A Cross-layer Plausibly Deniable Encryption System for Mobile Devices

No Author Given

No Institute Given

Abstract. Mobile computing devices have been used broadly 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). Security analysis and experimental evaluation show that CrossPDE can ensure deniability with a modest decrease in throughput.

Keywords: PDE, mobile devices, coercive attacks, confidentiality, flash memory

1 Introduction

Mobile computing devices like smartphones, tablets, smart watches are increasingly used today and have played an important role in processing personally private or even mission critical data. With their increased use, a large amount of sensitive data are collected, stored, and managed in those devices. 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 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, we need a new encryption mechanism which can allow a victim to deny the existence of the secret key, and the existence of sensitive data as well. Plausibly deniable encryption (PDE) was proposed for this purpose. 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 [9, 10, 18, 24, 36, 41] (Category I), and 2) integrating the PDE with a flash-specific file system YAFFS [12, 35] (Category II), and 3) integrating the PDE with the flash translation layer [13, 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, 25]. 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 main-stream flash storage architecture. The mobile PDE systems in the Category III integrate the entire PDE (i.e., with expensive operations like disk encryption [25] or WOM codes [13]) 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 [19] of a flash-based block device, 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 in the application layer, and hence are difficult to be managed by users (i.e., **not user-oriented**).

The limitations observed from the existing mobile PDE designs motivate us to re-consider this problem 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 [19] 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 main-stream 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 novel challenges: *First*, 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. *Second*, the loss of hidden sensitive data is a general issue for all the PDE systems and, in our setting, this issue becomes 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. *Third*, eliminating all the deniability compromises in the flash memory is a challenging issue. We have identified two new deniability compromises in the flash memory and carefully eliminated 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 which is secure and compatible with the storage architecture of mainstream mobile devices. Our design imposes much less burden on the underlying low-power flash device and, meanwhile, provides interface for users to manage the PDE 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 access 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 a unique nature: 1) The unit of I/Os in NAND flash is a page, but the unit of erasure is a block. 2) A flash block can not be re-programmed before it is erased. 3) A flash block can only be programmed or erased for a limited number of times. Overwriting data stored at a flash page is expensive, since it requires first erasing the entire encompassing block, which further requires copying out valid data in this page and writing them back after erasure, leading to significant write amplification. Therefore, flash memory usually performs *out-of-place* rather than in-place updates.
Flash translation layer (FTL). Traditional file systems created for HDDs cannot be directly used in flash memory, since the flash memory exhibits completely different characteristics from the HDDs. We therefore need a flash-specific file system which has been optimized for the nature and characteristics of flash memory. To use flash memory, a much more popular alternative today is to emulate the flash storage medium as a block device via flash translation layer (FTL) and, in this way, traditional block-based file systems like FAT and EXT can be 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 uses the out-of-place update strategy, i.e., for each overwrite operation performed by the OS, the FTL will place the data to a new page, and invalidate the page storing obsolete data. From the OS's view, the logical block address (*LBA*) of the data remains the same, but the physical block address (*PBA*) 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 invalid flash pages. The garbage collection usually follows a few steps: 1) selecting a victim block which has the most invalid pages, and 2) copying valid data from this victim block to a free block, and 3) erasing the victim block. 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 bad blocks, so that healthy data stored in the bad blocks can be relocated, and bad blocks will not be used again.

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 [39], VeraCrypt [3], BitLocker [31], etc. Android FDE uses dm-crypt [16], a Linux kernel module working at the block layer.

Plausibly deniable encryption (PDE). To implement PDE, we can use a steganographic file system [4, 23, 29, 34], which hides sensitive data in either regular files or randomness arbitrarily filled. As hidden sensitive data may be overwritten by regular file data, the system needs to maintain a few redundant copies of hidden data across the disk. We can also use **the hidden volume technique** [3,39]. Two volumes, a public and a hidden volume, are deployed on the disk. The public volume is used to store public non-sensitive data, while 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. In other words, *the public data will be using the entire disk space and, any remaining space in the disk not occupied by them remains usable, preventing the adversary from inferring the existence of the hidden volume by simply measuring available disk space*. 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. During initialization, the entire disk is filled with randomness, and the encrypted hidden volume will be hidden among the randomness. 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 discloses 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

Block-layer PDE systems. Steganographic file systems [4, 5, 23, 29, 34] hide sensitive data among either regular files or random data, and maintain additional redundancies of hidden data across the disk to avoid their loss. VeraCrypt [3]/TrueCrypt [39] hide sensitive data in a dedicated volume which is stored hidden at a secret offset towards the end of the disk, known as the hidden volume technique. Mobiflage [36,37] 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 [24, 41], file system friendliness [9], dynamic mounting of hidden volumes [18, 24]. MobiCeal [10] further addressed the multi-snapshot attack specifically for mobile devices. Major limitations of all those PDE systems are, they are deployed on the block layer, and do not consider deniability compromises in the underlying flash memory.

