmod (n-1)) + 2.$$

Storage Layout. The entire disk is divided into 3 parts, containing the metadata, data and encryption footer, respectively. The storage layout is shown in Figure 3. Specifically, the metadata part stores the information of virtual volumes, e.g, the global bitmap, the sizes and mappings of virtual volumes. The data part stores the data blocks for the virtual volumes while the encryption footer is a default part of Android. Note that in Android, the encrypted decoy key and the salt are stored in the encryption footer which is located in the last 16KB of the userdata partition.

D. Additional Design Considerations

Defending against Side Channel Attack. Existing PDE systems that defend against multi-snapshot adversaries like HIVE [15] and DEFY [33] suffer from the side channel attack [23]. Due to the shared OS, the information of the hidden files may be recorded in the public volume [23], leading to compromise of the deniability. However, our design can defend against this side channel attack, since we isolate the hidden volume from the public volume. Although the hidden password is entered in the public mode, the Android screen lock does not record the entered password and we assume the mobile OS, the boot-loader, as well as the firmware and the baseband OS are all malware-free (Sec. III). As a result, the security of the hidden password is ensured.

We consider four possible leakage paths for the side channel attack: 1) the public volume, 2) logs at */devlog*, 3) */cache* and 4) RAM. The information of the hidden volume or hidden files may be recorded in the public volume, */devlog*, or */cache*, if the hidden volume is in the system together with others. To prevent the leakage, after the hidden password is verified, the system unmounts these three partitions immediately, and mounts two tmpfs RAM disks to */devlog* and */cache*, respectively. Then the system decrypts the hidden volume and mounts it as the userdata partition. In this way, the information leakage is prevented.

Additionally, if the RAM is not cleared after the hidden mode is off, the deniability may be compromised. To prevent this threat, we only support fast switching from the public mode to the hidden mode. When the user wants to switch from the hidden mode to the public mode, he/she has to reboot the phone. In this way, the information of the hidden volume or hidden files in the RAM will be cleared. Note that the one-way fast switching is reasonable, since the mobile device is assumed to be usually in the public mode. When the user wants to switch to the hidden mode, he/she needs to enter the hidden password and the system will begin switching.

Switching without Rebooting. The existing PDE systems for mobile devices [21], [34], [43] all require rebooting to switch modes. However, if time is limited which does not allow a slow mode switching, the user may miss a best moment to capture sensitive information (e.g., an opportunistic sensitive photo). We propose a fast switching mechanism without rebooting the entire device. Our main concern is how to switch fast from the public mode to the hidden mode, without compromising deniability. We find it unnecessary to reboot the entire device. Instead, we can simply restart the Android framework. In this way, the switching time can be significantly reduced. We choose the default screen lock app of Android as the entrance of the hidden mode, because it is widely used and allows the user to enter the password conveniently.

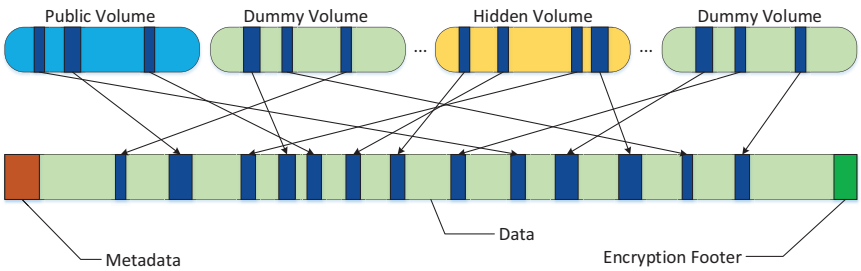

Fig. 3. Storage layout of the extended MobiCeal scheme.

The screen lock app runs as usual if the user does not enter the hidden password. When the user wants to switch to the hidden mode, he/she needs to enter the hidden password. After that, the system will unmount the public volume, decrypt and mount the hidden volume, switching to the hidden mode. Note that it is necessary to unmount the public volume. Otherwise, the traces of the hidden volume and hidden files may be leaked to the public volume which may lead to compromise of deniability. Since the public volume is mounted to "/data" and Android framework requires "/data" to run [7], we unmount the public volume by shutting down the Android framework.

