6-5-2018

# Ensuring data confidentiality via plausibly deniable encryption and secure deletion – a survey

Qionglu Zhang
*Chinese Academy of Sciences*

Shijie Jia
*Chinese Academy of Sciences*

Bing Chang

Bo Chen
*Michigan Technological University*, bchen1@mtu.edu

## Recommended Citation

Zhang, Q., Jia, S., Chang, B., & Chen, B. (2018). Ensuring data confidentiality via plausibly deniable encryption and secure deletion – a survey. *Cybersecurity, 1*(1). http://dx.doi.org/10.1186/s42400-018-0005-8
Retrieved from: https://digitalcommons.mtu.edu/michigantech-p/836

Follow this and additional works at: https://digitalcommons.mtu.edu/michigantech-p
Part of the Information Security Commons

Cybersecurity

SURVEY Open Access

# Ensuring data confidentiality via plausibly deniable encryption and secure deletion – a survey

Qionglu Zhang[1,2,3], Shijie Jia[1,2*], Bing Chang[4] and Bo Chen[5*]

**Abstract**

Ensuring confidentiality of sensitive data is of paramount importance, since data leakage may not only endanger data owners' privacy, but also ruin reputation of businesses as well as violate various regulations like HIPPA and Sarbanes-Oxley Act. To provide confidentiality guarantee, the data should be protected when they are preserved in the personal computing devices (i.e., *confidentiality during their lifetime*); and also, they should be rendered irrecoverable after they are removed from the devices (i.e., *confidentiality after their lifetime*). Encryption and secure deletion are used to ensure data confidentiality during and after their lifetime, respectively.

This work aims to perform a thorough literature review on the techniques being used to protect confidentiality of the data in personal computing devices, including both encryption and secure deletion. Especially for encryption, we mainly focus on the novel plausibly deniable encryption (PDE), which can ensure data confidentiality against both a coercive (i.e., the attacker can coerce the data owner for the decryption key) and a non-coercive attacker.

**Keywords:** Data confidentiality, Plausibly deniable encryption, Secure deletion

## Introduction

Modern computing devices (e.g., desktops, laptops, smart phones, tablets, wearable devices) are increasingly used to process sensitive or even mission critical data. Protecting confidentiality of those sensitive data is of paramount importance because: First, data leakage will endanger data owners' privacy. For example, the data leakage of iCloud in 2014 disclosed almost 500 private pictures of various celebrities (Cbsnews: Apple's celebrity icloud leak probably has mundane causes 2014). Second, it will ruin reputation of businesses. For example, Equifax data breach in July 2017 caused a leak of 145,500,000 consumer records; a few local governments like cities of Chicago and San Francisco, as well as the Commonwealth of Massachusetts, have filed enforcement actions against Equifax (Ballard Spahr LLP: State and local governments move swiftly to sue equifax 2017); Third, it will directly violate regulations like HIPAA (Congress 1996), Gramm-Leach-Bliley Act (Congress 1999), and Sarbanes-Oxley Act (Sarbanes and Oxley 2002). The data confidentiality should be ensured not only during their lifetime (i.e., the data are preserved in the devices), but also after their lifetime (i.e., the data have been removed from the devices). This is because, by recovering sensitive data which have been deleted, the attacker can achieve a similar gain compared to successfully attacking the confidentiality of the data being preserved in the devices. For example, by recovering a naked picture deleted by a victim, the adversary can still use it to embarrass the victim or ask the victim for ransom money.

Correspondingly, the research efforts for protecting data confidentiality can be divided into two categories: encryption and secure deletion. Encryption can protect confidentiality of the data stored at rest by transforming them into another format using some secrets (e.g., keys), such that the adversary is not able to correlate the transformed format to the original format without obtaining the secrets. All types of existing encryption mechanisms like symmetric encryption and asymmetric

*Correspondence: jiashijie@iie.ac.cn; bchen@mtu.edu
[1] Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China
[2] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[5] Department of Computer Science, Michigan Technological University, Houghton, USA
Full list of author information is available at the end of the article

encryption can achieve the aforementioned security property. Secure deletion is to ensure that once the sensitive data are deleted, the probability of recovering them is negligibly small. This requires special techniques to completely destroy data, eliminating any traces which may lead to a full/partial data recovery.

To protect confidentiality of the data stored in a computing device, conventional encryption may not work when both the computing device and the device's owner are captured by an attacker, since the attacker can coerce the owner to disclose the secret (i.e., a *coercive adversary*). Once the secret is disclosed, the transformed format created by encryption will be reversed, and the sensitive data will be leaked. A novel encryption technique, plausibly deniable encryption (PDE) (Canetti et al. 1997), has been designed to complement the traditional encryption to handle such a coercive attack. The high-level idea of PDE is, the original sensitive message is encrypted into ciphertexts in a special way, such that during decryption, if a true key is used, the original sensitive message can be recovered, but if a decoy key is used, a plausible message will be generated. Therefore, upon being coerced, the owner can simply disclose the decoy key to protect confidentiality of the original sensitive message.

To protect confidentiality of the data deleted from a computing device, the deleted data should be made completely unrecoverable. Conventionally, this is ensured by carefully over-writing the storage medium storing the data using garbage information (Joukov and Zadok 2005; Wei et al. 2011; Garfinkel and Shelat 2003; Sun et al. 2008; Gutmann 1996) or deploying encryption using ephemeral keys (Perlman 2005a, b, Geambasu et al. 2009, Tang et al. 2012, Reardon et al. 2012, Zarras et al. 2016). This unfortunately was shown to be insufficient, since past existence of the deleted data will create impacts on both the data organization (Bajaj and Sion 2013b) and the other data which have not been deleted (Bajaj and Sion 2013a). Those impacts can then be utilized by the adversary as an oracle to derive sensitive information about the deleted data. In the worst case, the adversary is able to completely recover the data being deleted (Chen et al. 2016). Therefore, recent secure deletion approaches focus on eliminating those impacts (Bajaj and Sion 2013a, b, Chen and Sion 2015; Chen and Sion 2016; Jia et al. 2016).

In this work, we aim to conduct a thorough literature review on data confidentiality protection. We believe that data confidentiality should be always ensured no matter the data are preserved in the computing device or have been removed. Therefore, our survey covers techniques being used to protect data confidentiality for both cases: 1) data are stored in the devices; 2) data have been removed from the devices. For the first case, we mainly focus on PDE which can provide confidentiality guarantee against both the coercive adversaries and the non-coercive adversaries. We summarize the research for PDE theory and systems (including both the desktop systems and the mobile systems). For the second case, we summarize secure deletion approaches in various storage media including hard disk drives (HDDs) and NAND flash memory. We also outline the new direction of secure deletion approaches by eliminating impacts of operation history.

**Organization.** In "Background" section, we introduce the background knowledge of flash memory, the architecture of a storage system, PDE as well as secure deletion. In "Models and assumptions" section, we unify the adversarial model for both PDE and secure deletion, and also summarize the assumptions required by PDE. We then summarize the literature for PDE in "Protecting data confidentiality against coercive adversaries via PDE" section and for secure deletion in "Ensuring confidentiality of the deleted data via secure deletion" section, respectively. In "Future directions" section, we discuss a few future directions of PDE and secure deletion. We conclude in "Conclusion" section.

