DEFTL: Implementing Plausibly Deniable Encryption in Flash Translation Layer

Shijie Jia*† jiashijie@is.ac.cn
Luning Xia*† halk@is.ac.cn
Bo Chen‡ bchen@mtu.edu
Peng Liu§ pliu@ist.psu.edu

Abstract

Mobile devices today have been increasingly used to store and process sensitive information. To protect sensitive data, mobile operating systems usually incorporate a certain level of encryption to protect sensitive data. However, conventional encryption cannot defend against a coercive attacker who can capture the device owner, and force the owner to disclose keys used for decrypting sensitive information. To defend against such a coercive adversary, Plausibly Deniable Encryption (PDE) was introduced to allow the device owner to deny the very existence of sensitive data stored on his/her device. The existing PDE systems, built on flash storage devices, are problematic, since they either neglect the special nature of the underlying storage medium (which is usually NAND flash), or suffer from deniability compromises.

In this paper, we propose DEFTL, a Deniability Enabling Flash Translation Layer for devices which use flash-based block devices as storage media. DEFTL is the first PDE design which incorporates deniability to Flash Translation Layer (FTL), a pervasively deployed "translation layer" which stays between NAND flash and the file system in literally all the computing devices. A salient advantage of DEFTL lies in its capability of achieving deniability while being able to accommodate the special nature of NAND flash as well as eliminate deniability compromises from it. We implement DEFTL using an open-source NAND flash controller. The experimental results show that, compared to conventional encryption which does not provide deniability, our DEFTL design only incurs a small overhead.

1 Introduction

Mobile computing devices (e.g., smart phones and tablets) are increasingly ubiquitous nowadays. Due to their portability and mobility, more and more people today turn to such devices for daily communications, web browsing, online shopping, electronic banking, etc. This however, leaves large amounts of sensitive personal/corporate data in these devices. To protect sensitive information, all the major mobile operating systems have incorporated a certain level of encryption [25, 2]. A broadly used encryption technique is full disk encryption (FDE), which has been available on Android phones since version 3.0 [35]. FDE can defend against a passive attacker who tries to retrieve sensitive

*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, China
†Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, China
‡Department of Computer Science, Michigan Technological University, USA
§College of Information Sciences and Technology, The Pennsylvania State University, USA

information from the data storage. However, it cannot defend against an active attacker, who can capture the device owner, and force the owner to disclose the key used for decrypting the sensitive information. This applies to a lot of real world scenarios. For example, a professional journalist or human rights worker collects criminal evidence using his/her mobile device in a region of oppression or conflict, and stores the information encrypted. However, when he/she tries to cross the border, the boarder inspector may notice the existence of the encrypted data and may coerce him/her to disclose the decryption key. We need a technique which can protect the sensitive data even when the data owner faces such a coercive attack. This is a necessary technique for protecting the sensitive data as well as the people who possess them. In 2012, a videographer smuggled evidence of human rights violations out of Syria. Due to lack of effective data protection mechanisms, he instead hid a MicroSD card in a wound on his arm [27, 31].

Plausibly Deniable Encryption (PDE) has been proposed to defend against adversaries who can coerce users into revealing the encrypted sensitive content (e.g., by forcing the victims to disclose the keys for decryption). The high-level idea of PDE is: the original sensitive data are encrypted into a cipher-text in such a way that, when using a decoy key, a different reasonable and innocuous plain-text will be generated; only when using the true key, the original sensitive data will be disclosed. Upon being coerced, the victim can simply disclose the decoy key to avoid being tortured. Leveraging the concept of PDE, a variety of deniable storage systems have been proposed for PC platforms, e.g., TrueCrypt [38], Rubberhose [16], HIVE [3], Gracewipe [45], Steganographic File Systems [1, 23, 30], etc.

However, achieving deniability in modern mobile platforms is much more challenging than in PC platforms because: First, compared to a PC platform, a mobile platform is usually equipped with limited computational resources and sensitive to energy consumption. In other words, the PDE designs for mobile platforms have much higher requirements in efficiency and energy effectiveness. Therefore, the existing PDE systems [38, 3, 1, 23, 30] built for PC platforms are not immediately applicable to the mobile platforms due to their large overhead. Second, modern mobile devices usually use NAND flash as storage media. Flash storage products like SD cards, MicroSD cards, and eMMC cards have dominated the storage of mobile devices. Compared to mechanical disks which are used broadly in the PC platforms, flash memory has significantly different nature: 1) Flash memory is update unfriendly. A flash cell cannot be over-written before it has been erased. However, the erase can only be performed on the basis of a large region (i.e., a 128-KB block); 2) Flash memory is vulnerable to wear. A flash cell can only be programmed/erased for a limited number of times before the wear begins to deteriorate its integrity.

Although deniable storage systems have also been proposed for mobile platforms (e.g., Mobiflage [33, 34], Mobihydra [44], Mobipluto [6], DEFY [31], etc.), most of the existing PDE systems for mobile devices [33, 34, 44, 6], unfortunately, neglect the above unique nature of the underlying flash memory and view the flash storage as block devices. This can simplify the PDE designs, but the resulting PDE systems may suffer from deniability compromises in the underlying flash storage, which usually incorporates special techniques to accommodate flash memory's unique nature (concrete attacks are shown in Sec. 3.1). When deniability is (partially) compromised, the concrete attacks we will present shortly in Sec. 3.1 show that the attacker can figure out the existence of hidden sensitive data.

DEFY [31] is till now the sole PDE system which works with flash memory to achieve deniability while being able to accommodate the special nature of flash memory. However, DEFY suffers from several deficiencies. First, DEFY heavily relies on the special properties provided by a specific flash file system (i.e., YAFFS2), which significantly limits its broad applications in mobile devices. Due to DEFY's strong coupling with YAFFS2, it cannot be applied to other flash file systems including F2FS and UBIFS. In addition, an overwhelming majority of mobile devices nowadays do not allow applications to directly access the raw NAND flash, which significantly inhibits the deployment of flash file systems. Instead, many mobile devices (e.g., Android phones from Samsung, LG, HTC and Motorola) use flash memory in the form of a flash-based block device, by introducing a new flash translation layer (FTL) to transparently handle the special nature of flash memory and exposing a block-based access interface. DEFY unfortunately is incompatible with this popular flash storage architecture. Second, DEFY cannot completely eliminate deniability compromises in flash memory. This is because, DEFY relies on disabling active garbage collection in a lower security level to avoid overwriting the hidden sensitive data in a higher security level. This however, opens a door for the attacker to compromise deniability (Sec. 3.2).

In this paper, we present DEFTL, a **D**eniability **E**nabling **F**lash **T**ranslation **L**ayer for mobile devices which use flash-based block devices as storage. DEFTL is the first design that incorporates PDE in FTL, a pervasively deployed "translation layer" which stays between the physical flash layer and the file system layer in literally all the mobile computing devices. Our design relies on several key insights:

First, having observed that most of the existing PDE systems incorporate deniability in the upper layers (e.g., file system layer [1, 23, 30] and block layer [38, 3, 33, 34, 44, 6]) and may suffer from deniability compromises in the lower layers (e.g., storage medium layer), we move the PDE design downwards. This is especially necessary for mobile devices equipped with flash storage, because the adversary can have access to the raw flash, and obtain a view different from the PDE systems working on the upper layers. Such a different view may allow the adversary to observe those unexpected "traces" of the sensitive data (due to handling the special nature of flash memory), whose existence need to be denied. By moving PDE to the lower layers, we make it possible to eliminate those deniability compromises in the raw flash, since we can now have the same view as the adversary. In addition, our design is compatible with flash-based block devices, the most popular form of flash storage which exposes a block access interface. This can allow any block-based file systems (e.g., EXT4, FAT32) to be deployed, achieving file system friendliness.

Second, we incorporate two modes, a public mode and a PDE mode, into FTL, and carefully isolate the two modes to achieve deniability. When operating in the public mode, the user can store and process public non-sensitive data which can be known by the adversary; when operating in the PDE mode, the user can store and process sensitive data which should be hidden and their existence should be able to be denied. The deniability is achieved by using the data (and their behavior) in the public mode to deny the data (and their behavior) in the PDE mode. To avoid deniability compromises, the public mode should not have any knowledge on the existence of the PDE mode. This however, will lead to an over-write issue, in which the data written in the public mode may over-write the data written in the PDE mode. To address the over-write issue, we carefully modify the block allocation and garbage collection strategies in FTL such that the two modes can be "stealthily" isolated without being known by the adversary.