Key Derivation. Different keys can be derived using `PBKDF2` with different passwords, and different master keys can be derived by decrypting the same random data stored in the encryption footer using these different keys. Further, each virtual volume can be encrypted using a different master key via *dm-crypt* (Sec. II-A).

Reclaiming Space Occupied by Dummy Writes. The data created by dummy writes will accumulate and may fill the entire disk space over time. This issue can be mitigated by periodically performing garbage collection, reclaiming part of the space occupied by dummy writes. Note that the garbage collection cannot reclaim all the space occupied by dummy data. Otherwise, the adversary can easily identify where the hidden data are by comparing snapshots. This is because the space occupied by the hidden data remains unchanged. As a result, when performing garbage collection, the system reclaims a random percentage of the space occupied by dummy writes. To make the garbage collection more efficient, the percentage should be large with a high probability. A side effect of this approach is, when performing garbage collection, the system may not be able to distinguish dummy data and the hidden data. This issue can be solved by performing garbage collection in the hidden mode[4].

V. MobiCeal Implementation

We implement a prototype of MobiCeal on an LG Nexus 4 phone using the 4.2.2 (Jelly Bean) Android source code and the 3.4 Linux kernel. *Note that we only implement/evaluate the extended MobiCeal scheme supporting multiple levels of deniability, since the basic MobiCeal scheme is a special case of MobiCeal with multi-level deniability support.* To allow creating multiple virtual volumes, we rely on thin provisioning (Sec. II-C), but modify it for PDE considerations. We also test MobiCeal on a Huawei Nexus 6P with Android 7.1.2 and Linux kernel 3.10. The transplant can be done with a little work on SEAndroid [36]. The source code of the implementation has been released[5]. There are three parts of implementation, including changes to 1) the Linux kernel, 2) Android volume daemon, and 3) Android screen lock. The implementation requires approximately one thousand lines of C and Java code. We also compile LVM and thin provisioning tools for Android and put them in the boot image.

A. Changes to the Linux Kernel

Tweaking Thin Provisioning. To implement the random allocation and the dummy write, we modify the thin provisioning target in the device mapper. We add the dummy write mechanism to thin provisioning and change the original sequential allocation strategy of thin provisioning to random allocation. The reasons why we choose thin provisioning are as follows: First, when the thin volumes are initialized, they do not really occupy disk space until the actual data are written to the thin volumes. This makes it cost effective to hide a thin volume that contains sensitive data among dummy thin volumes. Second, thin provisioning does not allocate data blocks for a thin volume until the data are written to it. This feature helps us to realize a dummy write mechanism to hide sensitive data written to a thin volume. Third, thin provisioning has an inborn ability to prevent overlap among thin volumes by using a free space bitmap to track allocated blocks. Fourth, it is feasible to create an encrypted block device on a thin volume and an arbitrary file system can be deployed.

Dummy Write Implementation. To implement the dummy write, we use *jiffies* as the random seed to determine the probability of the dummy write, which is a global variable in the Linux kernel. *Jiffies* holds the number of ticks that have occurred since the system booted. We store this variable in the thin pool structure. It is updated when data are written to the thin volume and the time interval is longer than one hour since the last update. The variable is random because its update is triggered by a write operation to the thin volume and the time of the write operation is random. The variable $rand$ is a random number between 0 and 100 and it is generated by

[4]There is no need to frequently perform garbage collection as long as the user does not frequently store large amount of public and hidden data. In addition, the user can choose to perform garbage collection when the mobile device is idle, e.g., during night time, to avoid disturbing regular use.

[5]https://github.com/changbing1/MobiCeal

the function *get_random_bytes()*. To conduct a dummy write, a free block is found using random allocation and then filled with random noise. In the bitmap, the corresponding bit of this block is set to "allocated", so that it will not be reallocated.