## Background

### Flash memory

Flash memory is a solid-state non-volatile computer storage medium that can be electrically erased and reprogrammed. It can eliminate mechanical limitations and latency of hard drives, achieving a much higher I/O throughput with a much lower power consumption. Therefore, it gains popularity in main-stream mobile devices like smart phones, tablets, wearable devices. In addition, a lot of high-end laptops like Apple MacBook use flash memory as external storage. Even cloud providers allow their users to choose solid state drives (SSDs) as the underlying storage media (Amazon: New SSD-Backed Elastic Block Storage 2016). The flash memory being used as the storage medium is mainly NAND flash. NAND flash stores data using an array of memory cells, which are grouped into pages (each page can store 512-byte, 2KB, or 4KB data), and multiple pages are further grouped into blocks (can contain 32, 64, or 128 pages).

Compared to traditional mechanical disks, NAND flash has several unique characteristics. First, NAND flash has an erase-before-write design. Specifically, to overwrite a flash page, the page needs to be erased before any new data can be programmed to it. Second, the unit of read/program of NAND flash is a page, but the unit of erase is a block, which consists of multiple pages. The first and the second special characteristics of NAND flash make it expensive to perform an in-place update. Therefore, flash memory usually prefers an out-of-place update. Third, each flash block has a finite number of program-erase (P/E) cycles. In other words, a flash block will be worn out if the number of programs/erasures performed over it exceeds a certain threshold. Last, NAND flash is vulnerable to read/program disturb (Cai et al. 2015). In other words, frequently reading/writing the same flash location may corrupt the data stored nearby. Due to read/program disturb (Cai et al. 2015), NAND flash usually requires computing an error-correcting code (ECC) for each page, and storing the parity in the "spare area" of the corresponding page. In general, the raw flash can be managed through either flash translation layer (FTL) or flash-specific file system.

– **Flash translation layer**. To be compatible with traditional block-based file systems (e.g., EXT4), flash memory can be emulated as a block device by exposing a block-based access interface, which is the most popular form of flash-based products (e.g., SSDs, eMMCs, USB sticks). This is usually achieved by introducing special firmware, flash translation layer (FTL), between the file system and the raw flash. FTL can translate the logical block addresses to the underlying physical flash addresses, providing a block access interface to upper layers.
– **Flash file systems**. Another alternative of using raw flash is to directly build a flash-specific file system over it. A flash file system is a file system optimized specifically for flash memory. Popular flash file systems include YAFFS (Robust Flash Storage: YAFFS 2002), UBIFS (Memory Technology Devices: UBIFS 2015), JFFS2 (Sourceware: Jffs2 2003), and F2FS (Lee et al. 2015). Note that flash file systems become less popular nowadays. Most of the recent mobile devices are only designed to be compatible with FTL-based flash devices, and usually do not allow directly accessing the raw flash. For example, the Google Nexus 6P Android phone uses eMMC cards as storage media, and only the old Android phones like Nexus One and Nexus S allow directly accessing the raw flash.

Special internal management (being incorporating into FTL or flash file systems) is usually required to handle the characteristics of flash memory, which may easily lead to deniability compromise or data leakage, making it challenging to provide deniability/secure deletion on flash-based storage systems.

### The architecture of a storage system

The storage media like magnetic HDDs and NAND flash are usually managed through a storage system, which is organized into a few layers. The layers interact with each other, and provide users a unified interface to access the data being stored in the storage media. Figure 1 shows a typical architecture for storage systems.

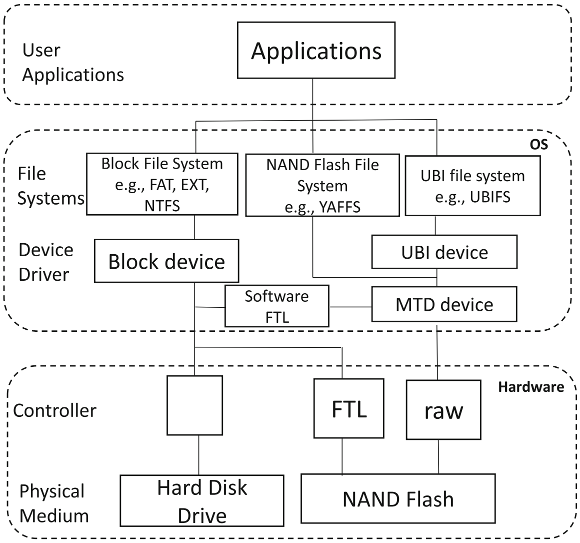


**Fig. 1** The architecture of storage systems

The lowest layer is the physical medium layer, where data are actually stored, e.g., HDDs or NAND flash. The physical storage medium is always accessed through a controller. The basic function of the controller is to translate data format on the physical storage medium (e.g., electrical voltage) into another format (e.g., binary values) understandable by upper layers. The controller offers a standardized and well-defined hardware interface, e.g., ATA (Team work systems: Advanced technology attachment 2017) and SCSI (Incits: Scsi storage interfaces 2016), which allows data to be read from/written to the physical storage medium. The HDD adopts in-place updates, and hence its controller usually consistently maps a logical block address to a certain storage location on the physical storage medium. On the contrary, NAND flash prefers out-of-place updates due to its special features, and is usually managed through FTL or a flash specific file system ("Flash memory" section). Device drivers are to consolidate access to different types of hardware by exposing a common simple interface in the form of software. The block device driver interface allows reading and writing of blocks in logical addresses. The block device driver can be used on top of an HDD control or NAND flash being encapsulated by FTL. The memory technology device (MTD), another type of device driver, is used to access raw NAND flash memory directly. MTD permits reading and writing, but blocks must be erased before being written, which occurs at a large granularity. Unsorted block images (UBI) is another interface for accessing flash memory, which builds on top of MTD interface and simplifies some aspects of using raw flash memory.

File systems are responsible for organizing logical sequences of data among the available blocks on the physical storage medium through the interface provided by the device driver. These include: 1) block file systems built on top of block device, e.g., FAT32, EXT4, and NTFS; 2) flash file systems built on top of MTD device, e.g., YAFFS (Robust Flash Storage: YAFFS 2002); 3) UBI file system (Memory Technology Devices: UBIFS 2015) built on top of UBI device.

The highest layer is the application layer, offering an interface to the users.

### Plausibly deniable encryption (PDE)

Conventional encryption is broadly used by individuals and businesses to protect sensitive data. Major operating systems now increasingly support the use of full disk encryption. For example, FileVault (Apple: Use filevault to encrypt the startup disk on your mac 2017) in Mac OS X 10.3 and later, BitLocker (Microsoft: Bitlocker 2017) in Windows Vista and later, full-disk encryption in Android 4.4 and later, etc. However, traditional encryption may not work when data owners are captured by the adversary and coerced into disclosing their decryption keys. To protect sensitive data against this type of coercive adversaries, plausibly deniable encryption (Canetti et al. 1997; Dürmuth and Freeman 2011; Howlader and Basu 2009; Ibrahim 2009; Klonowski et al. 2008; Meng and Wang 2009; 2010; O'Neill et al. 2011) (PDE) can be utilized to hide the sensitive data by denying their very existence. Different from conventional encryption, PDE encrypts original sensitive message into ciphertexts in such a way that, upon decryption, if a true key is used, the original sensitive message can be recovered, but if a decoy key is used, a plausible message will be generated. When being coerced, the victim can simply disclose the decoy key. Using the decoy key, the adversary is able to decrypt the ciphertext into the plausible message which is non-sensitive, and is hence convinced that no sensitive information is stored.