Contributions. Our contributions are summarized in the following:

- We introduce the first concrete attacks on the existing PDE systems for mobile devices. Our attacks can successfully compromise deniability provided by all the prior mobile PDE systems.
- Eliminating deniability compromises from flash-based block devices is an open problem which has not been addressed in prior work. DEFTL is the first design which directly incorporates deniability in the flash translation layer, and is able to eliminate the deniability compromises

present in flash storage. In addition, the inherent over-write issue of PDE systems can be mitigated by modifying the block allocation and the garbage collection such that the public non-sensitive data and the hidden sensitive data can be "stealthily" isolated.

- We provide security analysis and a proof-of-concept implementation of DEFTL using an open source NAND flash controller framework. Compared to conventional encryption (e.g., FDE) in flash-based block devices, our DEFTL can achieve deniability with a small additional overhead.

2 Background

2.1 Flash Memory

Flash memory is a non-volatile storage medium which can be electrically erased and reprogrammed. Compared to traditional mechanical drives, flash memory can achieve much higher I/O throughput with much lower power consumption, and thus gains popularity in the modern computing devices. The flash memory family contains NAND-type and NOR-type flash. In this work, we focus on NAND flash, which has been used extensively in popular flash storage media including eMMC cards, SD cards, MicroSD cards, SSD drives and USB sticks.

NAND flash stores information using an array of memory cells. Different from mechanical disks, NAND flash has several unique characteristics. First, it has an *erase-before-write* design. Specifically, overwriting the same flash cell is not feasible before an erasure is performed over it. *Note that a block erasure will re-set all the bits in the entire block to "1" bits.* Second, the unit for a read/program operation is a page (which can be 512 bytes, 2KB, or 4KB), but the unit for an erase operation is a block, which consists of multiple (typically 32, 64, or 128) pages. Therefore, overwriting a page requires first erasing the entire encompassing block. If a few other pages of this block are filled with valid data, erasing this block requires copying the valid data elsewhere and writing them back after the erase operation is performed, leading to significant *write amplification.* To mitigate write amplification, flash usually adopts an out-of-place update strategy [17], in which when a logical disk region is overwritten, it is simply re-mapped to a new empty page without erasing the original invalid page. Third, each flash block has a finite number (e.g., 10K) of program-erase (P/E) cycles. In other words, a flash block will be worn out if the number of programs/erases performed over it exceeds a certain threshold. Therefore, *wear leveling* is required to distribute programmings/erasures evenly across the entire flash to prolong its lifetime.

2.2 Flash Translation Layer (FTL)

There are two common options of using NAND Flash. The first option is to build a file system specifically for raw NAND Flash (i.e., a flash file system). Popular flash file systems include YAFFS [42], UBIFS [26] and F2FS [21]. However, a flash file system usually requires directly accessing the raw flash memory, which is unfortunately rarely supported by modern mobile devices. For example, the most recent Android phones like Nexus 6P do not allow a direct access to the raw flash; only the old versions of Android phones (e.g., Nexus One and Nexus S) can allow such an access [7]. Therefore, flash file systems are becoming less and less popular.

The second option is to emulate the flash medium as a block device by exposing a block-based access interface, such that the flash medium can be compatible with traditional block-based file

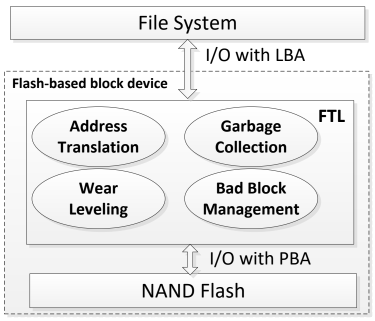

Figure 1: The architecture of a flash-based block device

systems (e.g., EXT4 and FAT32). All the popular flash storage products (e.g., eMMC cards, SD cards, MicroSD card, SSD drives and USB sticks) are manufactured following this manner. In this option, a piece of special flash firmware, Flash Translation Layer (FTL), is introduced between the file system and the raw NAND flash to transparently handle the unique nature of NAND flash. In this work, we mainly consider flash storage devices which are exposed as block devices using FTL. We call this type of flash devices **"flash-based block devices"**. Figure 1 shows the architecture of a flash-based block device. In general, FTL implements four key functions: address translation, garbage collection, wear leveling and bad block management.

Address translation. To reduce write amplification, flash storage usually implements an out-of-place update mechanism, in which to overwrite data stored on a page, the new data will be programmed to a fresh empty page, while the page storing the stale data will be simply marked as invalid. Therefore, the location of valid data may change over time, which requires maintaining mappings between the addresses (i.e., Logical Block Address, or **LBA**) from upper layer and the actual flash addresses (Physical Block Address, or **PBA**). FTL manages such mappings and provides a block-based access interface.

Garbage collection. The out-of-place update mechanism used in flash may result in a large number of invalid pages/blocks over time, which need to be reclaimed. This is usually handled by garbage collection. To reclaim blocks, garbage collection usually periodically performs the following operations [37]: 1) select those blocks satisfying certain reclaim criteria as *victim blocks*; 2) copy the valid data stored in the victim blocks to free blocks, and update the corresponding mappings; 3) erase the victim blocks.

Wear leveling. Each flash block can only be programmed/erased for a limited number of times (i.e., a limited P/E cycles). To prolong the lifetime of flash memory, FTL usually performs wear leveling by distributing programmings/erasures evenly among the entire flash. The existing wear leveling strategies mainly include dynamic and static wear leveling. Dynamic wear leveling always writes data to those blocks which have less P/E cycles. Comparatively, static wear leveling will periodically swap hot and cold blocks [43].

Bad block management. As flash cells degrade over time, flash memory will eventually develop blocks that are not able to reliably store data. Those blocks should be marked as "bad" and avoid being used to store data. During manufacturing, the entire flash blocks will be scanned and an initial table of bad blocks will be generated. During run time, the FTL will monitor the newly generated bad blocks and add them to the bad block table [36].

2.3 Hidden Volume Technique

The hidden volume technique can be used to defend against a coercive adversary, and is implemented in popular encryption software like VeraCrypt [10] and TrueCrypt [38]. It works as follows: Two volumes, a public volume and a hidden volume, are created on the disk. The public volume is encrypted using a *decoy key* and is placed across the entire disk. The hidden volume is encrypted using a secret *true key* and is placed towards the end of the disk from a secret offset. The sensitive data being protected will be stored in the hidden volume. Note that the entire disk will be filled with random data initially. Upon being coerced by the adversary, to protect the true key, the victim can simply disclose the decoy key. Using the decoy key, the adversary can decrypt the public volume, and is not able to detect the existence of the hidden volume, since he/she cannot differentiate the encrypted hidden volume from the randomness being filled initially.

A significant issue for the hidden volume technique is, the data written to the public volume may over-write the data stored in the hidden volume, since the existence of the hidden volume is unknown to the system which manages the public volume. This work achieves deniability in mobile devices equipped with flash-based block devices by: 1) adapting the hidden volume technique to flash translation layer; 2) mitigating the over-write issue by carefully tuning block allocation and garbage collection in FTL. In the remainder of this paper, we use *a hidden volume-based PDE system to represent a PDE system which relies on the hidden volume technique to achieve deniability.*

3 Attack Scenarios

In this section, we provide concrete attack scenarios, in which the adversary is able to compromise deniability provided by prior PDE systems for mobile devices [33, 34, 44, 6, 31]. For each attack scenario, we assume the adversary is able to obtain the physical image of the raw NAND flash [4].

3.1 Attacking the Hidden Volume-based PDE Systems for Mobile Devices

When attacking a hidden volume-based PDE system for mobile devices equipped with flash-based block devices, the adversary can identify three types of flash blocks (Figure 2) by having access to the raw NAND flash:

- **Type-I**: blocks filled with random data.
- **Type-II**: blocks with a few pages filled with random data followed by pages filled with all “1” bits.
- **Type-III**: blocks filled with all “1” bits.

Note that a block erasure in flash will re-set the entire block to all “1” bits (Sec. 2.1). Thus, the type-III blocks are those which have been erased to prepare for new writing requests, which do not contain any valid data. In the following, we show that by analyzing the type-I and type-II blocks, the adversary may be able to compromise the deniability provided by the existing mobile PDE systems [33, 34, 44, 6].

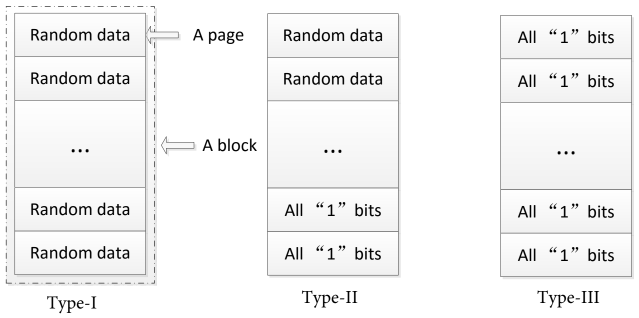

Figure 2: Three types of blocks in the flash physical image (Type-I: all random data; Type-II: a few random data followed by all "1" bits; Type-III: all "1" bits)

3.1.1 Attack 1: Deniability Compromises from Analyzing Type-I and Type-II Blocks

In hidden volume technique, any data that cannot be decrypted by the decoy key (i.e., the key for decrypting public volume) will be interpreted as random data to deny the existence of the hidden volume. However, in a hidden volume-based PDE system for mobile devices, by observing the physical image of raw flash, the adversary may be able to identify a few type-I and type-II blocks which appear to store random data but actually store hidden sensitive data, leading to compromise of deniability.

