A Cross-layer Plausibly Deniable Encryption System for Mobile Devices

Niusen Chen[1], Bo Chen[1(✉)], and Weisong Shi[2]

[1] Department of Computer Science, Michigan Technological University, Michigan, USA
bchen@mtu.edu

[2] Department of Computer Science, Wayne State University, Michigan, USA

Abstract. Mobile computing devices have been used to store and process sensitive or even mission critical data. To protect sensitive data in mobile devices, encryption is usually incorporated into major mobile operating systems. However, traditional encryption can not defend against coercive attacks in which victims are forced to disclose the key used to decrypt the sensitive data. To combat the coercive attackers, plausibly deniable encryption (PDE) has been introduced which can allow the victims to deny the existence of the sensitive data. However, the existing PDE systems designed for mobile devices are either insecure (i.e., suffering from deniability compromises) or impractical (i.e., unable to be compatible with the storage architecture of mainstream mobile devices, not lightweight, or not user-oriented).

In this work, we design CrossPDE, the first cross-layer mobile PDE system which is secure, being compatible with the storage architecture of mainstream mobile devices, lightweight as well as user-oriented. Our key idea is to intercept major layers of a mobile storage system, including the file system layer (preventing loss of hidden sensitive data and enabling users to use the hidden mode), the block layer (taking care of expensive encryption and decryption), and the flash translation layer (eliminating traces caused by the hidden sensitive data). Experimental evaluation on our real-world prototype shows that CrossPDE can ensure deniability with a modest decrease in throughput.

Keywords: PDE · Mobile devices · Coercive attacks · Confidentiality · Cross-layer · Flash memory

1 Introduction

With the increased use of mobile computing devices, a large amount of sensitive data are collected, stored, and managed in them. To protect sensitive data, full disk encryption (FDE) has been integrated into major mobile operating systems including Android and iOS. FDE transparently encrypts/decrypts all the user data at the block layer and, without having access to the secret key, an adversary will not be able to learn the sensitive data even if the adversary can steal the

© ICST Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2023
Published by Springer Nature Switzerland AG 2023. All Rights Reserved
F. Li et al. (Eds.): SecureComm 2022, LNICST 462, pp. 150–169, 2023.
https://doi.org/10.1007/978-3-031-25538-0_9

entire disk. FDE however, cannot defend against a coercive attacker who can capture the device owner and coerce the owner for the secret key. For example, a human right worker may be forced to disclose the encrypted data stored in his/her smartphone when crossing the border of a country in conflict. To defend against the coercive attacker, plausibly deniable encryption (PDE) was proposed. Its main idea is, the sensitive data are encrypted in such a way that, only if the *true* secret key is used for decryption, the original sensitive data will be revealed, but if a *decoy* key is used, the decryption will result in some non-sensitive data; therefore, when a device owner is coerced, the owner can simply disclose the decoy key, protecting the true key as well as the hidden sensitive data.

To implement PDE in a mobile device, currently there are three options: 1) deploying the PDE at the block layer of a mobile device [8,9,17,22,31,37] (Category I), and 2) integrating the PDE with a flash-specific file system YAFFS [11,30] (Category II), and 3) integrating the PDE with the flash translation layer [10,12,23,25] (Category III). The mobile PDE systems in the Category I are **insecure** due to the potential deniability compromises when the adversary can have access to the internal flash memory and extract traces of hidden sensitive data which are invisible to the block layer [14,23]. The mobile PDE systems in the Category II are strongly coupling with YAFFS which is rarely used in today's mobile computing devices; instead, a vast majority of the existing mobile computing devices (including the ever-growing IoT devices) use flash memory cards via flash translation layer (FTL) and, the Category-II PDE systems are **incompatible** with this mainstream flash storage architecture. The mobile PDE systems in the Category III integrate the entire PDE (i.e., with expensive operations like disk encryption [10,23,25], WOM codes [12] or dummy writes [10,25]) into the FTL, turning it a "**heavyweight**" software component. However, the FTL was originally designed for handling unique nature of NAND flash, which is usually run by low-end internal hardware [18] of a flash-based block device (e.g., a microSD card), and the heavyweight FTL may significantly decrease the I/O throughput of the PDE system. In addition, the Category-III mobile PDE systems are located at the lower FTL layer which stays far away from users staying at the application layer, and hence may be difficult to be managed by users (i.e., **not user-oriented**).

The limitations observed from the existing mobile PDE systems motivate us to re-consider the PDE system design in a holistic manner. Our resulted design, CrossPDE, is the first mobile PDE system which simultaneously satisfies four properties: (P1) *resistance against the coercive attackers*, i.e., the design is secure even if the adversary can have access to the internal flash memory; and (P2) *being compatible* with the architecture of mainstream mobile computing devices; and (P3) *keeping the FTL lightweight*; and (P4) *being user-oriented.* Our key idea is to decouple the PDE functionality, and to separate them among different storage layers (i.e., the file system layer, the block device layer, and the FTL layer) of a mobile device. The outcome is the first cross-layer PDE system design for mobile devices. To be resistant against the coercive attackers (P1), we distill the minimal functionality necessary for eliminating deniability compromises on the flash memory, and place it to the FTL layer. In this way,

the FTL remains thin and the extra overhead imposed on the less powerful [18] internal hardware of the flash-based block device will be minimized (P3). Note that the expensive PDE functionality including disk encryption/decryption as well volume management are conducted at the block layer which will be run by the more powerful hardware of the host computing device. In addition, the file system layer provides an immediate interface for the user to manage the PDE functionality at the lower layers (P4). Last, such a design is immediately compatible with mainstream mobile computing devices using flash memory cards via FTL (P2).