### Secure deletion

Secure deletion is a technique designed to ensure complete elimination of sensitive data once they become obsolete. It requires a guarantee that *an adversary should neither recover the deleted data, nor learn anything about them*. However, achieving such a guarantee in modern computing systems is a challenging task due to the complication of the storage systems ("The architecture of a storage system" section). A modern storage system usually consists of multiple layers, and performing secure deletion in one layer is usually not able to eliminate the data, since data leakage may be observed in other layers. For example, given a Microsoft Word document, removing data from the document itself cannot guarantee that the deleted data really become inaccessible. Upon receiving a delete request issued by Microsoft Word (which belongs to the application layer), the system may simply modify the metadata in the file system layer (e.g., changing the block allocation table and invalidating the data being deleted) to make the data appear to have been removed in the application layer. However, the actual content is still preserved in the physical storage medium layer and may be recovered by the adversary through disk forensics (Breeuwsma et al. 2007; Garfinkel and Shelat 2003). Therefore, secure deletion requires ensuring that the content being deleted should become inaccessible at each layer of the storage system.

In addition, the past existence of the deleted data may leave artifacts in the data organization (Bajaj and Sion 2013b) or side effects on the other data (Bajaj and Sion 2013a). After the data have been deleted, those artifacts or side effects may be utilized by the adversary to learn sensitive information of the deleted data (Chen et al. 2016). This also creates a barrier towards completely removing sensitive information.

## Models and assumptions

### A unified adversarial model for PDE and secure deletion

We unify the adversarial model for both PDE and secure deletion by considering a snapshot adversary who can have access to the state of a victim device. The adversary is assumed to be not able to control the code of the victim system, i.e., no malicious code can be injected and hence the code of the victim system is secure. Each access will allow the adversary to obtain a full snapshot of both the external storage and the memory. The adversary is computationally bounded, and tries to illegitimately derive sensitive information from the snapshots being captured. We consider that the adversary can have access to the victim device once (i.e., a *single-snapshot adversary*) and multiple times (i.e., a *multi-snapshot adversary*).

The single-snapshot adversary captures a lot of real-world scenarios. For example, an attacker steals a smart phone (Yu et al. 2014; Chang et al. 2015) or a laptop, or breaks into a data center obtaining a snapshot of a victim server. The multi-snapshot adversary also captures a lot of real-world scenarios. For example, an attacker periodically breaks into a hotel room, obtaining a "memory dump" of a victim's smart phone; a border checker periodically obtains snapshots from a victim's smart device (Blass et al. 2014; Peters et al. 2015).

The only unique attack behavior for PDE is the adversary may coerce the data owner for the decryption keys. This behavior is not applicable to secure deletion, because no key for the deleted data will be preserved after the data have been securely removed.

### The assumptions required by PDE

Since PDE systems usually require a few common assumptions. We summarize these assumptions in the following.

- The PDE software should be merged into the code stream of the device (e.g., part of the Android framework), such that its availability is widespread, and an attacker cannot simply compromise deniability based on the availability of software support. In addition, PDE requires changing a few system components (e.g., booting process). The adversary who can perform reverse engineering or dynamic analysis over those components will unavoidably compromise deniability. The PDE systems cannot defend against this type of adversary.
- The adversary will know the design of PDE. However, he/she does not have any knowledge on secret information (e.g., keys and passwords) being required to open/operate the PDE mode.
- The adversary is rational and will stop coercing the device's owner once he/she is convinced that the decryption keys have been revealed.
- The adversary can not capture a device working in the PDE mode or after a crash of the PDE mode. Otherwise, he/she can trivially retrieve sensitive hidden data, compromising deniability.
- The operating system, bootloader, baseband OS, firmware are all malware-free. In addition, the PDE mode is malware-free.

## Protecting data confidentiality against coercive adversaries via PDE

Ensuring confidentiality of the data being preserved in personal computing devices can be achieved by encryption. However, traditional encryption cannot defend against coercive adversaries ("A unified adversarial model for PDE and secure deletion" section). Therefore, we mainly focus on plausibly deniable encryption (PDE), which can protect confidentiality of the data present in the computing devices against both coercive and non-coercive adversaries.

### PDE – from Theory to Practice

An ideal PDE would be a special encryption, which can encrypt sensitive plaintexts into ciphertexts, such that the ciphertexts can be decrypted into either original sensitive plaintexts (using true key) or plausible non-sensitive plaintexts (using decoy key). This is to ensure that one key can be disclosed when the data owner is coerced. However, such an ideal encryption is impractical for storage systems because: First, the existing instantiation for PDE results in a growing size of ciphertexts (Canetti et al. 1997), which itself could be an indication of the existence of deniability. Second, a modern storage system is usually complicate, consisting of multiple layers, e.g., application layer, file system layer, physical storage medium layer ("The architecture of a storage system" section). Simply encrypting the data using PDE in the application layer cannot ensure that traces of the sensitive data will not be observed by the snapshot adversary in the underlying file system and physical storage medium layer, especially when the sensitive data need to be updated over time and the adversary can obtain multiple snapshots ("A unified adversarial model for PDE and secure deletion" section). Therefore, when being applied to storage systems, rather than simply use encryption, two types of PDE techniques, steganography and hidden volumes, are used to provide deniability.

The first type of PDE technique is *steganography* (Anderson et al. 1998). The basic idea of steganography is to hide sensitive data within regular file data. For example, the sensitive data can be computed by performing an XOR operation over a few cover files (Anderson et al. 1998). A main concern of the steganography technique is to avoid over-writing the hidden sensitive data, since they are actually part of the regular data. This can be mitigated by creating and storing (secretly) multiple copies of the sensitive data, which in return will lead to inefficient use of disk space.

The other type of PDE technique is *hidden volumes* (TrueCrypt: Free open source on-the-fly disk encryption software.version 7.1a 2012). The hidden volumes technique works as follows: There are two encrypted volumes on the disk, a public volume and a hidden volume. The public volume is encrypted using a decoy key and the hidden volume is encrypted using a hidden key (i.e., the true key). The public volume is placed on the entire disk and the hidden volume is usually placed from a secret offset towards the end of the disk (i.e., the hidden volume is part of the public volume, see Fig. 2). Note that initially the entire disk is filled with random data and the data written to the public volume should be placed sequentially from the beginning of the disk to reduce probability of overwriting the hidden volume. When the victim is coerced into revealing the encryption key, he/she can disclose the decoy key, and the attacker will use the decoy key to decrypt the entire disk and cannot distinguish the hidden volume from the random noise being filled initially, and is thus convinced that no sensitive data are stored. The hidden volumes technique can be viewed as a special type of steganography technique, which always hides the sensitive data in a contiguous region being placed at the end of the disk and remains undetected by the coercive adversaries.

Leveraging hidden volumes and steganography, various PDE systems have been built to enable deniability on computing devices. All those works can be divided into two categories, PDE for desktop computers and PDE for mobile devices.

### PDE for desktop computers

The existing PDE systems for desktop computers mainly rely on steganography and hidden volumes. In a few

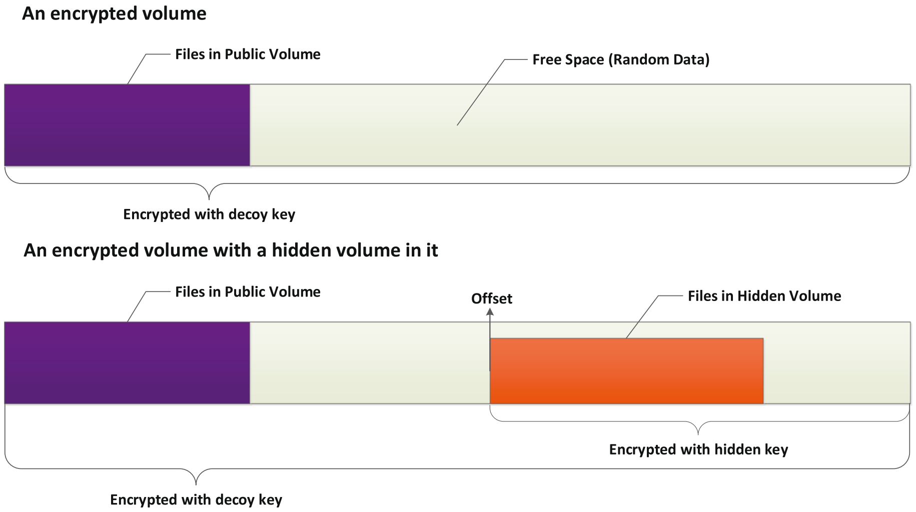


**Fig. 2** The hidden volumes technique

recent works, researchers also incorporate Oblivious RAM (ORAM) in order to defend against multi-snapshot adversaries.