Flash-layer PDE systems. DEFY [35] and INFUSE [12] have both integrated a PDE design into the flash-specific file system YAFFS, which is rarely used nowadays. DEFTL [25] and PEARL [13] have moved the PDE to the FTL. Both

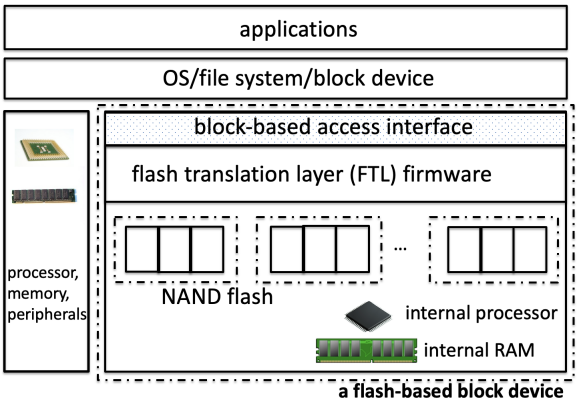

Fig. 1. The architecture of a main-stream mobile device.

unfortunately, suffer from common drawbacks: 1) They impose a significant burden on the flash memory firmware (managed by low-end internal hardware of the flash-based block 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. For example, for a (3,5) WOM code, to access 12KB hidden data, 60KB data need to be accessed; to access 12KB public data, 20KB data need to be accessed [13]. 2) They do not consider upper layers including the file system and the application layer, and the end users would be difficult to communicate with the PDE integrated into the FTL.

3 Model and Assumptions

System model. We consider a mobile computing device which is equipped with a flash-based block device. Such flash devices include MMC/eMMC cards, SD/miniSD/microSD cards, UFS cards, etc. Note that we do not consider powerful flash storage devices like SSDs, which are typically used in the more powerful personal computers. Each flash device is equipped with its own processor, 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. The architecture of the mobile device is shown in Figure 1.

Adversarial model. We consider a computationally-bounded adversary, which can capture a victim user and his/her mobile device. The adversary has access to the external storage of the device, and coerce the user for keys to decrypt any encrypted sensitive data stored. 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

anomaly in the file hash; moreover, the adversary may analyze [27] the disk to identify the existence of PDE. For analysis purposes, we assume the adversary can acquire a copy of the raw flash memory image via state-of-the-art laboratory techniques. Note that it is not difficult to obtain the raw image of a flash device. Breeuwsma et al. [6] introduce a few low-level data acquisition methods for flash memory, including flasher tools, using an access port commonly used for testing and debugging, etc. Chen et al. [13] refer to a method of obtaining raw data from SSDs "by opening the covers and directly reading the memory chips with cheap off the shelf readers".

Assumptions. We rely on a few common assumptions which are also required in prior mobile PDE systems [9, 25, 36]: 1) The adversary is assumed to be rational and will stop coercing the victim once convinced that the decryption key is disclosed [36]. 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 that 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 that the adversary will not perform reverse engineering over the code of the device after capturing it. Various software protection techniques including code obfuscation [40], code encryption [7] can be used to mitigate the aforementioned threat. However, to narrow down the research scope of this work, we would leave the code protection problem of PDE to our future work. 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 when the user is caught and released, he/she will take actions. For example, he/she suspects the adversary may have installed malware in the device; therefore, he/she disconnects the device from the network, copies the data out, and conducts a factory reset to re-initialize the entire device.

4 CrossPDE: A Cross-layer Mobile PDE System

4.1 Design Rationale

Typically, we can rely on the steganographic file system and the hidden volume technique to build mobile PDE systems. We choose the hidden volume technique (Sec. 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 sensitive 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 sensitive data due to writing the redundant copies. On the contrary, the hidden volume technique can efficiently write the hidden sensitive data as no redundant writes are needed. Using the hidden volume technique, a

public volume encrypted by a decoy key will be used to store non-sensitive data and, a hidden volume encrypted by a true key will be used to store sensitive data. Two modes, a *public mode* and a *hidden mode*, will 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. Our design rationale is to 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; 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. A few key insights for our design are elaborated as follows:

Key insight 1: mitigate loss of hidden sensitive data in both the file system and the block layer. 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 hidden sensitive 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 (Sec. 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 block device. Therefore, preventing data loss merely at the block layer is insufficient. 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 the upper layer 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 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 sensitive