Random Allocation Implementation. To implement the random allocation, we first obtain the number of free blocks (denoted by x), and then we generate a random number i between 1 and x. The ith free block is the result. A transaction problem happens when an allocated block is allocated again before it is committed to the bitmap. To resolve the transaction problem, the block numbers allocated within a transaction are recorded. When a new block is allocated, MobiCeal judges whether this new block has been allocated in this transaction, so an allocated block will not be allocated again.

B. Changes to the Android Volume Daemon

In order to set up and use the public volume, the hidden volume, and the dummy volumes, we modify Android volume daemon (`Vold`) [6]. We implement the initialization process and the boot process. We also implement a function for switching to the hidden volume.

The Initialization Process. Users can active MobiCeal using `vdc`, a command-line utility, as follows: "`vdc cryptfs pde wipe` ⟨*pub_pwd*⟩ ⟨*num_vol*⟩ ⟨*hid_pwds*⟩". MobiCeal uses LVM to initialize the public, hidden and dummy volumes. Note that MobiCeal generates a random key as the decoy key that is used as the encryption key of the public volume. The decoy key is encrypted by the decoy password and the resulting cipher-text is stored in the encryption footer. The hidden key can be derived from decrypting the aforementioned cipher-text using the hidden password, without wasting additional space for storing the encrypted hidden key.

The Boot Process. MobiCeal attempts to mount the userdata volume when the device is booted up. If the system fails to find a valid Ext4 file system, it asks the user to enter a password. When the user enters a password, the system enables the thin volumes and then decrypt the decoy key (stored in the encryption footer) using the password. After that, the system creates an encrypted block device on the public volume using the decrypted key. If a valid Ext4 file system can be mounted, the password is correct and the system continues to boot. Otherwise, the system asks the user to enter another password.

Switching to the Hidden Volume. In order to verify the password and switch to the hidden mode, we implement a switching function in `Vold` [7]. This function accepts a string parameter (password) and switches to the hidden mode if the password is the hidden password. Otherwise the function returns "-1". The switching function first reads the salt and the encrypted decoy key from the encryption footer. Then a number k is derived using the password and the salt. A key is also derived by decrypting the decoy key using the password. After that, the function reads the encrypted password at the beginning of V_k. To verify the password, the system encrypts the password using the derived key. If the result is the same as the previous encrypted password, the password is correct and the system begins to switch to the hidden mode. Otherwise the password is wrong and the function simply returns "-1".

To switch to the hidden mode, the system first shuts down the Android framework to unmount "/data" partition. Then a new encrypted block device will be created on V_k using the hidden key. The encrypted block device will be mounted to "/data" and the Android framework will be restarted. After the Android framework is restarted, the hidden mode is activated, and users can store sensitive data in the hidden volume.

C. Changes to the Android Screen Lock

We modify the default Android screen lock app as an entrance of the hidden mode. We add a process to verify whether the password is the hidden password. That is, the system checks whether the password is the screen lock password as usual. If not, the system calls "`IMountService`" to pass the password to `Vold` which checks whether the password is the hidden password. If so, the system switches to the hidden mode. Otherwise the password is wrong, the system asks the user to enter another password.

VI. Analysis and Evaluation

A. Security Analysis

Lemma VI.1. *A hidden volume can be efficiently simulatable.*

Proof. We build a simulator $\mathcal{S}$, who is not allowed to reveal the hidden passwords $\{\mathcal{P}_l\}$ or any knowledge of the access patterns beyond its length, aims to simulate identical operations on public volumes if an operation (e.g., Write) occurs on hidden volumes. Note that in the MobiCeal system, if a data block is assigned to store data on public volumes, then a random noise will be written into a "dummy" volume with probability p (see below), we denote it as "dummy" Write. Specifically, the random noise on "dummy" volumes can be interpreted as either random strings or public key encryptions (e.g., IND-CPA secure) indistinguishable from random. Therefore, the freshly random strings Write on "dummy" volumes will be indistinguishable from an actual Write on hidden volumes. □

Remark. Note that adversary cannot estimate the amount of "dummy" Write, since p is a random and untraceable value. Therefore, the adversary cannot distinguish an actual Write on hidden volume from a "dummy" Write on dummy volume by statistical analysis.