**PDE using steganography.** Steganographic file systems have been initially proposed by Anderson et al. (1998), with two alternative schemes. In their first scheme, the system has a number of cover files originally consisting of random bits and the user's files can be computed using a subset of cover files. This scheme uses a password and the file name to determine which cover files are used. Its drawback is requiring storing a large number of cover files. Moreover, to accommodate files of arbitrary length, the cover files must be relatively large. Their second scheme is based on the computational security of block ciphers. The system is initialized by filling the entire hard disk with random data. A sensitive file is encrypted and stored at disk location being derived from the file name and the password, and the encryption key is also derived in a similar manner. In this way, the adversary is not be able to distinguish blocks containing hidden data from free blocks being filled with random noise. However, when the disk is increasingly filled by hidden files, collision of disk locations may increase, leading to significant over-writes. This is mitigated by writing each block to several disk locations.

Inspired by the second scheme of Anderson et al. (1998), McDonald et al. proposed StegFS (McDonald and Kuhn 2000), an EXT2-based file system which uses an external block allocation table to record entries for disk blocks. StegFS supports a few security levels, each with a separate password. To prevent overwriting the data from a security level which is closed, StegFS writes data in a redundant manner. When multiple security levels are open, since hidden and regular files are present in the same file system, data leakage may occur. A user of StegFS cannot deny the existence of hidden files, due to existence of the modified EXT2 driver and the external block table. However, the user can keep the number of security levels secret.

To further improve efficiency and reliability, Pang et al. designed another steganographic file system (Pang et al. 2003; 2004). They use a bitmap to mark the blocks being used by the hidden files, and thus eliminate the need of storing multiple copies of a file, alleviating the reliability issues and I/O inefficiencies. However, the adversary may be able to identify existence of hidden files, because the hidden blocks, being marked as used, do not possess a directory record. Three approaches are used to mitigate the aforementioned compromise. First, a few blocks, which do not store hidden data, are abandoned and marked as being used during initialization. Second, when the system creates a new file, several additional blocks are allocated and filled with random noise. Third, to prevent adversaries from identifying whether a block stores hidden data, the system maintains a few dummy hidden files and periodically updates them in the background. These mitigation approaches however, increase overhead of disk space usage.

Zhou et al. further improved Pang et al.'s work by adding dummy transactions to obfuscate hidden files in cloud storage (Zhou et al. 2004). Although the reliability and I/O efficiency have been improved, disk space overhead remains large. Troncoso et al. (2007) presented traffic analysis attacks on the file update algorithm proposed by Zhou et al. (2004). Their attacks can detect file updates and reveal existence as well as location of files. Specifically, they can detect files which occupy multiple data blocks with only two updates. Moreover, their attacks can also reveal files which occupy only one data block after a sufficient number of access operations. Han et al. proposed a dummy-relocatable steganographic (DRSteg) file system (Han et al. 2010) to provide deniability in a multi-user environment. By sharing dummy data among multiple users in the system, DRSteg is able to increase the level of deniability being provided to individual users. In DRSteg, in order to free dummy data without destroying user data, a novel dummy relocation mechanism is used to allow individual users to distinguish dummy data from other users' data. It can also prevent adversaries from distinguishing dummy and user data even after obtaining multiple snapshots. There are also a few StegFS-based implementations including RubberhoseFS (Assange et al. 2012) and Magikfs (Varun et al. 2017), which are no longer maintained and the existing implementations may not be compatible with modern Linux operating system.

**PDE using hidden volumes.** Disk encryption tools like TrueCrypt (TrueCrypt: Free open source on-the-fly disk encryption software.version 7.1a 2012) and FreeOTFE (Sourceforge: FreeOTFE A free "on-the-fly" transparent disk encryption program for PC and PDAs 2017) use hidden volumes to provide plausible deniability. TrueCrypt supports user data encryption with several ciphers including AES, TwoFish, Serpent, and a cascade of these ciphers in the XTS mode. The header of each TrueCrypt volume is filled with random data (e.g., salt) or encrypted with the header key which is derived from the corresponding password using PBKDF2. Therefore, the entire volume appears as being filled with randomness. When TrueCrypt loads a volume, since it does not store the cipher specification, all supported ciphers will be tried with a header key (being derived from the user's password using PBKDF2) until it decrypts the volume and obtains the ASCII string "TRUE" from a certain block in the header. Then TrueCrypt decrypts the encrypted master volume key using the header key. Note that the master volume key is generated randomly upon creating the volume. If a TrueCrypt hidden volume is created, there will be also a hidden header, which contains offset of the hidden volume. The hidden header is tested before the public header when mounting a volume. TrueCrypt can also create a hidden OS in a hidden volume by creating a new partition and copying the current OS to the hidden volume. When the system is booted into the hidden OS, all unencrypted volumes and non-hidden encrypted volumes are mounted in a read-only manner, ensuring that any OS/application-specific leakage stays within the hidden volume.

Czeskis et al. (2008) analyzed TrueCrypt and proposed three types of attacks against it. They consider three types of leakage sources: 1) the operating system; and 2) the primary applications (i.e., an application that is used to manage hidden data); and 3) the non-primary applications. Modern operating systems are not designed to preserve deniability and may perform many unexpected behaviors. As a result, even when the operating system runs properly, information relevant to hidden data may be leaked. For example, some operating systems (e.g., Windows) automatically create shortcuts to hidden files when they are used, and those shortcuts may be stored in regular non-hidden directories. The shortcuts may contain information about the hidden file, e.g., file name, location, length, access time, creation time and even volume serial number of the file system on which the hidden file is stored. If the adversary localizes those shortcuts, it may suspect existence of the hidden data, compromising deniability. Another possible leakage source is the primary application. The primary applications are not necessarily designed to preserve deniability, and may leak information about the hidden data. For example, primary applications may create redundant files to prevent data loss. If those files are not properly deleted, the content of the hidden data may be leaked. Finally, non-primary applications, such as desktop search applications, may access the files being stored in the hidden volume. Those applications may cache snapshots of the hidden files and store them for a later use. The adversary may also compromise deniability from those cached data.

**Other PDE systems for desktop computers.** If the adversary can capture multiple snapshots of hidden volume at different points of time, they can detect existence of the hidden volume, by simply comparing different snapshots and identifying whether "free" blocks have been changed. Therefore, TrueCrypt cannot provide deniability when facing a multi-snapshot adversary. Blass et al. proposed HIVE (Blass et al. 2014) to allow the user to deny existence of the hidden volume when facing a multi-snapshot attack. This is achieved by hiding every access of the disk using Oblivious RAM (ORAM) (Goldreich and Ostrovsky 1996; Stefanov et al. 2013), which was originally designed to hide patterns of access to remote storage. However, ORAM is known as expensive in terms of both computation and I/O. Although HIVE uses a less expensive write-only ORAM that only supports write operations, its performance is still far from being practical.