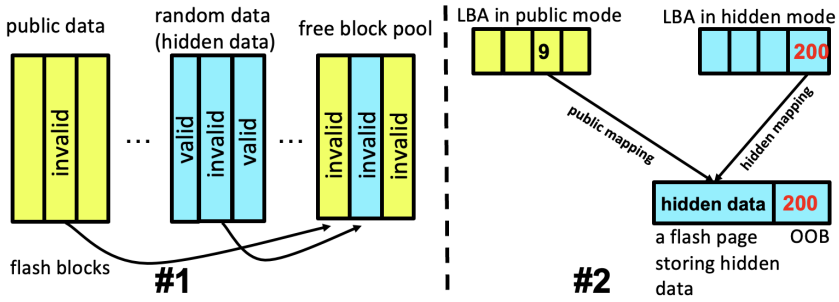

Fig. 2. Newly discovered deniability compromises in the flash memory.

data hidden at the end of the disk. Therefore, the flash blocks storing hidden sensitive data will not be turned invalid and hence will not be deleted by garbage collection of the FTL.

Key insight 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 [25]. Prior works have identified a few such compromises [14, 25]. We have discovered two new compromises which have not been identified before (see Figure 2):

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 data occupy the beginning of the disk, and the hidden data occupy the end of the disk. Correspondingly in the flash memory, the public data will occupy those flash blocks at the beginning of the flash memory, while the hidden data will occupy flash blocks at the end of the flash memory. Without PDE, the public file system will write data sequentially from the beginning of the disk, and hence the flash blocks at the beginning of the flash memory may be invalidated and swapped to the free block pool. However, with PDE, the user may enter the hidden mode to delete/overwrite hidden data, and the flash blocks located at the end of the flash memory may be invalided and swapped to the free block pool. This 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 sensitive 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 an independent mapping table which keeps track of mappings between LBAs and PBAs and, to allow restoring this mapping table upon sudden failures (e.g., power loss), the OOB area of each flash page will also keep track of its corre-

[1] The data stored at the beginning of the block layer are mapped to the flash blocks at the beginning of the flash memory, as the public file system writes data sequentially from the beginning of the disk, while the FTL uses the log-structured writing.

sponding LBA. Therefore, when writing a flash page in the hidden mode, the flash page's OOB area will keep track of the corresponding LBA of the hidden volume. This will be detected by the adversary when accessing the flash memory, 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 independently with its own functions (i.e., block allocation, garbage collection, and wear leveling) and data structures (i.e., a mapping table specific 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 flash block which is completely filled with un-decryptable randomness, with a few pages in arbitrary locations of the block 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 [25] via strategies introduced in their work.

Key insight 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 sensitive 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

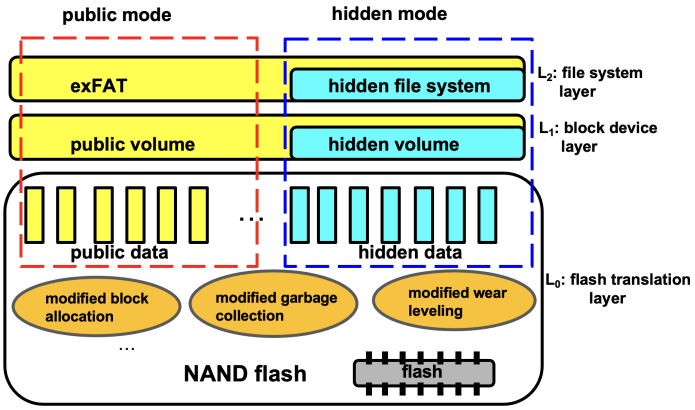

Fig. 3. An overview of CrossPDE.

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 aforementioned rationale, we have designed CrossPDE, the first cross-layer mobile PDE system. CrossPDE separates PDE functionalities into major layers of a mobile storage system (Figure 3): the flash translation layer (L_0), the block layer (L_1), and the file system layer (L_2). The detailed design of each layer is elaborated below.

L_0: Flash Translation Layer. CrossPDE modifies a few major functions in the FTL to support PDE.

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 designed 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 (those blocks 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.

As the entire flash has been filled with randomness initially, upon allocating a block, for each page of the block, the FTL will read the LBA stored in its OOB, and immediately erase this block. The hidden data will be written sequentially to each page of this block, and the corresponding LBA should be committed back to the OOB of each page (i.e., we do not update its OOB area with the new LBA of the hidden mode; instead, we reuse its old LBA of the public mode). When the user quits the hidden mode, if there are still empty pages not used in a block, those pages should be written with randomness, and the corresponding LBAs of the public mode should be also committed back to the OOBs. 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 individual data structure which keeps track of which pages have been invalidated in the hidden mode, and this data structure is only used by the hidden mode and remains invisible to the public mode.