Deniability compromises from the type-I blocks. The adversary tries to decrypt the data in all the type-I blocks using the decoy key. Without the existence of hidden volume, the adversary should only obtain two types of decryption outcomes: 1) all the data in a type-I block can be successfully decrypted, i.e., this is a type-I block filled with non-sensitive public data; 2) all the data in a type-I block cannot be decrypted, i.e., this is a type-I block storing random data.

However, the existing hidden volume-based PDE systems for mobile devices access the flash-based block device via a block access interface, and are thus not able to control the block allocation in FTL. As a result, the public volume and the hidden volume may share a common flash block (e.g., after having erased a block and used a portion of the pages in this block, the public volume is unmounted and the hidden volume is mounted, and the empty pages in this block will be used by the sensitive data stored to the hidden volume). Therefore, if the adversary detects a few "special" type-I blocks, in which the data stored at a few pages can be decrypted into meaningful public data, and the data stored at the remaining pages cannot be decrypted (i.e., the data stored at them are purely randomness), he/she may suspect the existence of hidden volume, leading to compromise of deniability.

Deniability compromises from the type-II blocks. The adversary tries to decrypt all the data stored in the type-II blocks using the decoy key. Without the existence of hidden volume, the adversary should be able to obtain only one type of decryption outcome: all the random data stored in a type-II block can be successfully decrypted into non-sensitive public data. This is because, without the existence of hidden volume, a type-II block can only be generated under the following circumstance: when data are written to the public volume, due to the erase-before-write design of flash storage, a flash block needs to be first erased to all "1" bits. If the amount of data written to this block is smaller than the block size, they will only occupy a few pages in this block. Therefore, if the adversary detects a few "special" type-II blocks, in which the random data cannot be decrypted into meaningful public data using the decoy key, he/she may suspect the existence of hidden volume, leading to compromise of deniability.

3.1.2 Attack 2: Deniability Compromises from Duplicate Random Data

All the hidden volume-based PDE systems for mobile devices [33, 34, 44, 6] deny the hidden sensitive data by random data, which are cryptographically secure and are initially filled in the entire flash medium. In general, the random data filled in flash pages will not be duplicate with each other due to their randomness nature. However, with the presence and activities of the hidden volume, the adversary may be able to find out duplicate random data among flash pages in the prior PDE systems, leading to compromise of deniability.

Deniability compromises introduced by garbage collection. Upon garbage collection, the data stored at the valid pages of a victim block will be copied to a new free block, and this victim block will be erased. However, for performance consideration, the victim blocks are usually not erased immediately [8, 18]. Therefore, duplicate data may appear in both the victim blocks which have not yet been erased and the new blocks used to store the valid data. When such a garbage collection is used to reclaim the data stored in the hidden volume, the adversary may be able to detect duplicate random data in different flash blocks, though he/she does not have any knowledge of the existence of the hidden volume. This will lead to the compromise of deniability.

Deniability compromises introduced by bad block management. A flash block will be considered as "bad" due to the permanent failure of one of its pages, even though all the other pages in it still function correctly. When a block is detected as "bad", all the valid data being stored will be copied to a new free block. However, most of the bad block management schemes [14, 24] simply mark the block as bad, without removing the valid data stored in it. When such a bad block management scheme is used in the hidden volume, the adversary may be able to detect duplicate random data between a normal block and a bad block, without knowing the existence of the hidden volume. This will also lead to compromise of deniability.

3.1.3 Making Attacks Easier: The Decreasing Amount of Random Data

The existing hidden volume-based PDE systems for mobile devices try to hide sensitive data in the randomness being filled to the entire storage medium initially (Sec. 2.3). This implies that the system should maintain an enough amount of randomness over time. However, we show in the following that the expected amount of randomness may not be maintained.

Unable to completely fill the entire flash medium with randomness during initialization. During initialization, the existing hidden volume-based PDE systems [33, 34, 44, 6] try to fill the entire storage medium with randomness. To achieve this, write requests will be passed to the FTL via a block-based access interface, informing the FTL to fill the LBAs with random data. However, due to the internal complexity of FTL, over-writing all the LBAs is not able to ensure that all the physical flash blocks will be filled with randomness [15, 41].

Block erasure decreases randomness. In a hidden volume-based PDE system, when new data are written to the public volume, they will over-write the corresponding disk locations, and the remaining disk locations are still filled with randomness. Flash storage, however, does not allow an over-write operation before a block erasure has been performed. In other words, before any data can be written, the corresponding flash block (originally filled with randomness) will be erased with all "1" bits. If this block is not completely filled by the new data, it will create a few pages in this block, which are filled with all "1" bits, rather than randomness. Additionally, when data are updated/deleted, the corresponding flash blocks will be erased with all "1" bits, rather than randomness. The existing PDE systems for mobile devices [33, 34, 44, 6] unfortunately only operate in the block layer, and thus are not able to re-fill those pages which do not hold valid data with randomness.

3.1.4 Summary of The Attacks on The Hidden Volume-based PDE systems

A common issue for all the existing hidden volume-based PDE systems for mobile devices is that they are incorporated into the block device layer, which can only allow them to access the flash memory via a block-based access interface. In other words, they are not able to handle the deniability compromises in flash memory, by either controlling the block allocation strategy (Attack 1 described in Sec. 3.1.1 is thus possible), or manipulating garbage collection and bad block management (Attack 2 described in Sec. 3.1.2 is thus possible). This observation motivates us to push the PDE design to flash translation layer (FTL) to eliminate those deniability compromises.

3.2 Attacking DEFY

DEFY [31] is till now the sole PDE system which directly works with flash memory to achieve deniability. It heavily relies on the properties offered by flash file system YAFFS2. To achieve PDE, DEFY introduces multiple security levels, and the lower security levels will not have any knowledge on the existence of the higher security levels. Upon facing coercive attack, the victim can use the data/activities at the lower security levels to deny the data/activities at the higher security levels. However, the data from the lower security levels may overwrite the data from the higher security levels. To mitigate this issue, DEFY disables garbage collection at the lower security levels.

As the adversary is able to enter a lower security level, he/she can compromise the deniability offered by DEFY as follows: The adversary copies the device's data elsewhere, deletes them from the device, and writes the data back to the device. After repeating the aforementioned operations a few times, he/she will find that no more data are allowed to be written to the device, even though there is still a large amount of empty space. This is because, by disabling garbage collection in this (lower) security level, no invalid space can be reclaimed. This provides the adversary a clear indication of the existence of deniability.

4 Model and Assumptions

4.1 System Model

We mainly consider mobile computing devices which use flash-based block devices (e.g., eMMC cards, SD cards, and MicroSD cards) as storage media. Such devices are pervasive nowadays. For example, a majority of smart phones (e.g., Android phones, iPhones, Windows phones) and tablets use eMMC cards as internal storage, and MicroSD cards as external storage. We also consider other computing devices (e.g., laptops) which are equipped with SSDs.

4.2 Adversarial Model

We consider a computationally bounded adversary who can capture a victim device equipped with flash-based block device. The adversary can obtain root privilege of the device, and have a full control over the device's internal and external storage. In addition, the adversary can coerce (e.g., by torture) the device owner to disclose the decryption key. Note that as the adversary is in possession of the victim device, he/she is able to obtain a full snapshot of the storage medium. This includes the physical image of the raw NAND flash, obtainable by forensic data recovery tools [4]. We call the aforementioned adversary a "**single-snapshot**" adversary. *We do not consider an adversary who can periodically obtain a snapshot of the storage medium from the victim device by monitoring the device unbeknownst to the victim (i.e., a multiple-snapshot adversary).*

4.3 Assumptions

Our design relies on multiple assumptions, which are also required in the prior PDE systems for mobile devices [33, 34, 44, 6]. The assumptions are summarized in the following:

- The adversary will know the design of DEFTL. However, he/she does not have any knowledge on the keys and passwords of the PDE mode.
- The adversary will stop coercing the device's owner once he/she is convinced that the decryption keys have been revealed.
- The adversary cannot capture a device working in the PDE mode or after a crash of the PDE mode. Otherwise, he/she can trivially retrieve the sensitive data or detect the existence of PDE.
- The operating system, bootloader, baseband OS, and flash firmware are all malware-free. In addition, the adversary will not be able to perform reverse engineering over the bootloader and the flash firmware, since PDE always requires incorporating addition code into those components and performing reverse engineering will unavoidably lead to compromise of deniability.