To improve performance of HIVE, Chakraborti et al. proposed DataLair (Chakraborti et al. 2016; 2017). Having observed that revealing access patterns to the public data is unnecessary, DataLair only protects operations on the hidden data and ensures that they are indistinguishable from operations on the public data. In addition, DataLair optimizes the oblivious access mechanism being deployed for hidden data. Compared to HIVE, DataLair is two orders of magnitude faster in terms of public data access, and five times faster in terms of hidden data access.

Roche et al. designed DetWoORAM (Roche et al. 2017), an entirely new technique for write-only ORAM, which uses a deterministic and sequential writing pattern, eliminating the need of any "stashing" of blocks in local state. In DetWoORAM, since the write will always succeed and occur in a free block, the notion of stash can be completely removed. They also pointed out that the construction of DataLair does not satisfy write-only obliviousness, since the process of finding free blocks leaks information about which blocks are free, and the adversary can tell whether recent writes have been performed on the same address or not.

Zuck et al. presented Ever-Changing Disk (ECD) (Zuck et al. 2017) to achieve deniable storage system. Their design follows three requirements: 1) resistance to multi-snapshot adversaries; and 2) ensuring that hidden data will not be destroyed when a user is writing the public volume; and 3) using normal system operations on public data to disguise writes to hidden data. In ECD, the storage space is separated into two parts: a part containing the public data volume and the other containing the hidden data volume. The hidden volume is visible to the system when the user enters the secret key. ECD uses a large volume of pseudorandom data to hide the sensitive data. A portion of data from the volume are periodically migrated using normal firmware operations to obfuscate writes to the hidden data. Since hidden and pseudorandom data blocks are constantly relocated and modified, the hidden data may eventually be overwritten without knowing the secret key. To mitigate the overwrite issue, the rate of internal data migration is controlled by the user and the user should enter the secret key periodically.

Zhao et al. proposed Gracewipe (Zhao and Mannan 2015), by which the victim can provably destroy/erase data when being coerced, hoping that a reasonable adversary will find no reason to keep holding him/her. Gracewipe works as follows: During setup, the user selects three passwords, which can be used to derive the key for encrypting the corresponding volume: 1) hidden password that only derives the hidden volume key; and 2) decoy password that derives only the decoy volume key; and 3) deletion password that derives the decoy volume key and overwrites the hidden volume key. When coerced, the victim can fake compliance, and enter the deletion password, and then can prove to the attacker that Gracewipe has been executed and the real key is no longer available.

Table 1 summarizes the existing PDE systems for desktop computers, being incorporated into different layers of the storage systems. To defend against snapshot adversaries, they may rely on hidden volumes, steganography, or ORAM, and provide one or multiple deniability levels with different overheads. Schemes based on steganography (e.g., McDonald and Kuhn 2000; Pang et al. 2003) suffer from high performance overhead since they write multiple copies of files or dummy writes for steganography. Hidden volume mechanism (TrueCrypt: Free open source on-the-fly disk encryption software.version 7.1a 2012) introduces low overhead since the hidden files are stored in the free space and the overwrite problem is mitigated by linear space allocation. Since ORAM is known to be expensive, schemes based on ORAM (e.g., Blass et al. 2014; Chakraborti et al. 2017; Roche et al. 2017) unavoidably have high overheads.

## PDE for mobile devices

Compared to desktop computers, mobile devices are usually different in two aspects: First, they are equipped with less computational power. Second, they usually use flash memory ("Flash memory" section) as storage. The existing PDE systems for mobile devices can be divided into two categories: 1) The PDE systems built on top of block devices. This type of PDE systems views flash memory as a black box, which exposes a block-access interface through FTL ("The architecture of a storage system" section). 2) The PDE systems built on top of flash memory. This type of PDE systems directly work on top of flash memory to provide deniability while handling the special nature of flash.

**Table 1** Summary of PDE systems for desktop computers

| Scheme | Method | Adversary | Deniable level | Layer |
|---|---|---|---|---|
| Scheme 1 (Anderson et al. 1998) | Steganography | Multi-snapshot | Arbitrary | File system |
| Scheme 2 (Anderson et al. 1998) | Steganography | Single-snapshot | Arbitrary | File system |
| StegFS (McDonald and Kuhn 2000) | Steganography | Single-snapshot | Arbitrary | File system |
| StegFS (Pang et al. 2003; 2004) | Steganography | Multi-snapshot | Arbitrary | File system |
| StegHide (Zhou et al. 2004) | Steganography | Multi-snapshot | Arbitrary | File system |
| TrueCrypt (2012) | Hidden volume | Single-snapshot | One | Block device |
| HIVE (Blass et al. 2014) | ORAM | Multi-snapshot | Multiple | Block device |
| DataLair (Chakraborti et al. 2017) | ORAM | Multi-snapshot | Multiple | Block device |
| DetWoORAM (Roche et al. 2017) | ORAM | Multi-snapshot | Multiple | Block device |
| ECD (Zuck et al. 2017) | Hidden volume | Multi-snapshot | One | Block device |

**PDE systems on top of block devices.** Skillen et al. (Skillen and Mannan 2013, 2014; Skillen 2013) designed Mobiflage, the first PDE scheme for mobile devices. They provided two versions of implementations: one in external storage for FAT32 file system (Skillen and Mannan 2013), and the other in internal storage for modified EXT4 file system (Skillen and Mannan 2014). The main contribution of Mobiflage lies in its first incorporation of hidden volumes technique to the Android devices. It works as follows: First, it fills the external storage with random data. Second, it creates two volumes, a public volume for storing public non-sensitive data, applications and settings, and a hidden volume for storing sensitive data. Correspondingly, there are two operation modes, a public mode and a PDE mode. The public mode is used to manage the public volume for daily use, providing storage encryption without deniability. In this mode, the data are encrypted with a decoy key, which is derived from a decoy password. The user will be asked to provide the decoy password during the booting in order to enter the public mode. The PDE mode is used to manage the hidden volume which stores sensitive data whose existence needs to be denied when being coerced. In this mode, the data are encrypted with a true key, which is derived from a true password. The user should provide the true password during system boot to activate the PDE mode. The exact location of the hidden volume is derived from the true password.

The FAT32 version of Mobiflage (Skillen and Mannan 2013) is specially designed for external storage formatted using FAT32 file system, which requires the support of a physical or an emulated FAT32 SD card. This is because, the hidden volume is part of the public volume, and is placed at the end of the disk. In other words, if an EXT-like file system is deployed for the public volume, the data stored in the hidden volume may be easily overwritten by the public data due to the nature of EXT file system's random allocation. To eliminate the aforementioned assumption, the EXT4 version of Mobiflage (Skillen and Mannan 2014) modifies the EXT4 driver such that it can support a sequential inode allocator and can be deployed for the public volume to avoid overwriting the hidden volume. However, the modification of the EXT4 driver itself may be an indication of the existence of PDE.

Yu et al. proposed MobiHydra (Yu et al. 2014) to improved Mobiflage in three aspects: 1) It can mitigate a novel booting-time attack being faced by Mobiflage; 2) It can support multiple levels of deniability. 3) It supports mode switching without rebooting. To mitigate the booting-time attack, MobiHydra obfuscates the time required for a wrong password (i.e., an arbitrary password except the true and the decoy password) during booting such that the adversary is not able to identify the existence of PDE by simply entering a wrong password. To support fast mode switching, MobiHydra introduces a special partition called shelter volume on the external storage, which is used as a temporary storage partition which can temporarily store the sensitive data being created in the public mode, without the need of entering the PDE mode for storing hidden sensitive data. The data stored in the shelter volume will be immediately synchronized to the hidden volume when the hidden mode is entered, and are then eliminated from the shelter volume. To avoid deniability compromise, the sensitive data stored in the shelter volume will be encrypted by a random key which is encrypted by the public key and stored in the shelter volume. In addition, some dummy files are maintained in the shelter volume and updated periodically to obfuscate the writes of hidden sensitive data to the shelter volume. MobiHydra, however, cannot eliminate the assumption of requiring a physical or an emulated FAT32 SD card.