Garbage collection. The garbage collection usually 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 (Sec. 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, and copies out all valid data in the dirty block; 2) It erases the dirty block; 3) It stores the valid data back to the block, sequentially from the first page; the remaining empty pages should be filled with randomness; the corresponding LBA should be committed to the OOB of each page. If no dirty block can be found, all the reserved flash blocks for the hidden mode have been used. In this case, the user should be notified.

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. This is because 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 since the blocks for the hidden mode cannot be placed to the free block pool (Sec. 4.1). Upon 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 the 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 table should be invisible to the public mode. 2) The public mode typically will not swap their blocks with those storing hidden

data in the wear leveling because: the blocks storing public data will swap blocks with those in the free block pool, but the blocks storing hidden data will never enter the free block pool.

Bad block management. The bad block management *in both the public and the hidden mode* can work in the same way: When a flash block turns "bad", valid data stored on this block will be copied elsewhere, and the block will be placed to the bad block table. The FTL will simply leave a bad block there, and will not erase it. In the hidden mode, remnants in the bad block will not cause deniability compromise, since they are encrypted hidden data and, without having access to the true key, they are indistinguishable from the randomness.

Other operations. The FTL monitors I/Os from the upper layer on some reserved LBAs. If I/Os have been observed, 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 data structure which keeps track of whether a page is invalidated in the hidden mode, the erasure count table) and functions (block allocation, garbage collection, wear leveling, bad block management) specifically for the hidden mode. For the "quit" request, the FTL knows that the hidden mode ends. It will identify the blocks occupied by the hidden data but have not been completely filled, and fill those empty pages with randomness.

L_1: 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.

L_2: 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 use any type of block-based file system which is deployed on top of the hidden volume. This hidden file system acts as a "bridge" between the user and the lower 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 on the file, it will determine the request type and issue I/Os (with different I/O patterns for different requests) 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

After having installed CrossPDE, the user should manually enable the PDE. Note that for deniability purposes, CrossPDE is by default an FDE system, and the adversary is not able to find out whether the PDE is enabled or not (this is similar to any PDE software like TrueCrypt or VeraCrypt). After the PDE is enabled, CrossPDE should first perform system *initialization* for PDE. During the initialization, CrossPDE will generate a decoy key and a true key. CrossPDE will fill the entire disk (i.e., the block layer) with randomness, and create a public volume (encrypted via FDE using the decoy key) across the disk. In addition, CrossPDE will derive a secret offset based on the true key, and create a hidden volume (encrypted via FDE using the true key) starting from the secret offset towards the end of the disk. The hidden volume is hidden among the randomness filled initially. To ensure the encrypted hidden volume is indifferentiable from the randomness initially filled, the randomness is generated using the same encryption algorithm used to encrypt the hidden data. When processing non-sensitive data, the user should boot into the public mode. The user should use the mobile device in this mode in a regular manner to ensure a better plausibility [36]. When the user wants to process sensitive data, he/she should boot into the hidden mode. Upon entering the hidden mode, the hidden mode will issue a "start" request to a "special file" maintained by the hidden file system, telling the hidden file system that it is activated, and the hidden file system will then issue a "start" request downwards; similarly, when the user quits the hidden mode, it will issue a "quit" request to the "special file", telling the hidden file system that it ends. 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.

To ensure deniability, the user authentication (or *pre-boot authentication*) is necessary when entering a mode. To enter the public mode, the decoy key is required and, to enter the hidden mode, the true key is required. In practice, each corresponding key can be derived from each corresponding password using a password-based key derivation function like PBKDF2 [9]. In other words, we can derive the decoy key from the decoy password, and the true key from the true password, respectively. Each time during booting, the user is required to enter a password, and CrossPDE will derive a key from this password. It will then use the key to mount the public volume and, if the public volume can be successfully mounted [36], the corresponding password is the decoy password; otherwise, it will use the key to derive an offset, and try to mount the hidden volume at this offset and, if the hidden volume can be successfully mounted, the password is the true password; if both volumes cannot be mounted, the password is incorrect. Upon being coerced, the user can disclose the decoy password and, with the decoy password, the adversary can enter the public mode, but will not be aware of the existence of the hidden mode or the hidden volume.

5 Analysis and Discussion

5.1 Security Analysis

Security of CrossPDE is captured in Theorem 1, 2 and 3.