However, after decoupling the PDE functionality and moving them across multiple storage layers, we face a few new challenges:

1) To avoid deniability compromises, the FTL should be informed if the user is working in a hidden mode which manages hidden sensitive data. However, the user is typically located at the application layer, and how can he/she securely convey such information to the low-layer FTL? This issue becomes more challenging due to the deployment of disk encryption at the block layer, since everything going through the block layer will be encrypted transparently. To resolve this challenge, we have reserved a few logical block addresses which are accessible to the FTL, and used the file system as a bridge to issue I/Os on the reserved block addresses with secret patterns, so that the messages from the application layer can be conveyed stealthily to the FTL.
2) The loss of hidden sensitive data is a general issue for all PDE systems and, in our setting, this issue turns even more challenging, due to the separation of PDE functionality across multiple layers. Resolving this challenge requires coordinating the file system layer, the block layer and the FTL layer. By hiding an encrypted hidden volume at the end of an encrypted public volume, and deploying in the public volume a file system which has a low probability of writing the end of disk by nature, we can significantly mitigate loss of hidden sensitive data, without touching the lower FTL layer.
3) Eliminating all potential deniability compromises in the flash memory is challenging. We have identified two new deniability compromises in the flash memory and carefully mitigated all the known compromises discovered to date.

Contributions. We summarize our contributions as follows:

- We have discovered new PDE compromises in the underlying flash memory of mobile devices which have not been identified in the literature. We have also provided novel mitigation strategies.
- We have designed the first cross-layer PDE system that meets all following requirements: 1) being secure, and 2) being compatible with the mainstream storage architecture of mobile devices, and 3) imposing small burden on the underlying flash device and, 4) providing interface for users to manage the system.
- We have implemented CrossPDE by modifying and integrating a few open-source projects. In addition, we have ported our prototype to a real-world testbed for mobile devices to assess its performance.

2 Background and Related Work

2.1 Background Knowledge

Flash Memory. NAND flash has dominated storage media of today's mobile devices due to its high I/O speed and low noise. The NAND flash is usually divided into blocks (a few hundreds of KBs in size), and each block is divided into pages (a few KBs in size). Each flash page has a small out-of-band (*OOB*) area, which can store extra information like error correction code. Compared to regular hard disk drives (HDD), NAND flash exhibits some unique characteristics: 1) The unit of I/O is a page, but the unit of erasure is a block. 2) A flash block can not be re-programmed before it is erased. Therefore, flash memory typically performs out-of-place instead of in-place updates. 3) A flash block can only be programmed/erased for a limited number of times.

Flash Translation Layer (FTL). To use flash memory, the most popular approach today is to emulate it as a block device via flash translation layer (FTL). In this way, traditional block-based file systems (e.g. FAT32, EXT4) can be directly deployed. The FTL stays between the file system and the raw NAND flash, and implements four core functions: address translation, garbage collection, wear leveling, and bad block management.

Flash memory performs out-of-place updates, i.e., for an overwrite operation performed by the OS, the FTL will place the new data to a empty page, and invalidate the page storing the old data. From the OS's view, the logical block address (*LBA*) of the data remains the same. However, the physical block address (*PBA*) of the data has been changed. Therefore, the FTL needs to keep track of mappings between LBAs and PBAs for *address translation*. In addition, since the overwrite operations will invalidate flash pages storing the old data, *garbage collection* is needed to reclaim those blocks with a large number of invalid pages. Each flash block can be only programmed/erased for a limited number of times. Therefore, we need a mechanism which can distribute programmings/erasures (P/Es) evenly across the entire flash to prolong its service life. *Wear leveling* is such a mechanism which can even out P/Es among flash blocks by relocating frequently updated data to blocks with less P/Es. Flash memory is vulnerable to wear and, over time, a flash block may turn "bad" making it unable to reliably store data. *Bad block management* can manage those bad blocks.

Full Disk Encryption (FDE). FDE encrypts/decrypts the entire disk transparently to users. FDE includes both software-based and hardware-based disk encryption. The software-based FDE is usually deployed at the block layer, so that any data written to or read from the disk can be transparently encrypted or decrypted. Popular implementations include TrueCrypt [2], BitLocker [27], etc.

Plausibly Deniable Encryption (PDE). To implement PDE, we can use a steganographic file system, which hides sensitive data in either regular files or randomness arbitrarily filled. We can also use *the hidden volume technique*. The entire disk is filled with random data initially. Two volumes, a public and a

hidden volume, are deployed on the disk. The public volume is used to store non-sensitive data, and the hidden volume is used to stored sensitive data. The public volume is encrypted via FDE using a *decoy key* and placed across the entire disk. The hidden volume is encrypted via FDE using a truly secret key (i.e., *true key*), and placed to the end of the disk starting from a secret offset. In view of the public volume, *the space filling with the randomness is just the empty space and can be used to store public data*. Therefore, the public data may overwrite the hidden data, causing data loss. Upon being coerced, the device owner can simply disclose the decoy key; the adversary uses the decoy key to decrypt the public volume, but is unaware of existence of the hidden volume.

2.2 Related Work

Upper-Layer PDE Systems. Steganographic file systems [4,5,21,26,29] hide sensitive data among either regular files or random data, and maintain additional redundancies of hidden data across the disk to avoid their loss. Image steganography has also been leveraged to construct PDE systems [13]. VeraCrypt [3]/TrueCrypt [2] introduces a hidden volume technique, which hides sensitive data in a dedicated volume that is stored hidden at a secret offset towards the end of the disk. Mobiflage [31,32] extends the hidden volume technique for Android OS. Other follow-up works enhance the mobile PDE systems supporting various features, e.g., multi-level deniability [22,37], file system friendliness [8], dynamic mounting of hidden volumes [17,22]. Major limitations of the aforementioned PDE systems are, they are purely deployed at upper layers, and do not consider deniability compromises in the underlying flash memory.

Lower-Layer PDE Systems. DEFY [30] and INFUSE [11] both integrated a PDE design into flash file system YAFFS, which unfortunately is rarely used nowadays. DEFTL [23] and PEARL [12] have moved the PDE to the FTL, but all of them suffer from some common drawbacks: 1) They impose a significant burden on the flash memory firmware (managed by low-end internal hardware of the flash device), rendering them impractical for broad deployment. Especially, DEFTL performs the expensive disk encryption and decryption purely in the FTL. PEARL achieves PDE by encoding (i.e., WOM codes) both the public and the hidden data together in the FTL and, since both public and hidden data are "entangled", I/Os on either one would be expensive. 2) They do not consider upper layers and the user is difficult to manage the PDE staying in the FTL. Liao et al. [25] proposed a TrustZone-enhanced mobile PDE system which isolates sensitive data in the memory to avoid memory leaks. However, they still heavily rely on the FTL to isolate the sensitive data in the external storage.