To eliminate the limitations of Mobiflage and MobiHydra, Chang et al. designed MobiPluto (Chang et al. 2015), a file system friendly PDE design. The basic idea of MobiPluto is introducing an additional software layer between the PDE and the file system. This software layer should satisfy three requirements: 1) Its existence should not be an indication of PDE; 2) It should provide virtual volumes to file systems, and any block-based file systems can be deployed on a virtual volume; 3) It should convert non-sequential allocation from a file system to sequential allocation in the underlying PDE. To build such a layer, MobiPluto uses thin provisioning (Thornber), because: First, thin provisioning has been implemented by dm-thin-pool module, which has been a well-established tool in Linux kernel; Second, thin provisioning can allow to create thin volumes, each of which can be used to deploy any block-based file systems. Third, thin provisioning can transform non-sequential allocation on the thin volume to sequential allocation on the underlying storage. By combining thin provisioning and hidden volumes, MobiPluto is able to achieve a "file system friendly" PDE design.

Chang et al. further improved the usability of MobiPluto in their extended work (Chang et al. 2018) by introducing a fast switching mechanism and using NFC cards to store strong PDE passwords. For PDE systems on mobile devices, fast switching is a desired feature. When a device owner faces an emergency and wants to collect sensitive information, he/she needs to instantly switch the device to the hidden mode. However, it needs more than 1 min for the prior mobile PDE systems to switch modes, because a full device rebooting is usually required. Fast switching mechanism eliminates the need for rebooting the device and the switching time is reduced to less than 10 s (Chang et al. 2018). Their idea is to restart only the Android framework rather than the entire device, significantly reducing the switching time. For the hidden volume, a strong password is required to protect sensitive data. However, users tend to choose weak passwords, thus security may not be ensured. To address this issue, they proposed to use NFC cards to store strong passwords for the user. Their observation is that most of the modern mobile devices are equipped with NFC features.

MobiCeal (Chang et al. 2018) designed the first block-based PDE system for mobile devices which can defend against a multi-snapshot adversary. The fundamental idea of MobiCeal is to use dummy writes to obfuscate writes performed in hidden mode. In this way, even though the adversary can have access to multiple snapshots of the block image, it cannot tell whether a write is issued by the hidden mode or not. In addition, MobiCeal can support multiple deniability levels. Finally, it identifies various side channel leakage present in prior PDE schemes for mobile device and eliminates them.

**PDE systems on top of flash memory.** Flash memory has significantly different nature compared to mechanical disks, e.g., flash memory is update unfriendly and vulnerable to wear ("Flash memory" section). All the aforementioned block-based PDEs unfortunately may suffer from deniability compromises in the underlying flash storage due to the handling of the unique nature of flash memory. This is because: The unique characteristics of flash memory require a special internal management, which creates a different view of data in flash memory, independent of the view on the block layer. By having access to the raw flash, the adversary can obtain this different view, which may allow it to observe those unexpected "traces" of sensitive data, whose existence needs to be denied. To eliminate the aforementioned deniability compromise, a few PDE systems directly incorporate PDE into flash memory.

Peters et al. (2015) introduced DEFY, a deniable encrypted file system based on flash file system YAFFS2 (Robust Flash Storage: YAFFS 2002). In DEFY, operations on a higher security level are indistinguishable from the operations on a lower security level. In addition, DEFY can mitigate over-writes of hidden data in the higher security level by taking advantage of special properties offered by a log-structured file system. DEFY however, suffers from a few limitations. First, it strongly relies on system properties in YAFFS2 to provide deniability. Therefore, it is incompatible with the flash-based block devices using FTL, the most popular form of flash storage being used in mobile devices nowadays. Second, it suffers from deniability compromises (Jia et al. 2017). This is because, to prevent data at lower security levels from overwriting the data at higher security levels, it disables garbage collection at the lower security levels. The adversary can easily identify this abnormal behavior and suspect existence of PDE (Jia et al. 2017).

DEFTL (Jia et al. 2017) incorporated PDE into FTL. DEFTL also has two modes, a public and a hidden mode. The deniability of DEFTL is achieved by using the data (and their behavior) in the public mode to deny the data (and their behavior) in the hidden mode. Most importantly, to prevent the data written in the public mode from over-writing the data written in the hidden mode, DEFTL carefully modifies the block allocation and garbage collection strategies in the FTL such that the two modes can be "stealthily" isolated without being known by the adversary. Specifically, the public volume will allocate flash blocks from the head of the block pool and the hidden volume will allocate flash blocks from the tail of the pool. In addition, garbage collection in the two modes will be modified as: In the public mode, garbage collection will be performed actively to fill the head of the pool; in the hidden mode, garbage collection will be performed actively to fill the tail of the pool. This can avoid that the public mode has used all the blocks in the head and starts to use the blocks in the tail, over-writing the hidden sensitive data. DEFTL also provides a few attacks on the existing PDE systems for mobile devices.

Table 2 summarizes the existing PDE systems for mobile devices. They either use hidden volumes or steganography, and provide one or multiple deniability levels when facing single-snapshot or multi-snapshot adversaries. Schemes based on hidden volumes (Skillen and Mannan 2013, 2014; Yu et al. 2014; Chang et al. 2015, 2018) usually have low performance overhead, but they can only defend against single-snapshot adversaries. DEFY (Peters et al. 2015) relies on steganography to defend against multi-snapshot adversaries, but the performance

**Table 2** Comparison of PDE systems for mobile devices

| Scheme | Method | Adversary | Deniability level | Layer |
|---|---|---|---|---|
| Mobiflage-FAT32 (Skillen and Mannan 2013) | Hidden volume | Single-snapshot | One | Block device |
| Mobiflage-EXT4 (Skillen and Mannan 2014) | Hidden volume | Single-snapshot | One | Block device |
| MobiHydra (Yu et al. 2014) | Hidden volume | Single-snapshot | Multiple | Block device |
| MobiPluto Chang et al. (2015, 2018) | Hidden volume | Single-snapshot | Multiple | Block device |
| MobiCeal (Chang et al. 2018) | Steganography | Multi-snapshot | Multiple | Block device |
| DEFY (Peters et al. 2015) | Steganography | Multi-snapshot | Arbitrary | File system |
| DEFTL (Jia et al. 2017) | Hidden volume | Single-snapshot | One | Controller (FTL) |

overhead is high. MobiCeal (Chang et al. 2018) uses a lightweight "dummy write" mechanism to defend against multi-snapshot with acceptable performance overhead.

### Discussions

**About randomness being used in hidden volumes technique.** The idea of hidden volumes technique is to hide the encrypted hidden volume among the randomness being filled initially. To make sure that the ciphertexts being generated are indistinguishable from random bits, the random bits can be drawn from the same distribution as the ciphertext space by using the encryption function itself as the PRNG (Skillen and Mannan 2014). This can ensure that the ciphertexts and the randomness are from the same source. In addition, the encryption being used in the hidden volumes technique is full disk encryption (FDE), which treats each disk sector as an autonomous unit and assigns sector-specific IVs for chaining modes such as CBC and XTS (Skillen and Mannan 2014). This can help eliminate the correlations among ciphertexts due to similarities in files (e.g., file heads).