Theorem 1. *By running the public mode, the adversary is not able to identify the existence of PDE.*

Proof. (sketch) Using the decoy key coerced from the victim, the adversary can boot into the public mode, and can have access to all the 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, when running the public mode, the adversary may perform forensic analysis over the memory (e.g., extracting the memory content using memdump). However, each time when quitting the hidden mode, CrossPDE requires the user to shut down the device and hence the traces of the hidden mode will have been eliminated.

Theorem 2. *By analyzing the raw data on the disk (the block layer), the adversary is not able to identify the existence of PDE.*

Proof. (sketch) On the block layer, there is a public volume which occupies the entire disk, and a hidden volume which is hidden from a secret offset towards the end of the disk. Using the decoy key, the adversary is able to decrypt the public volume, obtaining the public non-sensitive data on the disk. However, without the true key, the adversary is not able to: 1) identify whether there is a hidden volume, and 2) locate the hidden volume. This is because: the entire hidden volume has been encrypted with the hidden key and, the encrypted hidden volume appears indistinguishable from the randomness filled among the entire disk initially (Note that the randomness initially filled is generated using the same encryption algorithm used to encrypt the hidden volume.); in addition, the starting offset of the hidden volume is determined by the true key and, the adversary cannot have access to the true key, and hence cannot derive this offset.

Theorem 3. *By analyzing the raw data on the flash memory, the adversary is not able to identify the existence of PDE.*

Proof. (sketch) By modifying the major functionality (block allocation, garbage collection, wear leveling, and bad block management) 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.

5.2 Discussion

Password-based pre-boot authentication. Deriving the decoy/true key from the corresponding password will essentially reduce the security provided by CrossPDE, since a memorable password implies that the adversary may become easier to guess it. A user with a high security requirement should still use keys which can be stored using NFC cards [8]. Even if the user chooses to use passwords, the passwords should be chosen following certain security guidelines [38].
About the size of the hidden volume. The size of the hidden volume is not recommended to exceed 50% of the entire disk space. In practice, the sensitive data are small in size [10] and half of the disk space should be sufficient. In the worst case if the hidden volume is filled, the user will not be able to write more data to it. A mitigation is that the user should back up some of the hidden data elsewhere, and reclaim some of the space used by the hidden volume. Also, the hidden volume technique itself cannot prevent the adversary from corrupting the hidden data, and the users are recommended to have their data backed up regularly to mitigate this attack.
Mitigating the timing attacks. Yu et al. [41] discovered a booting-time attack against PDE, which may happen when verifying 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 shortly, since the bootloader will always check the public volume first, but given a wrong key, the bootloader will return slowly, since it first checks whether it can boot the public volume and then checks the hidden volume, and finally returns with an error prompt. To obfuscate the difference, adding reasonable time delay when booting with the decoy key would help mitigate this attack [41]. In addition, the timing attack would not be effective in the public mode, as the public mode is no difference from an FDE. Also, the timing attack is not possible in the hidden mode, as the adversary cannot enter the hidden mode.
About deniability software itself. The deniability software can be disguised as regular disk encryption software. By noticing the existence of this software, the adversary cannot compromise PDE, since the victim can claim that the software is used for disk encryption. By disclosing the decoy key, the software can decrypt the public volume (i.e. the entire disk) which further convinces the adversary that it is just disk encryption software. Without knowing the true key, the adversary will not be enable to be aware of the existence of the hidden volume, and hence cannot compromise the PDE.
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 [33].
Mitigating multi-snapshot adversaries. CrossPDE can defend against a multi-snapshot adversary which can have access to the victim device multiple times, assuming that the victim is alert and will reset the device each time after being released. Specifically, after being released, the victim will back up the data, and conduct a full reset which clears both the memory and the external stor-

age (via secure deletion [21]), and then 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. This is reasonable, since a device which was caught by the adversary turns untrusted, and the victim should use it as "new"; in addition, the adversary will not be able to correlate two snapshots caught, since the reset makes them unrelated. However, if the victim is careless and does not reset the device after being released, CrossPDE will be problematic when facing the multi-snapshot adversary. A remediation is to perform dummy writes of randomness when writing public data [11] or to use the write-once memory (WOM) code [13] to encode both the public and the hidden data. However, both solutions bring significant extra overhead and may not be very suitable for mobile devices. A lightweight approach will be investigated in our future work.

About multiple-level deniability. A few PDE systems [23, 35, 41] introduce the multiple-level deniability. This allows the adversary to identify the existence of PDE at the lower security level, but cannot figure out the existence of PDE at the higher security level. CrossPDE can be easily extended to this multi-level design by recursively embedding a hidden volume of a higher security level at the end of that of a lower security level.