3 Model and Assumptions

System Model. We consider a mobile computing device (Fig. 1) which is equipped with a flash-based block device, e.g., an MMC/eMMC card, an SD/miniSD/microSD card, or a UFS card. We do not consider powerful flash

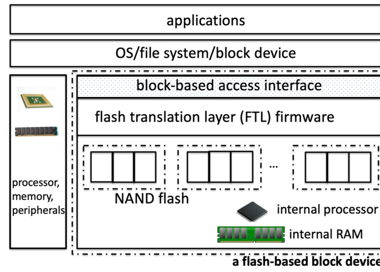

Fig. 1. The architecture of a mainstream mobile device.

devices like SSDs, which are typically used in the more powerful personal computers. Each flash device is equipped with its internal processor and RAM, and manages the raw NAND flash via the FTL. It usually exposes a block-based access interface, so that conventional block-based file systems like EXT4, FAT32, NTFS can be seamlessly deployed on top of it.

Adversarial Model. We consider a computationally-bounded adversary, which can capture a victim user and his/her mobile device. The adversary can access the external storage of the device, and coerce the user for keys to decrypt any encrypted sensitive data. The adversary does not trust the user and may try all means to identify the existence of PDE. For example, the adversary may enter the public mode and use it as a regular user to check any traces of hidden data; the adversary may check the file system in the public mode for anything abnormal in the file hash; moreover, the adversary may perform forensic analysis [24] on the disk. For analysis purposes, we assume the adversary can acquire a copy of the raw flash memory image via state-of-the-art laboratory techniques [6].

Assumptions. We rely on a few common assumptions which are also required in prior mobile PDE systems [8,23,31]: 1) The adversary is assumed to be rational and will stop coercing the victim once convinced that the decryption key is disclosed [31]. 2) The adversary cannot capture a victim user when he/she is right working in the hidden mode; otherwise, the hidden data are disclosed trivially. 3) We assume the bootloader and the OS are not infected by the malware controlled by the adversary; otherwise, the malware can monitor the system and trivially know the existence of PDE. In addition, the user will not use untrusted apps controlled by the adversary while working in the hidden mode, in case that sensitive information about the hidden mode will be leaked to those apps. 4) We assume the adversary will not perform reverse engineering over the code of the device after capturing it. To prevent the adversary from capturing the victim device multiple times and correlating the disk images captured each time to compromise PDE, we assume each time after the user is caught and released, he/she will take various actions including but not limited to: disconnecting the device from the network, scanning the device via antivirus tools, copying out the data, conducting a factory reset, etc.

4 CrossPDE: A Cross-layer Mobile PDE System

4.1 Design Rationale

Typically, we can rely on either the steganographic file system or the hidden volume technique to build a mobile PDE system. We choose the hidden volume technique (Sect. 2.1) which is more I/O efficient and fits the mobile devices better: First, the steganographic file system requires maintaining multiple redundant copies of hidden data across the disk to mitigate data loss, leading to significant storage overhead. On the contrary, the hidden volume technique does not require maintaining redundant sensitive data by smartly hiding them at the end of the disk. Second, the steganographic file system incurs significant overhead when writing the hidden data due to writing the redundant copies. On the contrary, the hidden volume technique can efficiently write the hidden data as no redundant writes are needed. Using the hidden volume technique, there are two modes. A *public mode* and a *hidden mode*, will be introduced which allow the user to manage the public and the hidden volume, respectively. Upon booting, if the user provides the decoy key, the OS will mount the public volume and the system will enter the public mode; otherwise if the user provides the true key, the OS will mount the hidden volume and the system will enter the hidden mode.

This simple adoption of the hidden volume technique is insufficient as the adversary can compromise PDE by having access to the underlying flash memory. Unlike prior works [12,23] which integrate the entire PDE with the flash translation layer to avoid the aforementioned compromise, we instead divide the PDE functionality and to separate them into multiple layers: the flash translation layer will manage flash blocks associating with the hidden mode to eliminate any deniability compromises; the block layer will manage both volumes and perform disk encryption/decryption; the file system layer will manage the file system deployed on each volume, and act as a bridge for the user to manage the hidden mode in lower layers of the storage system. An overview of our design is shown in Fig. 2, with three key ideas elaborated below:

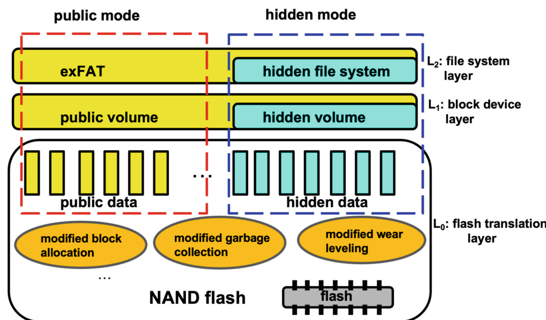

Fig. 2. An overview of our design.

Idea 1: Mitigating Loss of Hidden Sensitive Data. Loss of hidden data is a general problem for any PDE systems [4], as sensitive data are hidden among public data and, to ensure plausible deniability, the public mode should not be aware of the existence of the hidden data, and may overwrite them unintentionally. To avoid data loss at the block layer, we embed the hidden volume at the end of the disk, with three extra considerations: 1) The public volume and the hidden volume should be managed by a separate file system, i.e., a *public file system* for the public volume, and a *hidden file system* for the hidden volume. 2) To prevent the public file system from writing the end of the disk, we choose exFAT as the public file system, a mobile file system which writes data sequentially from the beginning of the disk, and has a low probability of overwriting the sensitive data stored hidden at the end of the disk. 3) The user is suggested to pay attention to the disk space used by the public data, because the public data are allowed to use the entire space of the disk (Sect. 2.1) and, if the disk is filled, the hidden data will be overwritten unavoidably.