**Transferring data between desktop computers and mobile devices.** Since PDE can be deployed in either desktop computers or mobile devices, there is a possibility that the data owner would like to exchange data (e.g., files) between these two different platforms. The data exchange should be conducted in such a way that deniability should not be compromised. When the file is non-sensitive, it can be simply transferred from a desktop computer to a mobile device (or vice versa) without any deniability concern. This can be done by simply using Internet or direct copying via cable, when both the source device and the destination device are working in the public mode. When the file is sensitive whose existence needs to be denied, to transfer it from a desktop computer to a mobile device (or vice versa), there are two known options: 1) If the file is transferred via Internet, to allow the data owner to deny the transferring of this file (the attacker can collude with Internet ISPs and identify this event), covert communications approaches (Frèche et al. 2017; Hu et al. 2017) can be used. 2) If the file is transferred via direct copying, the file can be read from the source device and written to the destination device, when both devices are working the hidden mode. Since the attacker cannot capture the device working in the hidden mode ("The assumptions required by PDE" section), the deniability will not be compromised. Once the file is successfully transferred, it will be protected by the PDE system in the destination device.

## Ensuring confidentiality of the deleted data via secure deletion

PDE is used to ensure confidentiality of sensitive data which are preserved in the personal computing devices. However, once sensitive data are discarded, the data owner may want to permanently remove them. Protection of data confidentiality requires securely disposing those data to prevent the adversary from recovering all or portion of them. This is achieved by using secure deletion.

Hard disk drives (HDDs) and NAND flash memory dominate the storage media of personal computing devices. However, they are completely different in nature, and hence different secure deletion approaches should be used to eliminate data from them. In the following, we summarize secure deletion approaches for HDDs-based and flash memory-based storage systems, respectively. We also summarize the recent works which complement conventional secure deletion approaches by taking care of the impacts created by past existence of the deleted data.

### Secure deletion for HDDs-based storage systems

An HDD is a magnetic medium which supports in-place updates. Therefore, in HDDs-based storage systems, when a file block is updated (or deleted), its old version can be simply replaced by its new version (or random noise) in the storage medium to achieve secure deletion.

*Physical medium layer.* Due to lack of semantics of file system, single-file sanitization is not feasible at the physical medium layer. To delete the data, a naivest way is to overwrite/destroy all the data on the physical medium. Tools such as degaussers can be used to sanitize data on HDDs (Kissel et al. 2006).

*Controller layer.* At the controller layer, there are several standardized interfaces that permit reading/writing of fixed-sized blocks. Similarly, there are no semantics of file system in this layer, thus the controller must sanitize every block to achieve secure deletion. To delete the data, sanitize commands and overwrite techniques are widely used in HDDs, e.g., secure erase commands offered by both SCSI (Incits: Scsi storage interfaces 2016) and ATA (Team work systems: Advanced technology attachment 2017). These sanitization commands work like a button that erases all data on the device by exhaustively overwriting every block with zeros or ones.

*Block device layer.* Reardon et al. (2013) proposed a secure deletion approach targeting persistent storage. Their approach relies on encryption and key wrapping. They use a key disclosure graph to model the adversarial knowledge about key generation and wrapping history. In addition, a small securely-deleting key-value map is used to discard encryption key of the data, achieving secure deletion.

*File system layer.* When deleting files in the EXT2, there is a sensitive attribute for files and directories to indicate that secure deletion should be used. (Bauer and Priyantha 2001) provided a patch that implements this attribute. By marking a block as free, the patch passes the free block to a kernel daemon, which maintains a list of blocks that must be sanitized. If the free block stores data from a sensitive file, instead of returning it to the file system as an empty block, this free block will be added to the work queue. When the system is idle, the sanitization daemon runs asynchronously to perform sanitization over the work queue, allowing the user to achieve immediate file deletion.

(Joukov and Zadok 2005) proposed a file system extension, purgefs, which uses block-based overwriting when blocks are returned to the file system's free blocks list. It supports overwriting file data and metadata for all files or just files marked as sensitive. (Joukov and Zadok 2006) also proposed three secure deletion approaches for EXT3. The first approach is called EXT3 basic, which securely deletes data (but not metadata) once by overwriting it. The second approach is called EXT3 comprehensive, which overwrites file data and metadata by a configurable overwriting scheme. Both the aforementioned two overwriting-based approaches can securely delete all the data or just files whose extended attributes include a sensitive flag. The third approach is based on intercepting files deletion events, i.e., unlinking and truncating a file. The file to be deleted is moved into a special secure deletion directory, and a background user-level tool shred (Plumb 2010) will delete the files in the secure deletion directory at regular intervals.

Peterson et al. (2005) optimized secure deletion for versioning file systems using an all-or-nothing transform (AONT). Using AONT, each data block is extended into an encrypted data block along with a small stub. If any part of the ciphertext is deleted, the entire message can not be decrypted any more. In this way, a specific version of a file can be quickly deleted by simply overwriting all the stubs. In addition, to delete a large log file to which data have been appended only, securely deleting all the blocks in its most recent version will achieve secure deletion on all its past versions.

*Application layer.* The application layer can only interact with file system through a POSIX-compliant interface. A user-level application can securely erase all the data on the storage medium by invoking a secure erase command (Incits: Scsi storage interfaces 2016; Team work systems: Advanced technology attachment 2017) in the hardware controller' interface. A few files overwriting tools, e.g, srm (Jagdmann 2015) and wipe (Durak 2006), can be used to securely remove files. Gracewipe (Zhao and Mannan 2015), as has been discussed in "PDE for desktop computers" section, can achieve secure and verifiable deletion of encryption keys through a special deletion password by taking advantage of TPM and Intel TXT, thus making the encrypted data permanently inaccessible. In the case of database, there is also a secure deletion interface, which can be used to overwrite data with zeros in the underlying file system. For MySQL, (Stahlberg et al. 2007) proposed an approach to delete entries by overwriting them with zeros, and the transaction log is encrypted and can be securely disposed by deleting the encryption key. For SQLite (SQLite: Pragma statements 2017), there is a compile-time option to enable a secure deletion feature that overwrites deleted records with zeros.

Table 3 summarizes the existing secure deletion approaches for HDDs-based storage systems. Those approaches are incorporated into different layers of a storage system and rely on either overwriting or encryption for secure deletion. They may also have different deletion granularity. Generally, the encryption-based secure deletion approaches for HDDs-based storage systems (Reardon et al. 2013; Peterson et al. 2005; Zhao and Mannan 2015) have relatively low performance overheads since only keys need to be deleted to achieve secure deletion.

### Secure deletion for flash memory-based storage systems

The aforementioned secure deletion approaches for HDDs-based storage systems rely on properties of hard disks: magnetism-based and supporting in-place updates. However, flash memory does not possess these properties. Wei et al. (2011) performed a series of experiments to show that the deletion techniques that work well for HDDs may not work properly for NAND flash. As NAND flash is not magnetism-based, the degaussing method may damage NAND flash chips and render data unreadable, but all the data may still remain intact (Wei et al. 2011). Wei et al. (2011) created 1000 small files on an SSD, then dismantled the drive, and searched for the content of those files. They found that the SSD contained up to 16 stale copies of the tested files. This is because the FTL creates redundant file copies during garbage collection and out-of-place updates, which unfortunately will complicate the secure deletion design for flash memory. They also tested 13 single-file overwriting-based sanitization tools (LSoft Technologies Inc: Active@ KillDisk 2017; GEEP EDS LLC: Darik's Boot and Nuke 2017), to find out whether they work correctly for flash memory. Unfortunately, all those tools were not able to sanitize data from flash memory: between 4 and 75% of the files' content remained in the SATA SSDs, and between 0.57 and 84.9% of the data remained in USB drives. All the aforementioned results indicate that securely deleting data from NAND flash is much more challenging compared to HDDs, due to the special nature of NAND flash. In the following, we summarize the existing works aiming to securely remove data from flash-based storage systems, which are divided into two categories: overwriting-based and encryption-based.