5 DEFTL Design

In this section, we present DEFTL, the first design that enables plausible deniability in Flash Translation Layer.

5.1 DEFTL Overview

In general, to achieve deniability, we should answer two key questions as follows:

Question 1: How to prevent the sensitive data from being leaked to a coercive adversary?

Intuitively, we can hide the sensitive data within the public non-sensitive data, and use the public data to deny the existence of hidden data. Specifically, any state/behavior of the hidden data can be interpreted as the state/behavior of the public data. We adapt the hidden volume technique (Sec. 2.3) to FTL to achieve deniability. Specifically, we introduce two volumes, a public volume and a hidden volume. The public volume will occupy the entire flash medium, and the hidden volume is stored stealthily among the public volume. All the public non-sensitive data will be stored in the public volume, which will be encrypted by a decoy key. All the sensitive data will be stored in the hidden volume, which will be encrypted by a true key. Correspondingly, we have two modes, a *public mode* and a *PDE mode*. The public mode refers to the system which manages the public volume, while the PDE mode refers to the system which manages the hidden volume. The entire flash medium will be filled with cryptographically secure randomness during *initialization*.

When implementing the hidden volume technique in FTL, we incorporate a few additional strategies to avoid deniability compromises being resulted from handling the special nature of flash memory: First, to eliminate deniability compromises from type-I blocks (Sec. 3.1.1), we enforce a special "no overlap block" policy, such that the public and the hidden volume will not share flash blocks; Second, to eliminate deniability compromises from type-II blocks (Sec. 3.1.1), we manipulate the type-II blocks belonging to the hidden volume. Specifically, when unmounting the hidden volume, we fill the empty pages of type-II blocks with randomness, such that each type-II block for the hidden volume will be indistinguishable from a block truly filled with randomness. Third, to eliminate deniability compromises from duplicate random data (Sec. 3.1.2), we modify the garbage collection and the bad block management in the PDE mode, such that the victim blocks and the bad blocks will be erased immediately after the valid data stored on them have been copied to other blocks. Last, to maintain an enough amount of randomness (Sec. 3.1.3): during initialization, we fill the entire flash medium with randomness by directly writing random data to all the physical flash blocks; when unmounting the hidden volume, we fill those empty pages (i.e., a page with all "1"s) with randomness.

Question 2: How to prevent the hidden sensitive data from being over-written by the pubic non-sensitive data?

To avoid deniability compromises, the public mode should not have any knowledge on the existence of the hidden volume. Otherwise, the adversary may take advantage of this to compromise deniability. However, without such knowledge, the data being written to the public volume may over-write the data stored in the hidden volume. A strategy which can mitigate this over-write issue while being able to be compatible with the flash translation layer, is desired. Our strategy is, we "stealthily" isolate the public volume and the hidden volume by manipulating block allocation and garbage collection in FTL as follows: First, we create a free block pool which stores all the blocks available to be allocated to the new data. When allocating blocks in the public mode, we always select blocks from the head of the pool; when allocating blocks in the PDE mode, we always select blocks from the tail of the pool; Second, we manipulate the garbage collection (in both the public mode and the PDE mode), such that blocks can be reclaimed frequently enough to fill the head of the free block pool and the public mode is less likely to allocate blocks belonging to the hidden volume.

Following the insights described above, we design DEFTL to enable deniability in flash translation layer. In the remainder of this section, we describe key components of DEFTL. We first introduce block types and metadata used by DEFTL to manage the public and the hidden volume. We then elaborate key operations in initialization, public mode and PDE mode. Finally, we introduce user steps for using DEFTL.

5.2 Block Types and Metadata

Block types. DEFTL differentiates four types of flash blocks: 1) The blocks which do not store any valid public or hidden data. These truly free blocks can be reclaimed by both the public and the hidden volume. We call these blocks in state **A**. 2) The blocks which do not store any valid public data, but store valid hidden data. These blocks can be reclaimed in the public volume, since the public mode should not have any knowledge on the existence of the hidden data to avoid deniability compromises. We call these blocks in state **B**. 3) The blocks which contain both the valid public volume pages (i.e., the pages which store valid public data) and the invalid public volume pages (i.e., the pages which store obsolete public data). We call these blocks in state **C**. 4) The blocks which only contain valid public volume pages. We call these blocks in state **D**.

Metadata. To manage both the public and the hidden volume, DEFTL maintains a few metadata:

- **Mapping table**. To provide a block-based access interface to the external entity (e.g., the file system), every logical block address (LBA) should be mapped to a physical flash address. In DEFTL, we maintain a mapping table for each volume. The external entity like the file system can access either the public volume or the hidden volume, and each volume is mapped to a set of flash blocks transparently by FTL using the corresponding mapping table. Note that the mapping table for the hidden volume is stored encrypted (using the true key) in a few flash blocks (see Sec. 5.5), such that those blocks are indistinguishable from other blocks truly filled with randomness. This is to prevent the adversary from identifying the existence of the hidden volume mapping table without the true key.

- **Dirty block table**. Both block allocation and the selection of the victim blocks during garbage collection are based on the number of valid/invalid pages in each block [20]. Therefore, for each volume, we maintain a dirty block table, which keeps track of the number of valid pages for each block belonging to this volume. Since the hidden volume should not use those blocks which have been occupied by the data written to the public volume, we maintain a separate dirty block table for the hidden volume, which includes all the free blocks (blocks in both state A and B) in the public volume. Note that the dirty block table for the hidden volume will be encrypted (using the true key) and stored together with the hidden volume mapping table.

- **Other metadata**. We also maintain other essential metadata shared by both volumes, e.g., a bad block table (which keeps track of the bad blocks), a root table (which keeps track of the location of the metadata for public volume), an erasure count table (which keeps track of the erasure count of each flash block for wear leveling purpose).

5.3 Initialization

During initialization, DEFTL first fills the entire flash medium with random data, and then initializes the public and the hidden volume.

Filling the entire flash with randomness. A fundamental requirement for a hidden volume-based PDE (Sec. 2.3) is that the entire storage medium must be filled with cryptographically secure random data, which will be used to deny the existence of hidden sensitive data. However, all the existing hidden volume-based PDE schemes for mobile devices [33, 34, 44, 6] can only fill the flash medium with randomness using a block-based access interface. This however, is not able to ensure that all the physical flash blocks will be filled with randomness (Sec. 3.1.3). DEFTL works in the FTL layer and can have direct access to the raw flash. Therefore, we can simply fill each physical flash block with randomness, which can be generated by taking advantage of random telegraph noise [40].

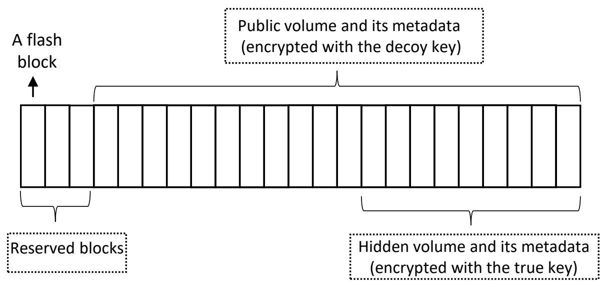
A flash block
Public volume and its metadata (encrypted with the decoy key)
Reserved blocks
Hidden volume and its metadata (encrypted with the true key)

Figure 3: A storage layout of DEFTL

Initializing the public and the hidden volume. We create a public volume across the entire flash, to store all the public non-sensitive data. The corresponding public volume metadata (i.e., the mapping table and the dirty block table) will be encrypted using the decoy key and stored in a few flash blocks, and this location information will be kept in the root table. The root table, the bad block table, and the erasure count table will be stored in the flash blocks reserved at the beginning of the flash medium. The mapping table is empty and can be filled with "0" initially, and the dirty block table should include all the flash blocks except the reserved blocks. Note that the dirty block table will organize the blocks according to the count of valid pages in each block in an increasing order. The entire public volume will be encrypted using the decoy key. A portion of flash blocks are used to create a hidden volume, which will be encrypted using the true key. The corresponding hidden volume metadata (i.e., the mapping table and the dirty block table, which are empty and can be filled with "0" initially) will be encrypted by the true key and stored in a few flash blocks (see Sec. 5.5) belonging to the hidden volume. Without obtaining the true key, the adversary is not able to identify the existence of the hidden volume and its metadata since they cannot be differentiated from the randomness filled by DEFTL. A storage layout of DEFTL is shown in Figure 3.