A unique hardware feature of mobile devices is the use of flash memory, which is encapsulated inside the flash-based block device. Therefore, preventing data loss merely at the block layer may not be sufficient. We need to ensure that there is no data loss at the flash translation layer (FTL) as well. We argue that by embedding a hidden volume at the end of the disk and deploying exFAT as the public file system, we will not suffer from loss of hidden sensitive data at the FTL because: The entire disk (i.e., the block layer) is initially filled with random data, and from the view of the FTL, those random data are written by upper layers and hence are all valid. The sensitive data written to the hidden volume are stored stealthily among random data and, the flash blocks storing them will not turn invalid if they are not overwritten by the public file system deployed at the block layer. Our deployed public file system exFAT has a low probability of writing the end of the disk, and hence has a low probability of overwriting the hidden volume stored stealthily at the end of the disk. Therefore, the flash blocks storing hidden sensitive data will not be turned invalid and hence will not be reclaimed by garbage collection of the FTL.

Idea 2: Thoroughly Eliminating Deniability Compromises in the Flash Translation Layer. Merely deploying a hidden volume at the block layer will suffer from deniability compromises as the underlying flash translation layer (FTL) will not be aware of the existence of the hidden volume at the block layer and hence will not hide those traces created by the hidden data [23]. Prior works

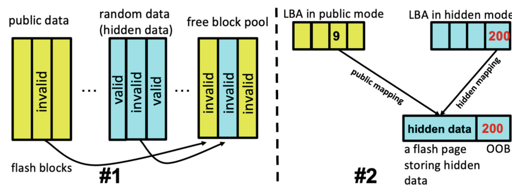

Fig. 3. Newly discovered deniability compromises in the flash memory.

have identified a few such compromises [14,23]. We have discovered two new compromises which have not been identified before (see Fig. 3):

New Deniability compromise #1: Initially, the hidden volume technique fills the entire disk with randomness which establishes an initial mapping[1] between the block layer and the flash memory blocks. At the block layer, the public file system writes at the beginning of the disk; therefore in the flash memory, the public data will occupy those blocks at the beginning of the flash memory. In addition, the hidden file system writes data at the end of the disk; therefore in the flash memory, the hidden data will occupy those blocks at the end of the flash memory. Without PDE, as the public file system writes data sequentially from the beginning of the disk, only the blocks at the beginning of the flash memory may be invalidated and moved to the free block pool. However, with PDE, the user may enter the hidden mode to delete/overwrite hidden data, and the blocks located at the end of the flash memory may be invalided and move to the free block pool. Such a difference may lead to compromise of PDE.

New Deniability Compromise #2: The hidden volume is part of the public volume. Therefore, a flash page used by the hidden volume data may be also used by the public volume data. The effect is, a flash page which stores hidden data may be mapped to two different LBAs, one for the public volume and the other for the hidden volume. Note that the FTL typically maintains a mapping table keeping track of mappings between LBAs and PBAs; to allow restoring this table upon sudden failures (e.g., power loss), the OOB area of each flash page will also keep its corresponding LBA. Therefore, when writing a flash page in the hidden mode, the OOB area of the page will keep track of its corresponding LBA of the hidden volume. This will be detected by the adversary, leading to compromise of PDE.

To mitigate the compromise #1, our strategy is: when working in the hidden mode, the FTL will not move flash blocks to the free block pool. Especially, the FTL in the hidden mode will work differently with the modified functions (i.e., the block allocation, garbage collection, wear leveling function specified for the hidden mode) and new data structures (i.e., the mapping table specified for the hidden mode). To mitigate the compromise #2, our strategy is: when writing a flash page in the hidden mode, the FTL will always commit the flash page's corresponding LBA of the public mode (rather than that of the hidden mode) to its OOB. The downside is that the OOB of the flash pages in the hidden mode cannot be used to restore the mapping table maintained in this mode upon sudden failures. This downside can be alleviated by embedding the corresponding LBA of the hidden mode to the content of each flash page.

Besides our newly discovered compromises, there is one compromise identified by [14], without a mitigation strategy being provided [14]. The compromise comes from a special type of flash block which is completely filled with undecryptable randomness, with a few pages in arbitrary locations of the block

[1] Data stored at the beginning of the block layer are mapped to those blocks at the beginning of the flash memory, as the system usually fills randomness sequentially from the beginning of the disk, and the FTL uses a log-structured writing strategy.

invalidated. This type of flash block is generated when the user modifies some of hidden sensitive data. To mitigate this compromise, we introduce an independent data structure to keep track of pages invalidated by the hidden data, and this data structure is only visible to the hidden mode. In other words, a page invalidated by the hidden data still appears as valid in the public mode. Finally, to ensure all the deniability compromises can be eliminated, we also handle those old compromises identified in [23] via strategies introduced in their work, including: 1) the hidden data will not share flash blocks with public data; 2) and if the hidden data cannot fill a flash block upon quitting the hidden mode, the remaining space of this block will be filled with randomness.

Idea 3: Secure Cross-Layer Communication. Our design is cross-layer and, therefore, components of the hidden mode will stay at different layers and need to communicate with each other securely. Especially, when entering the hidden mode, the user should securely inform the FTL that he/she is now in the hidden mode and the FTL should actively eliminate special traces in the flash memory caused by hidden data; when quitting the hidden mode, the user should inform the FTL as well. A strawman solution is that, the user crafts a special string and writes it to the disk via the regular "write" system call. The FTL will monitor any write requests issued by the block layer and, once it detects this special string, it will know that the user has conveyed a request. This solution is problematic because: The hidden volume is encrypted by FDE at the block layer and, all data written to the hidden volume will be encrypted before passing to the FTL; therefore, to search this special string, the FTL may need to decrypt all the data being received, which is expensive.

Our solution is, in the hidden mode, the user issues a request to the hidden file system and, upon receiving such a request, the hidden file system will pass it to the FTL. To allow the hidden mode to communicate with the hidden file system without affecting existing system calls, we create a unique file in the hidden file system, and the hidden mode can issue I/Os on this file via the regular system calls; the hidden file system can monitor I/Os on this special file to communicate with the hidden mode. To allow the hidden file system to send a request to the FTL without affecting the existing I/O interface of the block device, we reserve a few special LBAs, and the hidden file system will issue I/Os on the reserved LBAs via the regular block I/O interface; the FTL can monitor I/Os on the reserved LBAs to communicate with the hidden file system. To prevent this "hidden interface" of the FTL from being abused, authentication needs to be incorporated. Specifically, this "hidden interface" can only be activated if I/Os with a secret pattern are performed on the reserved LBAs and only the hidden file system knows this secret pattern.