Strict isolation between the two modes. The public and the hidden mode share the OS and, to avoid deniability compromises, the user should disable the memory dump file generation [30] and the paging file creation [32] in the hidden mode. In addition, the system logs and temporary files of the hidden mode should be committed to the hidden volume only. Furthermore, the public and the hidden mode should not share any apps (and data). Especially, the public mode should install those apps commonly used in a mobile device, and the hidden mode should install its own necessary apps and avoid using any apps from the public mode.

Precautions against untrusted networks. When using the PDE mode, a few user practices should be adhered to avoid collusion attacks [36]: in this mode, the user should avoid using web services to prevent the collusion between the adversary and the web service providers; also, in this mode, the device should remove the SIM card and run in an airplane mode, preventing the collusion between the adversary and the wireless carriers. If a network connection is necessary in the hidden mode, anonymous tools such as Tor should be used.

Preventing deniability compromises from memory. CrossPDE requires powering off the device to quit the hidden mode, eliminating traces of hidden sensitive data in the memory. TrustZone [2] can be also used to isolate the memory of hidden mode. Both solutions may suffer from a cold boot attack [22]. But we believe the cold boot attack would not be effective in practice as it requires the adversary to capture the victim and the device a few seconds [22] after the hidden mode is quit, and to immediately have access to a cooling technique which can cool the memory chip of the device to a very low temperature.

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 [17] for the file system layer. Note that exFAT is a file system introduced by Microsoft to optimize the performance for flash memory such as USB flash drives and SD cards, and has become non-proprietary since August 28, 2019. OpenNFM was used as the FTL which manages the flash memory under both the public and the hidden mode. VeraCrypt was used to manage (e.g., create, encrypt, etc.) both the public and the hidden volume at the block layer, and to manage both the hidden and the public mode, as well as for 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 version of 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). The code (anonymously available at [1]) for CrossPDE will be open-source after the paper is published.

Modifications to OpenNFM. For the public mode, we reused the existing block allocation, garbage collection, wear leveling and bad block management of OpenNFM. For the hidden mode, we implemented the new block allocation, garbage collection, and wear leveling, but reused the existing bad block management. In addition, we modified the FTL_Read function, so that once the reserved LBAs are read, the FTL knows that there is a request from the hidden mode, and determines whether to start or to quit the hidden mode. Note that we implemented different read patterns to differentiate starting and quitting the hidden mode.

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 the developed prototype of CrossPDE to a self-built mobile device testbed, 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 [28] (ARM9 32-bit ARM926EJ-S, 180Mhz, 32MB RAM, and 512MB NAND flash) and our modified OpenNFM as the FTL. 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 [9,18,24,36,41] 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 [20]. We also compared CrossPDE with the existing FTL-based PDE systems DEFTL [25] and PEARL [13].

Initialization and pre-boot authentication. The time for initialization (e.g., creating both the public and the hidden volume) is shown in Table 1. We can observe that: 1) For both VeraCrypt and CrossPDE, public volume creation takes much longer time than that of the hidden volume. This is because, the PDE system needs to fill the entire disk with randomness when creating the public volume, which is time-consuming. 2) The initialization in CrossPDE is similar to that of VeraCrypt, since CrossPDE does not perform additional operations during the initialization compared to VeraCrypt.

The time for entering the public and the hidden mode is shown in Table 2 and 3, respectively. Note that #iteration is a value that controls the number of rounds used by the key derivation using the PBKDF2-RIPEMD160 algorithm, which is determined by Personal Iterations Multiplier (PIM) as: #iteration = 15000 + (PIM * 1000). To observe how different PIM values affect the entering time, we chose different PIM values as 100, 200, 300, resulting in different iteration numbers 115,000, 215,000, 315,000. We also assessed the default iteration number 500,000. We can observe that: 1) Entering the hidden mode requires more time than entering the public mode in both VeraCrypt and CrossPDE. This is because, a hidden volume-based PDE system always will try to mount the public volume first and, if it fails, it will continue to mount the hidden volume. In addition, a larger iteration number will result in longer time, since more iterations will be involved when running the PBKDF2 to derive the key from the password. 2) CrossPDE spends more time (9%-19%) when entering the public mode. This is because, the FTL needs to continuously monitor the reserved

	public volume	hidden volume
VeraCrypt (s)	269.16	93.01
CrossPDE (s)	269.58	93.78

Table 1. Time for initialization

#iterations	115,000	215,000	315,000	500,000
VeraCrypt (s)	6.67	6.66	7.25	7.43
CrossPDE (s)	7.50	7.93	7.90	8.45

Table 2. Time for entering the public mode

LBAs to receive requests from the upper layer. 3) CrossPDE spends slightly more (2%-6%) time on entering the hidden mode compared to VeraCrypt, due to extra operations upon mounting the hidden volume (e.g., reading the special file to communicate with the FTL).