5.4 Public Mode

The public mode is used to manage the public volume, which stores the regular non-sensitive data that can be disclosed to the adversary (i.e., public data). In the following, we elaborate several key operations in this mode, including public volume mounting, block allocation, and garbage collection. These operations incorporate our design considerations for deniability purpose. Note that we neglect other regular operations (e.g., bad block management, public block unmounting), since no special designs need to be incorporated to them to achieve deniability.

Mounting the public volume. The public volume is encrypted by a decoy key, which can be derived from a decoy password using a key derivation function (e.g., PBKDF2 [19]). During booting, the user enters the decoy password to activate the public mode. Using the decoy password, DEFTL can derive the decoy key and use the decoy key to decrypt the public volume metadata, which can be localized by reading the root table. If the password is the correct decoy password, the decryption will be successful[1]; otherwise, the system aborts from mounting the public volume. Using the mapping table of the public volume, the blocks for the public volume can be localized, and the

[1]This can be easily verified by embedding a few known symbols at the beginning of the metadata.

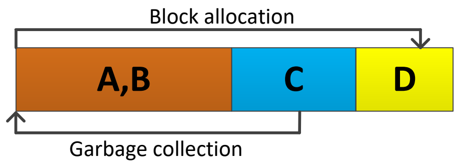

Figure 4: The dirty block table of the public volume

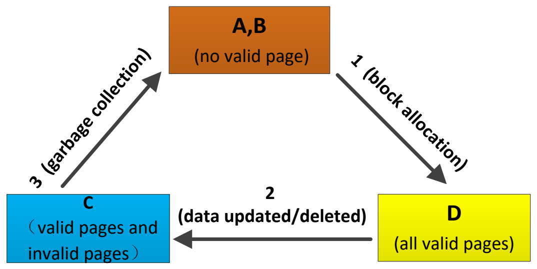

Figure 5: Block state transition in the public mode

entire public volume will be decrypted and mounted. Note that during this mounting process, all the flash blocks which store data/metadata of the hidden volume cannot be localized and decrypted. This can prevent the adversary from identifying the existence of the hidden volume. In addition, from the view of upper layer, the public volume appears as a regular block device, which can be used to deploy any block-based file systems (e.g., EXT4, FAT32). This file system friendliness [6] is an "unexpected" benefit of DEFTL.

Block allocation. In the public mode, DEFTL allocates flash blocks for the new writes according to the public volume dirty block table. Note that the dirty block table stores the count of valid pages for each flash block, and organizes the blocks according to their counts in an increasing order. When new write requests have been received and new blocks need to be allocated, DEFTL selects the free blocks from the head of the dirty block table (Figure 4). By smartly manipulating the dirty block table of the public volume, DEFTL ensures that it is more likely the blocks in state A will be allocated (see Sec. 5.5), rather than the blocks in state B, which store the hidden sensitive data. A free block being allocated needs to be erased if it has not yet been erased before.

As it is shown in Figure 5, the free blocks being allocated will be used to store valid public data, turning to state D (arrow 1 in Figure 5). The blocks in state D may then be updated/deleted, turning to state C (arrow 2 in Figure 5).

Garbage collection. In the public mode, both the blocks in state A and the blocks in state B are viewed as free blocks and can be allocated for storing new data. In other words, once the blocks in state A have been completely used, the blocks in state B will be allocated, leading to data lost in the hidden volume. It seems that the over-write issue cannot be completely addressed for the hidden volume-based PDEs [33, 34, 44, 6], since the hidden volume is part of the public volume. DEFTL further mitigates this issue by tuning the garbage collection strategy. Specifically, DEFTL performs an active garbage collection over those blocks in state C to reclaim the space occupied by the invalid pages. When the number of invalid pages in a block in state C reaches a certain threshold, DEFTL will copy the valid data from this block to a block in state D. After all the pages in this block are invalidated, it will turn to state A (arrow 3 of Figure 5). Correspondingly, it will be relocated to the head of dirty block table (Figure 4).

5.5 PDE Mode

The PDE mode is used to manage the hidden volume, which stores the sensitive data whose existence needs to be denied. In the following, we elaborate several key operations in this mode, including hidden volume mounting/unmounting, block allocation, garbage collection, and bad block management. These operations incorporate our design considerations in the PDE mode for deniability purpose.

Before elaborating different operations, we first answer a remained question about where to store the hidden volume metadata (including both the mapping table and the dirty block table). Two design concerns are: 1) they will not (or very unlikely) be over-written by the public data; and 2) they can be localized in the PDE mode. We choose to store them at the last several free blocks in the dirty block table of the public volume. Note that in the public mode, the free blocks include blocks in both state A and B. This solution can address the aforementioned concerns, because: First, it is very unlikely that the public data will over-write them since the public mode allocates free blocks from the head of the dirty block table. Second, they can be localized in the PDE mode by reading the public volume metadata. Although the adversary may also be able to localize them. Without the true key, he/she will not be able to decrypt the blocks storing those metadata, and cannot differentiate them from the blocks full of randomness.

Mounting the hidden volume. Hidden volume is encrypted with a true key, which is derived from a true password using a key derivation function (e.g., PBKDF2 [19]. During booting, the user can enter the true password to activate PDE mode. Using the true password, DEFTL can derive the true key. DEFTL further localizes the hidden volume metadata, and decrypts them using the true key. If the decryption is successful (refer to Sec. 5.4 for the approach about how to check whether the decryption is successful or not), the true key is correct and the hidden volume can be mounted. Otherwise, the system aborts from mounting the hidden volume. Using the mapping table of the hidden volume, the blocks storing hidden sensitive data (i.e, the blocks in state B) will be localized, and the entire hidden volume will be decrypted. Note that from the view of upper layer, the hidden volume appears to be a regular block device, which can be used to deploy any block-based file systems.

Block allocation. In the PDE mode, DEFTL allocates blocks according to the hidden volume dirty block table. The dirty block table stores the count of valid pages (i.e., the pages which store valid hidden data) for the blocks in state A and B, and organizes those blocks according to their counts in an increasing order. Note that each time when mounting the hidden volume, DEFTL will adjust the hidden volume dirty block table according to the public volume dirty block table, such that the blocks in state A and B will be exactly the same in both tables. When new sensitive data are written and new blocks need to be allocated, DEFTL selects free blocks from the dirty block table from the tail of the blocks in state A (Figure 6). Excluding blocks in state C and D from the hidden volume dirty block table is necessary, as it can prevent the hidden data from over-writing the data stored in the public volume. When a free block is allocated, it needs to be erased first if it has not yet been erased before. When the new data have been written, the mapping table/dirty block table need to be updated correspondingly. Especially for the dirty block table, the blocks should be always organized according to the count of valid pages in each block in an increasing order.

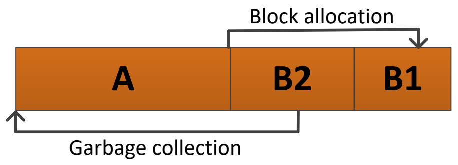

Figure 6: The dirty block table of the hidden volume

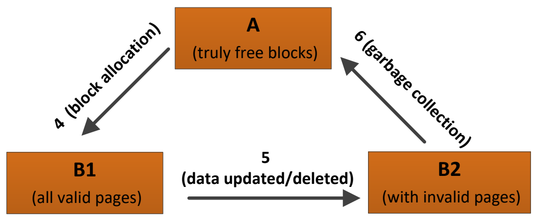

Figure 7: Block state transition in the PDE mode

We further differentiate the blocks in state B into two sub-types: B1 - the blocks which only contain valid hidden data; B2 - the blocks which contain both valid and invalid hidden data. As it is shown in Figure 7, the free blocks being allocated will be used to store valid hidden data, turning to B1 (arrow 4 in Figure 7). The blocks in state B1 may be updated/deleted, turning to B2 (arrow 5 in Figure 7).

Garbage collection. To ensure an enough amount of blocks in state A, DEFTL needs to reclaim the space occupied by the invalid pages from the blocks in state B2. This can be achieved by an active garbage collection in the PDE mode. Specifically, when the number of invalid pages in a B2 block reaches a certain threshold, DEFTL will copy the valid data from this block to a B1 block. After all the valid pages in this victim block are reclaimed, it will turn to state A (arrow 6 of Figure 7). Correspondingly in the dirty block table, it will be relocated to the head of the blocks in state A (Figure 6). To eliminate deniability compromises caused by the garbage collection (Sec. 3.1.2), DEFTL erases the reclaimed block immediately.

Bad block management. When a flash block is identified as "bad", DEFTL will replace it using a free block. The valid data from this bad block will be copied to the free block, and the mapping table will be updated. To eliminate deniability compromises from bad block management (Sec. 3.1.2), DEFTL will immediately erase this bad block [39].