Theorem VI.2. *The extended MobiCeal scheme achieves* multi-snapshot security*, if the hidden volumes are simulatable volumes.*

Proof. According to the definition of multi-snapshot security (see Definition III.1), the access patterns $(\mathcal{O}_0, \mathcal{O}_1)$ chosen by $\mathcal{A}$ will differ only on either a Read to volumes on disk or a Write to specific volumes $\mathcal{V}_j, j \geq l$. It is easy to see that a Read to $\mathcal{V}_i, i \neq j$ is indistinguishable from a Read to $\mathcal{V}_j$, while a "dummy" Write to a volume $\mathcal{V}_j, j \geq l$ is indistinguishable from an actual Write on hidden volume in

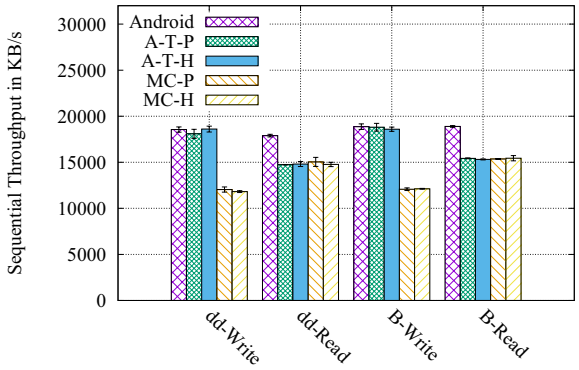

Fig. 4. Average throughput and standard deviation in KB/s (B: Bonnie++).

the sense of Lemma VI.1. Therefore, $\mathcal{A}$ cannot win the game with non-negligible advantage. □

According to Theorem VI.2, the extended MobiCeal scheme achieves multi-snapshot security, and the basic MobiCeal scheme is a special case of the extended MobiCeal scheme when the numbers of public and hidden volume are both one.

Metadata Security Issues.

The adversary can access the metadata for all virtual volumes, which keep track of blocks being assigned to each virtual volume. However, the modifications on the metadata caused by the hidden volume can be denied since the dummy writes can cause the same effects. Note that the adversary cannot decrypt the virtual volumes except the public volume, because the dummy volumes contain only random data and the encryption keys of the hidden volumes are protected by the hidden passwords. As a result, the adversary cannot identify whether any data blocks in a virtual volume are storing hidden data or dummy data.

B. Performance Evaluation

Throughput Performance. The main differences between MobiCeal and the default Android are that MobiCeal uses thin volumes and that the kernel is modified. We test how these two differences impact the performance on an LG nexus 4 phone. We measure the performance in the following settings: 1) Android: the default Android FDE, 2) A-T-P (Android-Thin-Public): the public volume of modified Android with thin volumes and the default kernel, 3) A-T-H (Android-Thin-Hidden): the hidden volume of modified Android with thin volumes and the default kernel, 4) MC-P (MobiCeal-Public): the public volume of MobiCeal, and 5) MC-H (MobiCeal-Hidden): the hidden volume of MobiCeal.

In our experiments, we use a popular Linux command tool, "*dd*" [5], to measure the sequential throughput. We measure the write speed using the following command, "*time dd if=/dev/zero of=test.dbf bs=400M count=1 conv=fdata-sync*". Note that "*conv=fdatasync*" is necessary because it ensures the data is written to the disk instead of a RAM buffer. To measure the read speed, we use "*time dd if=test.dbf of=/dev/null bs=400M*". Each time this command is executed, another command, "*echo 3 > /proc/sys/vm/drop_caches*", should be executed to empty the cache. Otherwise, the data in the cache may lead to wrong results.