4.2 Design Details

Following the rationale, we have designed CrossPDE, the first cross-layer mobile PDE system. CrossPDE separates PDE functionality into major layers of a mobile storage system: the flash translation layer (L_0), the block layer (L_1), and the file system layer (L_2). Design details of each layer are elaborated below:

L_0 : Flash Translation Layer. CrossPDE modifies a few major functions in the FTL to support PDE. The bad block management function does not cause deniability compromises and therefore, we do not need to modify it.

Block Allocation. *In the public mode*, the FTL uses the log-structured writing, which typically writes public data to blocks from the beginning of the entire flash. *In the hidden mode*, the block allocation should be carefully performed to avoid deniability compromises. There are a few rules: 1) When writing hidden data, the FTL should not use empty pages from those blocks occupied by the public data. 2) When writing hidden data, the FTL should use those blocks located at the end of the flash (they are typically mapped to the areas located at the end of the block layer which are very unlikely overwritten by the public mode). Especially, the FTL in the hidden mode will allocate blocks in a reverse direction starting from the end of the flash, excluding the free blocks reserved initially. When allocating a block[2], the FTL will read all the LBAs from the OOBs of this block (also copy out the valid data if there are any, which need to be written back after block erasure), and immediately erase this block. The hidden data will be written sequentially to empty pages of this block. Note that when writing a page in the hidden mode, we will reuse the LBA of the public mode, and commit it to the corresponding OOB to avoid deniability compromises (Sect. 4.1). The empty pages of the block will be used until they are exhausted in the hidden mode. If the user quits the hidden mode and there are still unused empty pages in a block, the empty pages should be filled with randomness. A special case is an overwrite on the existing hidden data, in which the corresponding flash pages should be first invalidated. For deniability, the FTL should maintain an independent data structure (i.e., page validity table) keeping track of which pages are invalidated in the hidden mode. The page validity table is only used by the hidden mode and remains invisible to the public mode.

Garbage Collection. The garbage collection is performed periodically during the idle time. *In the public mode*, the garbage collection runs as follows: The FTL finds a dirty block which has the largest number of invalid pages, copies all valid data in this block to a free block, and places the dirty block to the free block pool. *In the hidden mode*, however, the garbage collection should be performed differently, since its dirty blocks should not be placed to the free block pool (Sect. 4.1). Especially, among those flash blocks storing hidden data, the FTL will find a dirty block which has the largest number of invalid pages; it will then handle the dirty block as follows: 1) It reads all the LBAs from the OOBs of the dirty block, and copies out all the valid data from this block; 2) It erases the dirty block; 3) It writes the valid data back to the block, sequentially from the first page; the remaining empty pages should be filled with randomness. When writing each page, the original LBA should be committed to its OOB.

[2] Those blocks which are 1) reserved for the hidden mode, and 2) entirely or partially filled with actual randomness, can be allocated.

Wear Leveling. *In the public mode*, the wear leveling runs as follows: Upon a certain wear leveling threshold is reached, the FTL: 1) selects a block (X) which is currently in use with the smallest erasure count; and 2) selects another block (Y) from free block pool with the largest erasure count; and 3) erases block Y and copies all data from block X to block Y; and 4) updates the mapping table. The rationale is that the data stored in block X is cold and should be relocated to block Y, which has the largest erasure count. *In the hidden mode*, wear leveling needs to be implemented differently as the blocks for the hidden mode cannot be placed to the free block pool (Sect. 4.1). When a certain threshold is reached, the FTL selects a block (X) with the largest erasure count and a block (Y) with the smallest erasure count among blocks for the hidden mode; it then exchanges the data between block X and block Y (under the help of RAM or a free block, but the free block should be cleaned after it). Note that: 1) The hidden mode should maintain its own table for keeping track of erasure counts for its reserved blocks, and this erasure count table is invisible to the public mode. 2) The wear leveling in the public mode usually will not use blocks reserved for the hidden mode, as it only swaps blocks storing public data with those in the free block pool, but the blocks storing hidden data will never enter the free block pool.

Other Operations. The FTL monitors I/Os issued by the upper layer on some reserved LBAs. If such I/Os have been detected, the FTL will determine the request type based on the I/O patterns. The most important requests are "start" and "quit" request. For the "start" request, the FTL knows that the hidden mode is activated, and starts to use the data structures (e.g., the mapping table, the page validity table, the erasure count table) and functions (block allocation, garbage collection, wear leveling) specifically for the hidden mode. For the "quit" request, the FTL knows that the hidden mode terminates. It will identify the block occupied by the hidden data but has not been completely filled, and fill the empty pages with randomness.

$\underline{L_1 : \text{Block Layer}}$. The public/hidden volume is deployed on the block layer. Both volumes are encrypted by full disk encryption (run by the processor and memory of host computing device) via the decoy and the true key, respectively. The public volume will be managed by the public mode via the public file system and, any data written by the user in the public mode will be passed down by the public file system, and encrypted transparently with the decoy key at the block layer before being passed to the FTL. Similarly, the hidden volume will be managed by the hidden mode via the hidden file system and, any data written by the user in the hidden mode will be passed down by the hidden file system, and encrypted transparently with the true key at the block layer. Reading data from both volumes will be performed in a reverse manner.

$\underline{L_2 : \text{File System Layer}}$. *In the public mode*, we deploy exFAT, a block-based mobile file system which writes data sequentially from the beginning of disk. *In the hidden mode*, we can deploy any block-based file system on the hidden volume. This hidden file system acts as a "bridge" between the user working in

the hidden mode and the lower storage layers. To enable this bridge, we modify the hidden file system as follows: We maintain a "special file", which is created when the user enters the hidden mode for the first time. Note that the name for this special file should be unique and different from other files in the system, and a large enough random number can be used for this file name. The hidden file system will monitor I/Os on this special file and, once an I/O request is issued by the user on the file, it will determine the request type and issue I/Os (for different user requests, the hidden file system will use different secret I/O patterns) to the reserved LBAs. Note that we can easily convert the sector addresses on the block layer to the LBAs, e.g., if the sector size is 512 bytes, and the page size is 2KB, each sector address is translated to the LBA by dividing 4.