Unmounting the hidden volume. The PDE mode will create a few blocks in state B, which are unknown to the public mode. Therefore, the public mode may allocate those blocks to store data written to the public volume. To reduce this probability, upon unmounting the hidden volume, DEFTL will modify the dirty block table of the public volume in the following manner: 1) keep the positions of the blocks in state C and D unchanged; 2) relocate the positions of the blocks in state A and B according to the dirty block table of the hidden volume, so that they are exactly the same in both tables. Adjusting the public volume dirty block table is to ensure that *the public mode will not use the blocks in state B until all the blocks in state A have been used, significantly reducing the chance of over-writing.* In addition, to avoid deniability compromises (Sec. 3.1.1), DEFTL will find out those blocks in state B which have not yet been fully used, and fill the empty pages in those blocks with random data. This can ensure that those blocks will be indistinguishable from blocks truly filled with randomness from the view of the public mode. DEFTL also needs to fill those empty pages (i.e., pages with all “1” bits) in blocks of state A with randomness, to prevent the amount of randomness from decreasing (Sec. 3.1.3).

5.6 User Steps

To activate the mobile device equipped with a flash-based block device, the user will enter either the decoy or the true password. How to differentiate the decoy and the true password has been well discussed in the existing hidden volume-based mobile PDE systems [33, 34, 44, 6]. Note that the user can either memorize the password, or use a smart card if the password is long and hard to be memorized. When a password is entered, the bootloader will inform DEFTL to boot into the corresponding mode. There are a few reserved command block wrappers (CBW) in the SCSI command [11], which can be used to pass the commands from the upper layer to DEFTL.

For regular daily use, the user enters the decoy password to activate the public mode. When storing sensitive data, the user quits the public mode (i.e., unmounts the public volume and logs out), and enters the true password. A potential booting-time attack which may compromise deniability was mitigated by adding dump operations during password authentication [44]. Note that during initialization, the user should first enter the public mode, initialize the public volume metadata, and then enter the PDE mode to initialize the hidden volume metadata. Once the user and his/her mobile device is captured, the adversary will coerce him/her for the secret being used to decrypt the device. At this point, the user can disclose the decoy password to avoid being tortured. DEFTL aims to ensure that by using the decoy password and the device being captured, the adversary is not able to detect the existence of the hidden sensitive data.

6 Analysis and Discussion

6.1 Security Analysis

The hidden volume technique (Sec. 2.3) achieves deniability by hiding an encrypted hidden volume (using a true key) among an encrypted public volume (using a decoy key which can be disclosed to the adversary). The encrypted hidden volume cannot be identified since it cannot be differentiated from the randomness being filled initially. We show in the following that DEFTL can provide deniability in the flash medium layer comparable to the hidden volume technique.

Theorem 6.1. *Under a single-snapshot adversary, the deniability provided by DEFTL in the flash medium layer is comparable to the hidden volume technique.*

Proof. (sketch). when the adversary captures both the victim and the mobile device, he/she coerces the victim, and the victim discloses the decoy password. The adversary can attack DEFTL and try to compromise deniability in three cases:

Case 1: The adversary obtains a snapshot of the raw flash, and performs forensic analysis over it. For hidden volume technique, without using the decoy key, the adversary cannot learn more from the storage state except random bits, if the output of the encryption algorithm is random enough. For DEFTL, when performing the aforementioned attack on the raw flash, the adversary cannot learn more except randomness and regular data without any deniability indications. Random data include: 1) data encrypted using the decoy key; and 2) data encrypted using the true key; and

3) randomness filled by DEFTL. Without using the decoy/true key, the encrypted ciphertexts are indistinguishable from randomness if the output of the encryption algorithm is random enough. The regular data without any deniability indications include: 1) the system metadata stored in the few reserved blocks at the beginning of the flash medium, which are regular metadata for any FTLs and clearly do not contain any deniability indications; 2) a few empty pages with all "1" bit pages due to block erasure, which is regular in a full disk encryption over flash, and also do not contain any deniability indications. Therefore, under this attack, the adversary obtains an equivalent storage state for both the hidden volume technique and DEFTL in terms of deniability.

Case 2: Using the decoy password, the adversary enters the public mode, decrypts the public data, and performs forensic analysis on the raw flash. Under this attack, the storage state (in raw flash) resulted from DEFTL is equivalent to that resulted from the hidden volume technique. From the view of the adversary, the entire flash blocks are filled with public data/metadata and randomness, which logically form a volume which is equivalent to the public volume in the hidden volume technique. The flash blocks storing the hidden sensitive data cannot be localized by the adversary since the hidden volume metadata cannot be decrypted by the adversary. In addition, the adversary cannot differentiate the encrypted hidden data from randomness without the true key. Therefore, all the blocks for hidden volume logically form a volume equivalent to the hidden volume in the hidden volume technique.

Case 3: The adversary enters the public mode, and plays with it to try to compromise deniability. Specifically, the adversary tries to read/write/delete data in the public mode, and observes the change of the storage state in the raw flash. We show that by performing this attack, the adversary will not gain any advantages in detecting indications present in the raw flash which can be used to compromise deniability. Clearly, read operations will not create any impact on the flash state. Write operations will affect block allocation and garbage collection, and then create impacts on the flash state. Delete operations will also create impacts on the flash state since it will affect garbage collection. However, since both the block allocation and garbage collection in DEFTL follow the design of a regular FTL without deniability, the adversary will not be able to observe any "special" indications from the flash state and suspect the existence of PDE.

□

Theorem 6.1 confirms that DEFTL is able to provide deniability in the flash medium layer. Considering that there are no deniability compromises under the flash medium layer[2], we conclude that DEFTL is secure.

6.2 Discussion

Wear leveling. In both the public mode and the PDE mode, wear leveling will be triggered each time when the erasure count of a block (we call it wear leveling victim block) being allocated exceeds a certain threshold over the average erasure count. The wear leveling will be performed as follows: 1) select a young block (i.e., a block with a small erasure count) from the blocks which store valid data. These blocks correspond to the blocks in state C and D in the public mode, and the blocks in state B in the PDE mode, respectively; 2) copy the valid data stored in the young block to the wear leveling victim block; 3) update the mapping table and the dirty block table; 4) allocate this young block instead. The blocks reserved at the beginning of the flash medium will not have significant

[2]The raw NAND flash is the lowest layer in a flash-based storage system we can access at this point. The potential layer under the raw flash is out of the scope of this paper.

wear leveling issue, because: first, the root table and the bad block table will be updated rarely; second, the erasure count table can be only updated during unmounting a volume by temporarily storing/updating the erasure count information using RAM. Note that a flash-based block device is usually equipped with a certain amount of built-in RAM [28, 29].

Encryption issues. As for encryption, DEFTL can integrate with FDE technology. The encryption algorithm (e.g., AES-XTS) can make the cipher output of data indistinguishable from random data [33].

Defending against a multiple-snapshot adversary. DEFTL is not able to defend against a multiple-snapshot adversary, who can have periodical access to the storage medium of the victim device. This is because, by comparing multiple snapshots, the adversary can be able to detect unaccountable changes in the flash storage state. For example, a block located at the end of the free blocks in the dirty block table should be unlikely used during a short period according to the block allocation strategy in the public mode. However, the adversary may observe changes on this block in a subsequent snapshot (e.g., obtained in a few hours after the last snapshot) and may suspect that it is used by the PDE mode. Defending against a multiple-snapshot adversary is much more challenging, which will be investigated in our future work.

P/E cycles increase. To prevent randomness from decreasing, DEFTL fills randomness to empty pages in the blocks in state A and B, which will increase the P/E cycles of the blocks. The increased P/E cycles can be eliminated by one-block-one-key policy, which can only be implemented in FTL rather than block device layer. By using this strategy, the attack described in Sec. 3.1.2 will not happen, since the same data will be re-encrypted using a different key before being copied to another block. However, it will produce a huge overhead in FTL by introducing additional key management as well as data encryption/decryption.

System crash. We assume that the adversary cannot capture a device after a crash of PDE mode, e.g., due to power loss. Otherwise, the adversary can observe a non-deniable state, and compromise deniability. For example, when the PDE mode is performing garbage collection, the system crashes before the block being reclaimed has been erased. By capturing the device at this point, the adversary is able to observe that duplicate random data (the data encrypted by the true key) exist at two different blocks.