TABLE I
OVERHEAD COMPARISON. THE VALUES OF DEFY ARE FROM THE FIGURE 6 IN [33]. TEST ENVIRONMENT: DEFY: UBUNTU 13.04, SINGLE PROCESSOR, 4GB RAM, SIMULATED FLASH DEVICE; HIVE: ARCH LINUX X86-64, I7-930, 9GB RAM, SAMSUNG 840 EVO SSD; MOBICEAL: ANDROID 4.2.2, SNAPDRAGON APQ 8064, 2GB RAM, NEXUS 4 INTERNAL STORAGE.

	Ext4 (MB/s)	Encrypted (MB/s)	Overhead
DEFY	800	50	93.75%
HIVE	216.04	0.97	99.55%
MobiCeal	19.5	15.2	22.05%

We conduct each test 10 times and use "*dd-Write*" and "*dd-Read*" in Figure 4 to show the average results and standard deviations. About the write speed, the use of thin volumes has little influence on the performance as MobiCeal reduces the performance by about 18%. The reason of the decrement is that we modify the kernel to implement the dummy write and the random allocation. About the read speed, the use of thin volumes reduces the performance by about 18% while the modified kernel has little influence on the performance. Thin provisioning adds a layer between file system and disk, so the additional operations reduce the read performance.

We also use Bonnie++ [22], a benchmark suite conducting tests on hard drives and file systems, to evaluate the performance. We repeat each experiment 10 times and show the results in Figure 4. Note that the files created in the Bonnie++ benchmarks must be set to twice the size of the system RAM (2GB in our case) so as to reliably measure the performance. The "B-Write" and "B-Read" items in Figure 4 show the results of the average throughput from Bonnie++. The results are similar to the results in the "dd" test. In addition, Bonnie++ also shows the CPU overhead which indicates the power consumption difference. It shows that the CPU overhead results are similar in all operation cases.

Overhead Comparison. Table I shows the overhead comparison between MobiCeal, DEFY [33] and HIVE [15], three solutions which can defend against multi-snapshot adversaries. We obtained the results of DEFY by interpreting the Figure 6 in [33] since the original data are unavailable in the paper. We derived the overheads according to the experimental results. Because the test environments are different, we cannot compare the results directly. DEFY was evaluated with IOZone [31] on an Ubuntu 13.04 with 4GB of memory and a single processor. The tested device was a 64MB flash device with 2KB pages, which was emulated with the *nandsim* MTD device simulator [42]. HIVE was evaluated with Bonnie++ [22] on an Arch Linux x86-64 with an Intel i7-930 CPU and 9GB RAM. The tested device was an off-the-shelf Samsung 840 EVO SSD. MobiCeal was evaluated with Bonnie++ on Google Nexus 4 with Qualcomm Snapdragon S4 Pro APQ8064 CPU and 2GB RAM. The tested device was the internal storage of Nexus 4.

The different test environments cause the different results. However, we can make comparison among the overheads. The overheads of DEFY and HIVE are both higher than 90%,

TABLE II
INITIALIZATION TIME, BOOTING TIME, AND SWITCHING TIME.

	Initialization	booting time (decoy pwd)	switching time (enter hid-mod)	switching time (exit hid-mod)
Android FDE	18min23s±1s	0.29±0.02s	N/A	N/A
MobiPluto	37min2s±2s	1.36±0.02s	68±4s	64±5s
MobiCeal	2min16s±3s	1.68±0.04s	9.27±0.28s	63±6s

but the overhead of MobiCeal is only about 22%. The high overhead of DEFY is caused by the additional computation requirements necessary to support the cryptographic operations. The encryption of MobiCeal relies on the *dm-crypt* kernel module which is more efficient. HIVE is based on Oblivious RAM and its high overhead is caused by the high computation cost of ORAM. MobiCeal relies on the modified thin provisioning to provide deniability, so the overhead is much lower. Note that HIVE can defend against a stronger adversary who can constantly monitor the device, but the significant performance overhead makes it impossible to be deployed on practical devices.