4.3 User Steps

To process non-sensitive data, the user should boot into the public mode via the decoy key. The user should use this mode regularly to ensure a better plausibility [31]. To process sensitive data, the user should boot into the hidden mode via the true key. Upon entering the hidden mode, the user can issue a "start" request to the "special file" maintained by the hidden file system, and the hidden file system will then issue a "start" request downwards; similarly, upon quitting the hidden mode, the user can issue a "quit" request to the "special file", and the hidden file system will then issue a "quit" request downwards. To prevent traces of hidden sensitive data from remaining in the memory, the user is suggested to power-off the device upon quitting the hidden mode.

5 Analysis and Discussion

Security Analysis of CrossPDE. We first show that by running the public mode, the adversary is not able to identify the existence of PDE. Using the decoy key coerced from the victim, the adversary can boot into the public mode, and can have access to all data files, configuration files, system logs, file system metadata, etc. However, all the aforementioned data belong to the public mode and, none of the data belonging to the hidden mode can be found as both modes are strictly isolated. In addition, the adversary may perform forensic analysis over the memory (e.g., extracting the memory content using memdump) in the public mode. This would not help, as CrossPDE requires the user to shut down the device when quitting the hidden mode and the traces should have been eliminated from the memory. The adversary may also analyze the raw data on the disk (the block layer), but will not be able to identify the existence of the hidden volume which is stored stealthily among the randomness.

We also show that by analyzing the raw data on the flash memory, the adversary is not able to identify the existence of PDE. By modifying the major functionality (e.g., block allocation, garbage collection, wear leveling) of the FTL, CrossPDE successfully eliminates all the traces caused by the hidden mode, so that a flash block storing hidden sensitive data cannot be differentiated from a

flash block storing random data. Therefore, by analyzing the raw data on the flash memory, the adversary can only identify 3 types of flash blocks: 1) a flash block filled with public non-sensitive data; and 2) a flash block stores public data at the beginning and the remaining pages are empty; and 3) a flash block filled with (valid) random data. This is no different from a flash storage medium which is initially filled with randomness and has an FDE (via decoy key) deployed on the entire disk. In addition, the adversary is not able to identify the existence of PDE in the free block pool as well as the OOB areas of flash pages.

Mitigating Multi-snapshot Adversaries. CrossPDE can defend against a multi-snapshot adversary which can access the victim device multiple times, assuming that the victim is alert and will reset the device each time after being captured and released. To reset the device, the victim will 1) back up the data, and 2) conduct a full reset to clear both the memory and the external storage (via secure deletion [20]), and 3) re-fill new randomness and re-write the public and the hidden data back to the device, encrypted with a new decoy and true key, respectively. Such a reset operation allows the victim to plausibly deny the changes over the empty space of the disk.

Denying the Existence of a Partition Filled with Random Data. CrossPDE requires filling the entire disk with randomness initially. A plausible explanation can be, the user has securely erased the content in the partition using a tool which erases data by overwriting it with random data [28].

Mitigating Timing Attacks. Yu et al. [37] discovered a booting-time attack, which may happen when authenticating a given key (decoy or true key) for entering the corresponding mode. The reason is: given the decoy key, the public mode can be entered fast, as the bootloader will always try to boot the public volume first; however, given a wrong key, the bootloader will return slowly, as it first tries to boot the public volume and then the hidden volume, and finally returns with an error prompt which takes more time. To obfuscate this time difference, we can add extra time delay when booting with the decoy key [37].

Protecting the PDE Code. CrossPDE relies on an assumption that the adversary will not conduct reverse-engineering attacks over the code. To relax this assumption, a potential solution is to leverage the obfuscation technique [36] to obfuscate the code, concealing its purpose of PDE. This will be further investigated in our future work.

Pre-boot Authentication. To enter either the public or the hidden mode, the user needs to provide the corresponding key for authentication (i.e., the *pre-boot authentication*). We can derive the decoy/true key from the corresponding password [31], and choose strong passwords following certain security guidelines [33].

The password-based pre-boot authentication however, will essentially reduce the security provided by CrossPDE, as a memorable password implies that the adversary would be easier to guess it. Another option is to use NFC cards [7] to store keys so that the user does not need to memorize them.

6 Implementation and Evaluation

6.1 Implementation

We have implemented CrossPDE by integrating and modifying a few open-source software projects: OpenNFM [15] for the flash translation layer, VeraCrypt [3] for the block layer, and exFAT [16] for the file system layer. OpenNFM was used as the FTL. VeraCrypt was used to manage (e.g., create, encrypt, etc.) both the public and the hidden volume at the block layer. It also took care of initialization and pre-boot authentication. The exFAT was deployed as the file system for both the public and the hidden volume and, especially for the hidden volume, we deployed a modified exFAT, such that the hidden mode can communicate with the FTL using exFAT as a bridge (this implies that VeraCrypt and OpenNFM should be modified accordingly).

Modifications to OpenNFM. For the hidden mode, we implemented the new block allocation, garbage collection, and wear leveling, but reused the existing bad block management. We also modified the FTL_Read function, so that once the reserved LBAs are read, the FTL knows that there is a request from the user in the hidden mode. We implemented different read patterns to differentiate the starting and the quitting request.

Modifications to VeraCrypt. Each time when mounting a volume, we first check whether it is the public volume or the hidden volume. If it is the public volume, it will be mounted as usual. Otherwise, we will perform an I/O on a special file (we will create this special file when entering the hidden volume for the first time). In addition, when unmounting the hidden volume, we will perform another I/O on this special file.