7 Implementation and Evaluation

7.1 Implementation

We implemented a prototype of DEFTL using OpenNFM [9], an open source NAND flash controller framework. Note that DEFTL is applicable to any block-based flash device using FTL, though we use OpenNFM for our prototype implementation. To incorporate DEFTL, we modified OpenNFM to support two modes, a public mode and a PDE mode. Each volume in these two modes maintains a specific mapping table. We ported DEFTL to LPC-H3131 [22], a development board equipped with 180 MHz ARM microcontroller, 512MB NAND flash[3], and 32MB SDRAM. The NAND flash has 128KB block size and 2KB page size. The entire NAND flash has $4,096$ erase blocks, and each

[3]The mobile devices like smart phones and smart watches today can have a few gigabytes in storage, and 500MB may be a little small compared to the storage capacity of those main-stream mobile devices. We choose LPC-H3131 due to two reasons: 1) it is cost effective; 2) the main purpose of our evaluation is to assess the additional overhead by incorporating deniability, which is not significantly affected by the capacity of the storage.

block is composed of 64 pages. Each mapping table is less than 576KB in size, as it contains less than $4,096 * 64$ mappings, each of which can be represented by 18 bits. Our encryption algorithm is instantiated using AES-XTS, while our key derivation function is instantiated using PBKDF2 [19].

We benchmarked the original OpenNFM and different modes of DEFTL using fio [12] with the non-buffered I/O option. The fio is run in a host computer with Intel i5 CPU at 3.30GHz, 4GB RAM, and Windows 10 Pro 64-bit.

7.2 Performance Evaluation

DEFTL introduces a few additional strategies into the FTL to achieve deniability. To figure out how those strategies affect the performance of regular FTL (i.e., does not provide deniability), we perform evaluation over three systems: 1) the default OpenNFM (no deniability); 2) the public mode of DEFTL; and 3) the PDE mode of DEFTL. Note that encryption/decryption will be performed in the upper layer (e.g., file system), rather than in FTL, since the computational power in a flash-based block device is limited[4].

Throughput. To ensure an enough number of blocks in state A such that data loss of hidden volume can be mitigated, DEFTL adopts active garbage collection. If the number of blocks that contain invalid pages (i.e., the blocks in either state C or B2) has reached a threshold, DEFTL will trigger the garbage collection. We performed multiple experiments by alerting the garbage collection threshold, and compared the performance for the three systems. The benchmark results for their read/write throughput are shown in Figure 8 and 9. We observed that the reading (including both sequential and random read) throughput of both the public and the PDE mode of DEFTL are almost the same as OpenNFM. This is because, DEFTL does not need to modify read operation to achieve deniability. For write operation (including both sequential and random write), when the garbage collection threshold is small (e.g., 16), DEFTL slightly degrades in performance compared to OpenNFM. However, when the garbage collection threshold is increased, the performance will be improved. Potential reasons are:

a) To avoid blocking upcoming read/write requests for a long time, DEFTL only triggers active garbage collection during the system idle time. In this way, garbage collection will not significantly affect normal user access. If and only if the following two conditions are satisfied: 1) the total number of free blocks (blocks in state A and B) has decreased to a threshold, and 2) during the uninterrupted writing operations of the system, DEFTL will trigger the active garbage collection during system busy time. However, in this case, the large number of available victim blocks and invalid pages will help reduce the overhead (e.g., selecting victim blocks and copying valid pages) of garbage collection. Therefore, DEFTL will not significantly decrease the write throughput.

b) OpenNFM will search allocatable blocks for write requests only when they arrive. Comparatively, in DEFTL, we optimize the block allocation strategy by preparing the free allocatable blocks for upcoming write requests in advance. This helps decrease response time for write requests and slightly increase the write throughput in DEFTL.

Finally, we want to emphasize that *the experimental results of DEFTL do not include the overhead for encryption/decryption*. That is due to the poor performance of LPC-H3131 on encryption/decryption, which will significantly bias the results. We suggest the readers to refer to [7] for the impact of encryption/decryption on the throughput when the board is equipped with encryption/decryption module. That is why it seems from the results that the performance of DEFTL

[4]If the flash device is equipped with encryption hardware module, one may consider to perform encryption/decryption in FTL, which only slightly affects the overall read/write throughput [7]

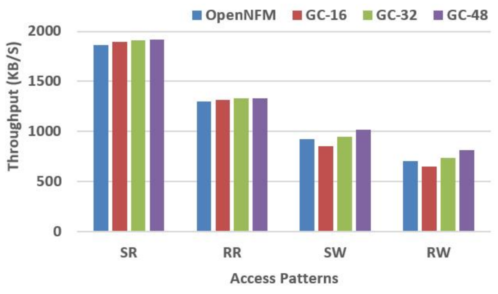

Figure 8: Throughput comparison between OpenNFM and the public mode of DEFTL. SR - sequential read, RR - random read, SW - sequential write, RW - random write. GC-XX, in which "XX" means the threshold of garbage collection in DEFTL

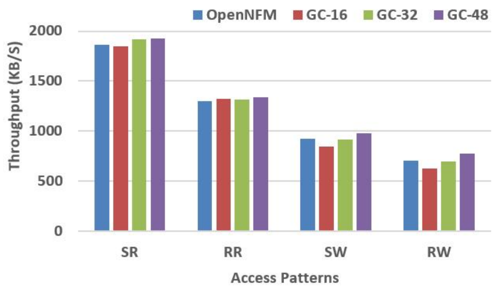

Figure 9: Throughput comparison between OpenNFM and the PDE mode of DEFTL

is better than OpenNFM, a system which does not provide deniability. When incorporating the overhead of encryption/decryption, the performance of DEFTL should be worse than OpenNFM.

Wear leveling. To distribute writes evenly across the entire flash (Sec. 2.2), DEFTL utilizes a global wear leveling strategy. We use Hoover economic wealth inequality indicator, which was used in previous work [32, 7] to evaluate wear leveling effectiveness. This metric calculates an appropriately normalized sum of the difference of each measurement to the mean. For flash memory, it indicates the fraction of erasures that must be re-assigned to other blocks in order to achieve completely even wear. Assuming the erasure counts of all the n erase blocks are f_1, f_2,..., f_n, and $F = \sum_{i=1}^{n} f_n$, the wear leveling inequality (WLI) can be computed as: $WLI = \frac{1}{2}\sum_{i=1}^{n}\|\frac{f_i}{F} - \frac{1}{n}\|$.

We evaluated the wear leveling effectiveness of DEFTL by varying the wear leveling threshold. To compute the WLI, we follow these steps: 1) Choose a wear leveling threshold; 2) Fill the entire flash with approximately 500MB data (we call this process a wear leveling test epoch). In DEFTL, we write data to both the public volume and the hidden volume; 3) Erase all the data being written; 4) Repeat each epoch until 250GB data has been written in total; 5) Compute WLI. In addition to WLI, we also calculated the average number of erasures per block performed during each epoch.

The results are shown in Table 1, under different wear leveling (WL) thresholds. We have following observations: 1) When the threshold decreases, a more active wear leveling will be performed. Meanwhile, more blocks will be erased and erasures will be distributed more evenly among blocks. Therefore, the increasing of average erasures per block and decreasing of WLI can be observed

Table 1: Wear leveling effectiveness of DEFTL

Wear leveling threshold	Average erasures (per test epoch)	WLI (%)
200	0.97	11.5
150	1.06	10.2
100	1.10	8.9
50	1.15	7.3

when the threshold decreases. 2) Under different wear leveling thresholds, DEFTL has small WLIs (around 10% as shown in Table 1). This implies that DEFTL can achieve an acceptable wear leveling.

8 Related Work

Deniable encryption was originally explored by Canetti et al. [5] in communications. In data storage, there are mainly two types of PDE systems: steganography-based and hidden volumes-based.

Steganography-based PDEs. Anderson et al. proposed the first steganographic file system [1]. They present two solutions: hiding sensitive data within cover files or within random data. StegFS [23] used the second approach to work on EXT2 file system with a block allocation table to track files. Pang et al. [30] introduced another steganographic file system, which uses an unencrypted global bitmap to ensure that blocks are not accidentally overwritten. All the steganography file systems try to hide sensitive data among regular file system, which may result in data loss of hidden files, as they may be overwritten by the regular file data. To mitigate the risk of data loss, they need to maintain a large amount of redundant data, which will lead to inefficient use of disk space. The inefficient use of disk space and increased IO operations make them unacceptable for mobile devices [33].

Hidden volume-based PDEs. Two well-known desktop PDE tools are TrueCrypt [38] and FreeOTFE [13]. TrueCrypt is sensitive to the type of file systems to avoid over-writing of the hidden volume. Blass et al. [3] present HIVE, which relies on the expensive write-only oblivious RAM. HIVE suffers from a high system overhead and is thus not suitable for mobile devices. Zhao et al. [45] present Gracewipe, which relies on a Trusted Platform Module (TPM) and CPU's trusted execution mode (e.g., Intel TXT). It requires extra special secure hardware and thus is not suitable for mobile devices.