I/O throughput. We compare the I/O throughput of CrossPDE with VeraCrypt in Table 4. We can observe that: 1) For the public mode, CrossPDE exhibits similar I/O throughput with VeraCrypt. This is because, we do not change the 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: A read operation does not create any traces hurting deniability, and there is no need for extra operations handling reads. However, a write operation will create traces leading to deniability compromise, and extra steps are needed to eliminate those traces for deniability purposes. For example, when writing a flash page in the hidden mode, the FTL will need to first read this page's OOB so that the data stored in the OOB can be maintained, causing an extra read for each single write.

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 [25]. Both results are shown in Table 5. From Table 4 and 5, 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 [25] and PEARL [13]. The comparison is shown in Table 6, in which we estimated the throughput

#iterations	115,000	215,000	315,000	500,000
VeraCrypt (s)	16.55	25.09	38.64	61.12
CrossPDE (s)	17.62	26.62	40.07	62.53

Table 3. Time for entering the hidden mode

patterns	VeraCrypt (KB/s)		MobiPDE (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 4. Throughput comparison between VeraCrypt and CrossPDE. SR - sequential read; RR - random read; SW - sequential write; RW - random write

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

Table 5. Throughput of "no encryption" and "encryption in FTL"

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 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.

Wear leveling. Effectiveness of wear leveling will determine the service lifetime of flash memory. We therefore calculate the wear leveling inequality (WLI). For an n-block flash storage medium, and each block has erase count $e_1, e_2, ...e_n$, respectively, WLI can be calculated as $\frac{1}{2}\sum_{i=1}^{n}||\frac{e_i}{E} - \frac{1}{n}||$, where $E = \sum_{i=1}^{n} e_i$. Intuitively, WLI indicates the fraction of erasures that must be re-assigned to

	PEARL [13]	DEFTL [25]	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%

Table 6. Estimation of throughput decrease in different FTL-based PDE schemes, compared to a regular system without a PDE deployed.

other blocks in order to achieve completely even wear and, a small WLI is an indication of good wear leveling. We created a public and a hidden volume, and continuously filled data to each volume, and erased the volume once it was filled. In the public volume, we repeated this process until the total amount of data written had reached 30 GB. In the hidden volume, we repeated this process until the total amount of data written had reached 10 GB. We measured the erase counts of the corresponding flash blocks, and calculated WLI values. The results are shown in Table 7. We can observe that WLI values are small [26] which indicates a good wear leveling effectiveness. In addition, the WLI value will be smaller if the wear leveling threshold is smaller, because: for a smaller threshold, wear leveling will be triggered more frequently, resulting in a more evenly distributed programmings and erasures.

wear leveling threshold	public volume (%)	hidden volume (%)
30	6.1	8.3
50	7.6	10.2

Table 7. WLI values under different wear leveling thresholds

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. Security analysis and experiments on real-world implementation show that CrossPDE can ensure deniability with a modest decrease on performance compared to the insecure VeraCrypt.