Timing Measurements. We also test the initialization time, the booting time, and the switching time, which affect the users' experience. The initialization time is the time used to finish the initialization process (Sec. V-B). To measure the initialization time, we use a timer to record the time interval between the moment when MobiCeal is activated by the *vdc* command, and the moment when the screen shows up the password entering interface. We analyze the booting time and the switching time by reading the logs of Android system. The booting time is the time interval between the moment when the decoy password is entered in the interface, and the moment when the public volume is decrypted. The switching time is the time interval between the moment when a password is entered in the screen lock and the moment when the system switches to the hidden mode.

We conduct each test 10 times and the means and standard deviations of the results are shown in Table II. The initialization of MobiCeal takes about 2 minutes, which is much shorter than MobiPluto [21]. The booting time is about 1.7 seconds and the switching time is less than 10 seconds. Previous solutions (Mobiflage [34], MobiHydra [43] and MobiPluto [21]) all need reboot to switch modes, which is time-consuming (more than 1 minute). Our solution does not need to reboot the phone and this is helpful in emergency.

VII. RELATED WORK

The concept of deniable encryption has been applied to network communications [17], disk storage and cloud storage [25]. In disk storage, the existing PDE designs can be classified into two categories: one against single snapshot adversaries and the other against multi-snapshot adversaries.

A. Designs against Single Snapshot Adversaries

The first file encryption scheme with PDE support is proposed by Anderson et al. [13]. Two solutions are presented: Hiding blocks within cover files and hiding blocks within random data. However, due to the high storage and I/O overheads, both solutions are not suitable for resource-limited mobile devices. StegFS [29] uses the second approach in [13] and works on Ext2 file system. However, their design relies on Ext2 file system and may not work on other file systems. In addition, the disk usage rate is very low due to the collision avoidance mechanism. TrueCrypt [2], FreeOTFE [3], EDS [10] and Fuyoal [8] are well-known desktop PDE tools that can defend against a single snapshot adversary.

Mobiflage [34], [35] presents the first PDE scheme for mobile devices, with two versions: One for FAT32 file system in external storage [34], and the other for Ext4 file system in internal storage [35]. MobiHydra [43] introduces multi-level deniability and supports sensitive data storing without rebooting. MobiPluto [21] introduces a file system friendly PDE design by combining hidden volume technique and thin provisioning. DEFTL[27] considers the nature of NAND flash and incorporates deniability to flash translation layer.

All the aforementioned PDE systems unfortunately cannot mitigate a multi-snapshot adversary, since they all rely on a static defense strategy, e.g., denying existence of hidden data using randomness being filled initially.

B. Designs against Multi-Snapshot Adversaries

Pang et al. [32] propose a steganographic file system design where blocks used by hidden files are marked as occupied in the bitmap. It uses "abandoned blocks" and "dummy blocks" to hide sensitive data. Although Pang et al.'s design has the concept of "dummy blocks", their design and our MobiCeal are different in the following aspects: 1) Their design is for desktop systems, and does not provide technical details on how the dummy data are written in order to defend against the multi-snapshot adversary. MobiCeal, on the contrary, designs a clear "dummy write" mechanism specifically for mobile devices, which is clearly shown to be secure against the multi-snapshot adversary. 2) Their design is based on legacy Linux kernel and requires special APIs for handling hidden files which may not be applicable to mobile devices. MobiCeal is incorporated into the block layer, and does not rely on any specific file system APIs.

Blass et al. present HIVE [15] to defend against a multi-snapshot adversary. HIVE relies on the expensive write-only ORAM, which suffers from a high system overhead and is thus not suitable for mobile devices. Chakraborti et al. [18], [19] improve HIVE, but their design still relies on ORAM, which cannot fit the resources limited mobile devices. Comparatively, MobiCeal eliminates the use of ORAM, and is lightweight enough to be used in mobile devices. Peters et al. introduce DEFY [33], a deniable encrypted file system for mobile devices based on YAFFS [40]. DEFY strongly relies on the special properties provided by YAFFS file system, which limits

its applications on the existing mobile devices since YAFFS is rarely deployed. On the contrary, MobiCeal is incorporated into the block layer, which ensures its broad applications.