Mobiflage is the first hidden volume-based PDE scheme for mobile devices. It is implemented in two versions: one for FAT32 file system in external storage [33], and the other for EXT4 file system in internal storage [34]. The FAT32 version is not suitable for mobile devices without external storage; the EXT4 version needs to significantly modify EXT4 file system that introduces a large attack surface against PDE. MobiHydra [44] improves Mobiflage by addressing a new booting-time attack and adding multi-level deniability. MobiPluto [6] further introduces a file system friendly PDE design by combining the hidden volume technology and thin provisioning.

Other PDEs. Peters et al. [31] introduced DEFY, a deniable encrypted file system based on YAFFS2, a flash-specific and single-threaded file system which directly handles raw NAND flash. DEFY is vulnerable to the deniability compromises described in Sec. 3.2. In addition, it is incompatible with the flash-based block device, which is the most popular form of flash storage media being used in computing devices today. This is because, the deniability achieved in DEFY strongly relies on the system properties offered by YAFFS2, which is unfortunately rarely used nowadays.

9 Conclusion

In this paper, we propose DEFTL, a Deniability Enabling Flash Translation Layer for devices that use flash-based block devices as storage media. DEFTL is the first design that incorporates deniability into the FTL, a pervasively deployed "translation layer" which stays between the physical flash layer and the file system layer in literally all the computing devices. A salient advantage of DEFTL is that, for the first time, it eliminates deniability compromises from the underlying flash medium, and meanwhile, accommodates the special nature of flash memory. Experimental evaluation confirms the efficiency of DEFTL, compared to the encryption storage for flash without deniability support.

References

[1] Ross Anderson, Roger Needham, and Adi Shamir. The steganographic file system. In *International Workshop on Information Hiding*, pages 73–82. Springer, 1998.

[2] Apple. Filevault. `https://support.apple.com/en-us/HT204837`, 2015.

[3] Erik-Oliver Blass, Travis Mayberry, Guevara Noubir, and Kaan Onarlioglu. Toward robust hidden volumes using write-only oblivious ram. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 203–214. ACM, 2014.

[4] Marcel Breeuwsma, Martien De Jongh, Coert Klaver, Ronald Van Der Knijff, and Mark Roeloffs. Forensic data recovery from flash memory. *Small Scale Digital Device Forensics Journal*, 1(1):1–17, 2007.

[5] Rein Canetti, Cynthia Dwork, Moni Naor, and Rafail Ostrovsky. Deniable encryption. In *Annual International Cryptology Conference*, pages 90–104. Springer, 1997.

[6] Bing Chang, Zhan Wang, Bo Chen, and Fengwei Zhang. Mobipluto: File system friendly deniable storage for mobile devices. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 381–390. ACM, 2015.

[7] Bo Chen, Shijie Jia, Luning Xia, and Peng Liu. Sanitizing data is not enough: towards sanitizing structural artifacts in flash media. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 496–507. ACM, 2016.

[8] Siddharth Choudhuri and Tony Givargis. Deterministic service guarantees for nand flash using partial block cleaning. In *Proceedings of the 6th IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, pages 19–24. ACM, 2008.

[9] Google Code. Opennfm. `https://code.google.com/p/opennfm/`, 2011.

[10] CodePlex. Veracrypt ssd. `https://veracrypt.codeplex.com/`, 2017.

[11] Computerhope. Scsi. https://www.computerhope.com/jargon/s/scsi.htm, 2017.

[12] Freecode. fio. http://freecode.com/projects/fio, 2014.

[13] FreeOTFE. FreeOTFE - Free disk encryption software for PCs and PDAs. version 5.21. *Project website:* `http://www.freeotfe.org/`, 2012.

[14] P Keith Garvin and H Duane Stanard. Method and system for managing bad areas in flash memory, July 10 2001. US Patent 6,260,156.

[15] Peter Gutmann. Secure deletion of data from magnetic and solid-state memory. In *Proceedings of the Sixth USENIX Security Symposium, San Jose, CA*, volume 14, pages 77–89, 1996.

[16] J. Assange, R.P. Weinmann, and S. Dreyfus. Rubberhose Filesystem. *Archive available at:* `https://web.archive.org/web/20100915130330/http://iq.org/~proff/rubberhose.org/`, 2001.

[17] Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. Nfps: Adding undetectable secure deletion to flash translation layer. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 305–315. ACM, 2016.

[18] Sanghyuk Jung and Yong Ho Song. Link-gc: a preemptive approach for garbage collection in nand flash storages. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1478–1484. ACM, 2013.

[19] B. Kaliski. Pbkdf2. https://tools.ietf.org/html/rfc2898, 2000.

[20] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. A flash-memory based file system. In *USENIX*, pages 155–164, 1995.

[21] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, 2015.

[22] Mantech. Lpc-h3131. http://www.mantech.co.za/, 2017.

[23] Andrew D McDonald and Markus G Kuhn. Stegfs: A steganographic file system for linux. In *Information Hiding*, pages 463–477. Springer, 2000.

[24] ST Microelectronics. Bad block management in nand flash memories. *Application note AN-1819, Geneva, Switzerland*, 2004.

[25] Microsof. Bitlocker. `https://technet.microsoft.com/en-us/library/hh831713.aspx`, 2013.

[26] MTD. Ubifs. `http://www.linux-mtd.infradead.org/doc/ubifs.html`, 2015.

[27] J. Mull. How a syrian refugee risked his life to bear witness to atrocities. In *toronto Star Online*, 2012.

[28] OpenSSD. Jasmine openssd platform. `http://www.openssd-project.org/wiki/Jasmine_OpenSSD_Platform`, 2011.

[29] OpenSSD. Cosmos openssd platform. `http://www.openssd-project.org/wiki/Cosmos_OpenSSD_Platform`, 2014.

[30] HweeHwa Pang, K-L Tan, and Xuan Zhou. Stegfs: A steganographic file system. In *Data Engineering, 2003. Proceedings. 19th International Conference on*, pages 657–667. IEEE, 2003.

[31] Timothy M Peters, Mark A Gondree, and Zachary NJ Peterson. DEFY: A deniable, encrypted file system for log-structured storage. In *22th Annual Network and Distributed System Security Symposium, NDSS*, 2015.

[32] Joel Reardon, Srdjan Capkun, and David Basin. Data node encrypted file system: Efficient secure deletion for flash memory. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 17–17. USENIX Association, 2012.

[33] Adam Skillen and Mohammad Mannan. On implementing deniable storage encryption for mobile devices. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27*, 2013.

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

[35] Source. Android full disk encryption. `https://source.android.com/security/encryption/`, 2016.

[36] Avinash Srinivasan, Jie Wu, Panneer Santhalingam, and Jeffrey Zamanski. Deaddrop-in-a-flash: Information hiding at ssd nand flash memory physical layer. *SECURWARE 2014*, page 79, 2014.

[37] Raja Subramani, Haritima Swapnil, Niharika Thakur, Bharath Radhakrishnan, and Krishnamurthy Puttaiah. Garbage collection algorithms for nand flash memory devices–an overview. In *Modelling Symposium (EMS), 2013 European*, pages 81–86. IEEE, 2013.

[38] TrueCrypt. Free open source on-the-fly disk encryption software.version 7.1a. *Project website: http://www.truecrypt.org/*, 2012.

[39] Chundong Wang and Weng-Fai Wong. Extending the lifetime of nand flash memory by salvaging bad blocks. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 260–263. EDA Consortium, 2012.

[40] Yinglei Wang, Wing-kei Yu, Shuo Wu, Greg Malysa, G Edward Suh, and Edwin 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*, pages 33–47. IEEE, 2012.

[41] Michael Yung Chung Wei, Laura M Grupp, Frederick E Spada, and Steven Swanson. Reliably erasing data from flash-based solid state drives. In *FAST*, volume 11, pages 8–8, 2011.

[42] Yaffs. Yaffs. `http://www.yaffs.net/`, 2002.

[43] Ming-Chang Yang, Yu-Ming Chang, Che-Wei Tsao, Po-Chun Huang, Yuan-Hao Chang, and Tei-Wei Kuo. Garbage collection and wear leveling for flash memory: past and future. In *Smart Computing (SMARTCOMP), 2014 International Conference on*, pages 66–73. IEEE, 2014.

[44] Xingjie Yu, Bo Chen, Zhan Wang, Bing Chang, Wen Tao Zhu, and Jiwu Jing. Mobihydra: Pragmatic and multi-level plausibly deniable encryption storage for mobile devices. In *Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14, 2014. Proceedings*, pages 555–567, 2014.

[45] Lianying Zhao and Mohammad Mannan. Gracewipe: Secure and verifiable deletion under coercion. In *Network and Distributed System Security Symposium*, 2015.