Modifications to exFAT. We modified the function exfat_get_block() in super.c so that exFAT can monitor I/Os on the special file and, once it detects an I/O on this file, it will issue I/Os on some reserved disk sector addresses which will be translated deterministically to some reserved LBAs in the flash memory. In our case, the disk sector address can be translated to a corresponding LBA by dividing 4, considering a disk sector is 512 bytes in size and a flash page is 2KB in size. Note that exFAT is not in the kernel (V4.4.194) of Firefly AIO 3399J originally, but we made exFAT as a kernel module when re-compiling the kernel.

6.2 Evaluation

Experimental Setup. We ported our developed prototype to a self-built mobile device testbed [14], which consists of a flash-based block device and a host computing device. The flash-based block device was built using a USB header development prototype board LPC-H3131 [1] (ARM9 32-bit ARM926EJ-S, 180Mhz, 32MB RAM, and 512MB NAND flash) and our modified OpenNFM as the FTL [34,35]. The host computing device was an embedded development board, Firefly AIO-3399J (Six-Core ARM 64-bit processor, 4GB RAM, and Linux kernel 4.4.194). Our modified exFAT and modified VeraCrypt were deployed to the Firefly AIO-3399J. The LPC-H3131 connects to the USB 2.0 interface of Firefly AIO-3399J via a USB A to Mini Cable. Using the VeraCrypt, we created both the public and the hidden volume, and the original exFAT was deployed on the public volume, and the modified exFAT was deployed on the hidden volume. For comparison, we created a baseline by deploying the original VeraCrypt on top of the same testbed, in which the original exFAT was used in both the public and the hidden volume, and the original OpenNFM was used as the FTL. We believe that VeraCrypt is representative, as other block-based mobile PDE systems [8,17,22,31,37] all implement a similar technique. For simplicity, we call the baseline "VeraCrypt", which is vulnerable to deniability compromises in the flash memory. The I/O throughput of CrossPDE and VeraCrypt were both measured using benchmark tool fio [19]. We also compared CrossPDE with the represented FTL-based PDE systems DEFTL [23] and PEARL [12].

We compare the I/O throughput of CrossPDE with VeraCrypt in Table 1. We can observe that: 1) For the public mode, CrossPDE exhibits similar I/O throughput with VeraCrypt, because we do not change the read/write operations in the public mode. 2) For the hidden mode, the read throughput of CrossPDE is similar to that of VeraCrypt, but the write throughput is reduced 50%–60%. This is because: The read operations of the hidden mode are similar to those of the public mode; however, the write operations in the hidden mode require extra steps including reading the LBAs, erasing the block, etc.

To justify the benefits of moving the disk encryption/decryption from the FTL to the block layer, we also evaluated the I/O throughput without disk encryption, i.e., "no encryption" (as implemented by the original OpenNFM [15]), as well as the I/O throughput when deploying disk encryption/decryption in the FTL [23]. Both results are shown in Table 2. From Table 1 and 2, we can observe that: compared to "no encryption", the throughput of CrossPDE decreases 3%–12% in the public mode, and 4%–64% in the hidden mode; but"encryption in the FTL" decreases the throughput more than 10× in both modes. This confirms that CrossPDE makes the FTL much more lightweight compared to those which perform disk encryption/decryption in the FTL.

To assess the benefits of CrossPDE in keeping the FTL lightweight, we have compared the I/O throughput among CrossPDE, DEFTL [23] and PEARL [12]. The comparison is shown in Table 3, in which we estimated the throughput decrease of each aforementioned PDE system compared to a normal system without a PDE deployed, based on our own experimental results as well as the

Table 1. Throughput comparison between VeraCrypt and CrossPDE. SR - sequential read; RR - random read; SW - sequential write; RW - random write

Patterns	VeraCrypt (KB/s)		CrossPDE (KB/s)	
	Public mode	Hidden mode	Public mode	Hidden mode
SR	2508	2473	2460	2424
RR	2174	2030	2086	2000
SW	2599	2372	2535	948
RW	1897	1842	1910	839

Table 2. Throughput of "no encryption" and "encryption in FTL"

Patterns	No encryption (OpenNFM)(KB/s)	Encryption in FTL (KB/s)
SR	2538	172
RR	2206	170
SW	2639	168
RW	2176	165

Table 3. Estimation of throughput decrease in different FTL-based PDE schemes, compared to a regular system without a PDE deployed.

	PEARL [12]	DEFTL [23]	CrossPDE
Public Read	41%	92%–93%	3.1%–5.4%
Public Write	48%	92.4%–93.6%	3.9%–12.2%
Hidden Read	80%	92%–93%	4.5%–9.3%
Hidden Write	90.4%	92.4%–93.6%	61%–64%

experimental results from PEARL. We can observe that: 1) For the public read, the public write and the hidden read, CrossPDE decreases slightly in throughput, but PEARL (41%–80% decreases) and DEFTL (more than 90% decreases) significantly decrease in throughput. 2) For the hidden write, CrossPDE has a modest decrease in throughput (61%–64%), but PEARL and DEFTL both significantly decrease in throughput (more than 90%). The comparison can justify that CrossPDE performs much better in I/O throughput by decoupling the PDE functionality and separating them across multiple layers of the mobile system. This is because: unlike DEFTL and PEARL, the expensive operations of PDE in CrossPDE are separated from the FTL and moved to the block layer and hence processed by the more powerful host computing device.

7 Conclusion

In this work, we propose CrossPDE, a cross-layer PDE system for mobile computing devices which has integrated the PDE functionality into major layers of a mobile storage system. Experimental evaluation on our developed real-world prototype shows that CrossPDE can ensure deniability with a modest decrease in performance compared to the insecure block-layer PDE systems.

Acknowledgments.. This work was supported by US National Science Foundation under grant number 1928349-CNS, 1928331-CNS, 1938130-CNS, and 2043022-DGE.