VIII. Conclusion

In this paper, we propose MobiCeal, a practical PDE solution for mobile devices. MobiCeal is the first block-layer PDE scheme that is resistant to multi-snapshot adversaries on mobile devices. MobiCeal is practical since it is file system friendly and supports fast switching. We have implemented a prototype of MobiCeal on an LG Nexus 4 phone using Android 4.2.2 and tested it on a Huawei Nexus 6P phone using Android 7.1.2 as well. The performance overhead of MobiCeal is significantly lower than other PDE systems that can defend against the multi-snapshot adversary, which justifies that MobiCeal suits the application of mobile devices.

Acknowledgment

Chang, Li, and Ching's work is supported by the Singapore National Research Foundation under NCR Award Number NCR2016NCR-NCR002-022. Dr. Fengwei Zhang is supported by the National Science Foundation Grant No. CICI-1738929 and IIS-1724227.

References

[1] "BitLocker Overview," https://technet.microsoft.com/en-us/library/hh831713.aspx, 2013.
[2] "Free open source on-the-fly disk encryption software. version 7.1a," Project website: https://andryou.com/truecrypt_orig/, 2013.
[3] "A free "on-the-fly" transparent disk encryption program for PC & PDAs," Project website: http://sourceforge.net/projects/freeotfe.mirror/, 2017.
[4] "Android full disk encryption," https://source.android.com/security/encryption/, 2017.
[5] "Benchmarking," https://wiki.archlinux.org/index.php/Benchmarking, 2017.
[6] "Device Configuration," https://source.android.com/devices/storage/config.html, 2017.
[7] "Full-disk encryption," https://source.android.com/security/encryption/full-disk.html, 2017.
[8] "Fuyoal - deniable encryption software," Website: http://tsmolen.eu/fuyoal/, 2017.
[9] "Samsung eMMC memory," http://www.samsung.com/semiconductor/products/flash-storage/emmc, 2017.
[10] "Sovworks — EDS," Website: http://www.sovworks.com/, 2017.
[11] "The Device Mapper," https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Logical_Volume_Manager_Administration/device_mapper.html, 2017.
[12] "Use FileVault to encrypt the startup disk on your Mac," https://support.apple.com/en-us/HT204837, 2017.
[13] R. Anderson, R. Needham, and A. Shamir, "The steganographic file system," in *Information Hiding*. Springer, 1998, pp. 73–82.
[14] B. Kaliski, "PKCS # 5: Password-based cryptography specification, version 2.0," *RFC 2898*, 2000.
[15] E.-O. Blass, T. Mayberry, G. Noubir, and K. Onarlioglu, "Toward robust hidden volumes using write-only oblivious ram," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2014, pp. 203–214.
[16] M. Broz, "dm-crypt: Linux kernel device-mapper crypto target," *https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt*, 2017.
[17] R. Canetti, C. Dwork, M. Naor, and R. Ostrovsky, "Deniable encryption," in *Advances in Cryptology-CRYPTO'97*. Springer, 1997.
[18] A. Chakraborti, C. Chen, and R. Sion, "Poster: Datalair: A storage block device with plausible deniability," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1757–1759.
[19] ——, "Datalair: Efficient block storage with plausible deniability against multi-snapshot adversaries," *Proceedings on Privacy Enhancing Technologies*, vol. 3, pp. 175–193, 2017.
[20] B. Chang, Y. Cheng, B. Chen, F. Zhang, W.-T. Zhu, Y. Li, and Z. Wang, "User-friendly deniable storage for mobile devices," *Computers & Security*, vol. 72, pp. 163–174, 2018.
[21] B. Chang, Z. Wang, B. Chen, and 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.
[22] R. Coker, "Bonnie++ file system benchmark suite," http://www.coker.com.au/bonnie++/, 2009.