References

1. MobiPDE source code. `https://github.com/112233test/mobiPDE`.
2. TrustZone. `https://developer.arm.com/technologies/trustzone`.
3. Veracrypt. `https://www.veracrypt.fr/code/VeraCrypt/`.
4. Ross Anderson, Roger Needham, and Adi Shamir. The steganographic file system. In *International Workshop on Information Hiding*, pages 73–82. Springer, 1998.
5. Austen Barker, Staunton Sample, Yash Gupta, Anastasia McTaggart, Ethan L Miller, and Darrell DE Long. Artifice: A deniable steganographic file system. In *9th {USENIX} Workshop on Free and Open Communications on the Internet ({FOCI} 19)*, 2019.
6. 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.
7. Jan Cappaert, Bart Preneel, Bertrand Anckaert, Matias Madou, and Koen De Bosschere. Towards tamper resistant code encryption: Practice and experience. In *International Conference on Information Security Practice and Experience*, pages 86–100. Springer, 2008.
8. Bing Chang, Yao Cheng, Bo Chen, Fengwei Zhang, Wen-Tao Zhu, Yingjiu Li, and Zhan Wang. User-friendly deniable storage for mobile devices. *computers & security*, 72:163–174, 2018.
9. 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, 2015.
10. Bing Chang, Fengwei Zhang, Bo Chen, Yingjiu Li, Wen-Tao Zhu, Yangguang Tian, Zhan Wang, and Albert Ching. Mobiceal: Towards secure and practical plausibly deniable encryption on mobile devices. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 454–465. IEEE, 2018.
11. B. Chen. Towards designing a secure plausibly deniable system for mobile devices against multi-snapshot adversaries–a preliminary design. *arXiv preprint arXiv:2002.02379*, 2020.
12. Chen Chen, Anrin Chakraborti, and Radu Sion. Infuse: Invisible plausibly-deniable file system for nand flash. *Proceedings on Privacy Enhancing Technologies*, 4:239–254, 2020.
13. Chen Chen, Anrin Chakraborti, and Radu Sion. Pearl: Plausibly deniable flash translation layer using wom coding. In *The 30th Usenix Security Symposium*, 2021.
14. Niusen Chen, Bo Chen, and Weisong Shi. The block-based mobile pde systems are not secure – experimental attacks. *arXiv preprint arXiv:2203.16349*, 2022.
15. Google Code. Opennfm. `https://code.google.com/p/opennfm/`, 2011.
16. dm-crypt. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-crypt.html.
17. exfat file system specification. https://docs.microsoft.com/en-us/windows/win32/fileio/exfat-specification.
18. Wendi Feng, Chuanchang Liu, Zehua Guo, Thar Baker, Gang Wang, Meng Wang, Bo Cheng, and Junliang Chen. Mobigyges: A mobile hidden volume for preventing data loss, improving storage utilization, and avoiding device reboot. *Future Generation Computer Systems*, 2020.
19. Typical hardware of flash storage devices. `https://github.com/112233test/Table/blob/main/table2.md`.

20. Freecode. fio. http://freecode.com/projects/fio, 2014.
21. 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.
22. J Alex Halderman, Seth D Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A Calandrino, Ariel J Feldman, Jacob Appelbaum, and Edward W Felten. Lest we remember: cold-boot attacks on encryption keys. *Communications of the ACM*, 52(5):91–98, 2009.
23. Jin Han, Meng Pan, Debin Gao, and HweeHwa Pang. A multi-user steganographic file system on untrusted shared storage. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 317–326, 2010.
24. Shuangxi Hong, Chuanchang Liu, Bingfei Ren, Yuze Huang, and Junliang Chen. Personal privacy protection framework based on hidden technology for smartphones. *IEEE Access*, 5:6515–6526, 2017.
25. Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. Deftl: Implementing plausibly deniable encryption in flash translation layer. In *Proceedings of the 24th ACM conference on Computer and communications security*. ACM, 2017.
26. Shijie Jia, Luning Xia, Bo Chen, and Peng Liu. Deftl: Implementing plausibly deniable encryption in flash translation layer. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2217–2229, 2017.
27. Neil F Johnson and Sushil Jajodia. Steganalysis: The investigation of hidden information. In *1998 IEEE Information Technology Conference, Information Environment for the Future (Cat. No. 98EX228)*, pages 113–116. IEEE, 1998.
28. Mantech. Lpc-h3131. http://www.mantech.co.za/, 2017.
29. Andrew D McDonald and Markus G Kuhn. Stegfs: A steganographic file system for linux. In *Information Hiding*, pages 463–477. Springer, 2000.
30. Memory dump files. `https://www.veracrypt.fr/en/Memory%20Dump%20Files.html`.
31. Microsof. Bitlocker. `https://technet.microsoft.com/en-us/library/hh831713.aspx`, 2013.
32. Paging file. https://www.veracrypt.fr/en/Paging%20File.html.
33. Plausible deniability. `https://www.veracrypt.fr/en/Plausible\%20Deniability.html`.
34. HweeHwa Pang, K-L Tan, and Xuan Zhou. Stegfs: A steganographic file system. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*, pages 657–667. IEEE, 2003.
35. Timothy M Peters, Mark A Gondree, and Zachary NJ Peterson. Defy: A deniable, encrypted file system for log-structured storage. 2015.
36. 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.
37. Adam Skillen and Mohammad Mannan. Mobiflage: Deniable storage encryption for mobile devices. *IEEE Transactions on Dependable and Secure Computing*, 11(3):224–237, 2014.
38. How to create a strong password (and remember it). https://www.howtogeek.com/195430/how-to-create-a-strong-password-and-remember-it/.
39. TrueCrypt. Free open source on-the-fly disk encryption software.version 7.1a. *Project website: `http://www.truecrypt.org/`*, 2012.
40. Gregory Wroblewski. General method of program code obfuscation. 2002.

41. 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 *International conference on information security*, pages 555–567. Springer, 2014.