References

1. Lpc-h3131. https://www.olimex.com/Products/ARM/NXP/LPC-H3131/
2. Truecrypt. http://truecrypt.sourceforge.net/
3. Veracrypt. https://www.veracrypt.fr/code/VeraCrypt/
4. Anderson, R., Needham, R., Shamir, A.: The steganographic file system. In: Aucsmith, D. (ed.) IH 1998. LNCS, vol. 1525, pp. 73–82. Springer, Heidelberg (1998). https://doi.org/10.1007/3-540-49380-8_6
5. Barker, A., Sample, S., Gupta, Y., McTaggart, A., Miller, E.L., Long, D.D.E.: Artifice: a deniable steganographic file system. In: 9th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 19) (2019)
6. Breeuwsma, M., De Jongh, M., Klaver, C., Van Der Knijff, R., Roeloffs, M.: Forensic data recovery from flash memory. Small Scale Dig. Dev. Forensics J. **1**(1), 1–17 (2007)
7. Chang, B., et al.: User-friendly deniable storage for mobile devices. Comput. Secur. **72**, 163–174 (2018)
8. Chang, B., Wang, Z., Chen, B., Zhang, F.: Mobipluto: file system friendly deniable storage for mobile devices. In: Proceedings of the 31st Annual Computer Security Applications Conference, pp. 381–390 (2015)
9. Chang, B., et al.: Mobiceal: towards secure and practical plausibly deniable encryption on mobile devices. In: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 454–465. IEEE (2018)
10. Chen, B.: Towards designing a secure plausibly deniable system for mobile devices against multi-snapshot adversaries-a preliminary design. arXiv preprint arXiv:2002.02379 (2020)
11. Chen, C., Chakraborti, A., Sion, R.: Infuse: invisible plausibly-deniable file system for nand flash. Proc. Priv. Enhan. Technol. **4**, 239–254 (2020)
12. Chen, C., Chakraborti, A., Sion, R.: Pearl: plausibly deniable flash translation layer using wom coding. In: The 30th Usenix Security Symposium (2021)
13. Chen, N., Chen, B., Shi, W.: MobiWear: a plausibly deniable encryption system for wearable mobile devices. In: Chen, B., Huang, X. (eds.) AC3 2021. LNICST, vol. 386, pp. 138–154. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-80851-8_10
14. Chen, N., Chen, B., Shi, W.: The block-based mobile pde systems are not secure - experimental attacks. In: EAI International Conference on Applied Cryptography in Computer and Communications. Springer, Heidelberg (2022). https://doi.org/10.1007/978-3-031-17081-2_9

15. Google Code. Opennfm (2011). https://code.google.com/p/opennfm/
16. exfat file system specification. https://docs.microsoft.com/en-us/windows/win32/fileio/exfat-specification
17. Feng, W., et al.: Mobigyges: a mobile hidden volume for preventing data loss, improving storage utilization, and avoiding device reboot. Fut. Gener. Comput. Syst. **109**, 158–171 (2020)
18. Typical hardware of flash storage devices. https://snp.cs.mtu.edu/techdoc/flash-devices.html
19. Freecode. fio (2014). http://freecode.com/projects/fio
20. Gutmann, P.: Secure deletion of data from magnetic and solid-state memory. In: Proceedings of the Sixth USENIX Security Symposium, San Jose, CA, vol. 14, pp. 77–89 (1996)
21. Han, J., Pan, M., Gao, D., Pang, H.: A multi-user steganographic file system on untrusted shared storage. In: Proceedings of the 26th Annual Computer Security Applications Conference, pp. 317–326 (2010)
22. Hong, S., Liu, C., Ren, B., Huang, Y., Chen, J.: Personal privacy protection framework based on hidden technology for smartphones. IEEE Access **5**, 6515–6526 (2017)
23. Jia, S., Xia, L., Chen, B., Liu, P.: Deftl: implementing plausibly deniable encryption in flash translation layer. In: Proceedings of the 24th ACM Conference on Computer and Communications Security. ACM (2017)
24. Johnson, N.F., Jajodia, S.: Steganalysis: the investigation of hidden information. In: 1998 IEEE Information Technology Conference, Information Environment for the Future (Cat. No. 98EX228), pp. 113–116. IEEE (1998)
25. Liao, J., Chen, B., Shi, W.: Trustzone enhanced plausibly deniable encryption system for mobile devices. In: 2021 IEEE/ACM Symposium on Edge Computing (SEC), pp. 441–447. IEEE (2021)
26. McDonald, A.D., Kuhn, M.G.: StegFS: a steganographic file system for linux. In: Pfitzmann, A. (ed.) IH 1999. LNCS, vol. 1768, pp. 463–477. Springer, Heidelberg (2000). https://doi.org/10.1007/10719724_32
27. Microsof. Bitlocker (2013). https://technet.microsoft.com/en-us/library/hh831713.aspx
28. Plausible deniability. https://www.veracrypt.fr/en/Plausible%20Deniability.html
29. Pang, H., Tan, K.-L., Zhou, X.: Stegfs: a steganographic file system. In: Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405), pp. 657–667. IEEE (2003)
30. Peters, T.M., Gondree, M.A., Peterson, Z.N.J.: Defy: a deniable, encrypted file system for log-structured storage (2015)
31. Skillen, A., Mannan, M.: On implementing deniable storage encryption for mobile devices. In: 20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, 24–27 February 2013 (2013)
32. Skillen, A., Mannan, M.: Mobiflage: deniable storage encryption for mobile devices. IEEE Trans. Depend. Secure Comput. **11**(3), 224–237 (2014)
33. How to create a strong password (and remember it). https://www.howtogeek.com/195430/how-to-create-a-strong-password-and-remember-it/
34. Tankasala, D., Chen, N., Chen, B.: A step-by-step guideline for creating a testbed for flash memory research via lpc-h3131 and opennfm. Technical report, Department of Computer Science, Michigan Tech (2020)
35. Tankasala, D., Chen, N., Chen, B.: Creating a testbed for flash memory research via lpc-h3131 and opennfm - linux version. Technical report, Department of Computer Science, Michigan Tech (2022)

36. Wroblewski, G.: General method of program code obfuscation (2002)
37. Yu, X., Chen, B., Wang, Z., Chang, B., Zhu, W.T., Jing, J.: MobiHydra: pragmatic and multi-level plausibly deniable encryption storage for mobile devices. In: Chow, S.S.M., Camenisch, J., Hui, L.C.K., Yiu, S.M. (eds.) ISC 2014. LNCS, vol. 8783, pp. 555–567. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13257-0_36