[23] A. Czeskis, D. J. S. Hilaire, K. Koscher, S. D. Gribble, T. Kohno, and B. Schneier, "Defeating encrypted and deniable file systems: Truecrypt v5.1a and the case of the tattling os and applications," in *Proceedings of the 3rd Conference on Hot Topics in Security*, 2008.
[24] Entropy512, "LVM Partition Remapping," http://forum.xda-developers.com/find-7/orig-development/ref-lvm-partition-remapping-t2865843, 2014.
[25] P. Gasti, G. Ateniese, and M. Blanton, "Deniable cloud storage: sharing files via public-key deniability," in *Proceedings of the 9th annual ACM workshop on Privacy in the electronic society*. ACM, 2010, pp. 31–42.
[26] A. Goodman, "Dn! exclusive: Authorities search and copy u.s. journalists notes, computer and cameras after returning from haiti," https://www.democracynow.org/2011/2/15/exclusive_authorities_search_and_copy_us, 2011.
[27] S. Jia, L. Xia, B. Chen, and P. Liu, "DEFTL: Implementing Plausibly Deniable Encryption in Flash Translation Layer," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.
[28] S. Levine, "Logical Volume Manager Administration," https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Logical_Volume_Manager_Administration/, 2017.
[29] A. D. McDonald and M. G. Kuhn, "StegFS: A steganographic file system for Linux," in *Proceedings of Information Hiding*. Springer, 2000.
[30] C. Mcfadden, E. Cauchi, W. M. Arkin, and K. Monahan, "American Citizens: U.S. Border Agents Can Search Your Cellphone," https://www.nbcnews.com/news/us-news/american-citizens-u-s-border-agents-can-search-your-cellphone-n732746, Mar 2017.
[31] W. D. Norcott, "IOzone Filesystem Benchmark," http://www.iozone.org, 2016.
[32] H. Pang, K.-L. Tan, and X. Zhou, "StegFS: A steganographic file system," in *Proceedings of the 19th International Conference on Data Engineering*, 2003.
[33] T. M. Peters, M. A. Gondree, and Z. N. Peterson, "DEFY: A deniable, encrypted file system for log-structured storage," in *the Network and Distributed System Security Symposium*, 2015.
[34] A. Skillen and M. Mannan, "On implementing deniable storage encryption for mobile devices," in *the Network and Distributed System Security Symposium*, 2013.
[35] ——, "Mobiflage: Deniable storage encryption for mobile devices," *IEEE Transactions on Dependable and Secure Computing*, vol. 11, no. 3, pp. 224–237, 2014.
[36] S. Smalley and R. Craig, "Security Enhanced (SE) Android: Bringing Flexible MAC to Android." in *NDSS*, 2013.
[37] O. Solon, "US border agents are doing 'digital strip searches'. Here's how to protect yourself," https://www.theguardian.com/us-news/2017/mar/31/us-border-phone-computer-searches-how-to-protect, Mar 2017.
[38] Steven676, "[HOWTO] "Partitioning" your Nexus S using LVM," http://forum.xda-developers.com/nexus-s/general/howto-partitioning-nexus-s-using-lvm-t1656794, 2012.
[39] L. Torvalds, "Thin provisioning documentation," https://www.kernel.org/doc/Documentation/device-mapper/thin-provisioning.txt, 2017.
[40] L. van Someren, "Yaffs — a flash file system for embeded use," http://www.yaffs.net, 2017.
[41] Y. Wang, W.-k. Yu, S. Wu, G. Malysa, G. E. Suh, and E. C. Kan, "Flash memory for ubiquitous hardware security functions: True random number generation and device fingerprints," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 33–47.
[42] D. Woodhouse, "Memory Technology Device Subsystem for Linux," http://www.linux-mtd.infradead.org/faq/nand.html, 2017.
[43] X. Yu, B. Chen, Z. Wang, B. Chang, W. T. Zhu, and 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.