**Table 3** Comparison of features offered by different secure deletion approaches for HDDs-based storage systems

| Scheme | Method | Layer | Deletion granularity |
|---|---|---|---|
| Degaussers (Kissel et al. 2006) | Degaussing | Physical medium | The entire storage medium |
| Secure erase commands (2016, 2017) | Overwriting-based | Controller | The entire storage medium |
| Reardon et al. (Reardon et al. 2013) | Encryption-based | Block device | Single file |
| Bauer et al. (Bauer and Priyantha 2001) | Overwriting-based | File system (EXT2) | Single file |
| Purgefs (Joukov and Zadok 2005) | Overwriting-based | File system (EXT2) | Single file |
| Joukov et al. (Joukov et al. 2006) | Overwriting-based | File system (EXT3) | Single file |
| Peterson et al. (Peterson et al. 2005) | Encryption-based | File system | Single file |
| File overwriting tools (Jagdmann 2015; Durak 2006) | Overwriting-based | Application | Single file |
| Gracewipe (Zhao and Mannan 2015) | Encryption-based | Application | The entire storage medium |
| Stahlberg et al. (Stahlberg et al. 2007) and SQLite (SQLite: Pragma statements 2017) | Based on overwriting and encryption | Application (for database) | Single entry/record |

***Overwriting-based secure deletion for NAND flash.***
A common idea for the overwriting-based secure deletion approaches is to replace the deleted data with meaningless information, e.g., noisy random data.

*Physical medium layer.* Similar to the physical medium layer in the HDDs-based storage systems ("Secure deletion for HDDs-based storage systems" section), data can not be deleted in this layer due to lack of semantics of the file system. To realize secure deletion in this layer, the entire flash chip should be destroyed.

*Controller layer.* A main type of flash controller is using FTL to handle the special nature of NAND flash and to provide a block access interface to upper layers ("The architecture of a storage system" section). To securely delete data, the simplest way is to erase the corresponding flash blocks in the FTL (i.e., block erasure). However, erasures can only be performed in terms of flash blocks ("Flash memory" section). This will be overkill if only a portion of data being stored in a flash block needs to be deleted. Considering content of a file may be distributed in different pages of different flash blocks, sanitizing a file using block erasure will be unavoidably expensive.

Wei et al. proposed scrubbing (Wei et al. 2011) to address the aforementioned issue. As programming individual pages is possible, the idea of scrubbing is to reprogram the page, where the data should be securely deleted, to turn all its remaining '1' bits into '0'. Note that flash memory allows to individually program bit '1' to '0', but the reverse operation is not feasible except performing a block erasure. A major concern of scrubbing is that it may result in undefined behaviors due to possibility of introducing read errors. To handle this concern, Wei et al. examined error rates for different types of flash memory and showed that the error rates vary widely. For some flash devices, scrubbing causes frequent errors, while for some others, it does not cause any errors. They introduced scrub budget, which refers to the number of times that the NAND flash can allow to be scrubbed without exhibiting a significant risk of data errors. When the scrub budget for a block is exceeded, secure deletion will be instead performed by other approaches (e.g., invoking garbage collection). More recently, Qin et al. (2013) incorporated RAID-5 architecture to enhance the reliability and eliminate the negative effect of reprogram on flash memory.

*File system layer.* Sun et al. (2008) proposed a secure deletion method in YAFFS by investigating characteristics of NAND flash memory. They proposed two secure deletion approaches, zero overwrite (similar to the scrubbing (Wei et al. 2011)) and block erase. Especially, they define a costs-benefits model by comparing the overwrite cost on the deleted pages and the erase cost on the block that contains the deleted pages. Additionally, a new adaptive hybrid scheme is applied to select the cheaper one between the two secure deletion approaches. Another kernel-level zero-overwriting secure deletion approach was also proposed by Reardon et al (2012).

By adding a new communication channel between the file system and the device driver, the file system can inform the device that particular blocks are no longer valid, e.g., Trim (Intel Corporation: Intel Solid-State Drive Optimizer 2009) command and TrueErase (Diesburg et al. 2012). With the information of invalid blocks, the device driver can implement its own efficient secure deletion without requiring data blocks to be explicitly overwritten by the file system. Especially, TrueErase is designed for the blocks belonging to files specifically marked as sensitive.

*Application layer.* Since the application layer cannot directly touch the lower layers, secure deletion can only be achieved by filling the entire remaining free space. Reardon et al. (2012) proposed two user-level filling-based secure deletion approaches for YAFFS: purging and ballooning. Their basic idea is to fill the entire free space of the file system, such that all unused blocks of the physical medium will no longer contain sensitive information. By completely filling the file system's empty space with noise, all previously data deleted by the user are guaranteed to have been erased. Compared to purging which ensures rapid secure deletion of data from user-space, ballooning achieves a probabilistic continuous secure deletion guarantee by reducing the block reallocation period. Braga et al. (2014) proposed two user-level approaches to securely delete files on Android smart phones. The one that is designed to delete unencrypted files is also based on filling. The cost of the filling-based secure deletion approaches is proportional to the size of free space available on the physical medium. A larger size of free space will lead to a higher overhead in order to fill it (Reardon et al. 2013). The efficiency can be improved by perpetually maintaining the free space of the physical medium within a limited range (Reardon et al. 2012).

#### *Encryption-based secure deletion for NAND flash.*

Data can be rendered inaccessible by encrypting them and deleting the corresponding key. Boneh et al. proposed the first encryption-based solution that securely deletes encrypted data stored on the tape by deleting the cryptographic keys (Boneh and Lipton 1996).

*Controller layer.* Reliably destroying keys is challenging, as side-channel attacks based on semiconductor memory data remnants (Halderman et al. 2009) may allow an attacker to recover the key or key-related information. Swanson et al. (2010) proposed scramble and finally erase (SAFE), which combines encryption and erasure techniques to provide almost instant secure deletion with verifiability. SAFE relies on the assumptions that data in the SSDs are stored encrypted and the SSDs implement best practices of key management (e.g., the keys should never leave the controller). It works as follows: Upon receiving a sanitize command, it erases the controller's key memory, such that the driver is not able to encrypt/decrypt the data. It then erases every block on the device, and writes all the pages with a known pattern, and erases them again. Finally, it reinitializes the device and performs a low-level format operation on the drive, and provides a new key to the controller.

*File system layer.* Lee et al. (2008, 2010) proposed a secure deletion approach for YAFFS, a log structured NAND flash file system. By modifying YAFFS, they encrypt files and force all keys of a specific file to be stored in the same block. Therefore, only one erase operation needs to be performed in order to securely delete a file. Lee et al. (2011) then extended the aforementioned approach to perform standard data sanitization which can satisfy government agencies' requirements (NSA/CSS and DoD 5220.22-M) for the secure deletion.

Reardon et al. (2012) proposed data node encrypted file system (DNEFS), which enables secure data deletion for flash memory. They also incorporate DNEFS into flash file system UBIFS (Memory Technology Devices: UBIFS 2015). In DNEFS, they divide the entire flash memory into two areas: a small key storage area and a large main data storage area. They encrypt each data node (i.e., the unit of I/O) with a unique key, and collocate the keys in the key area. Secure deletion is achieved by removing keys, which can be performed efficiently, as keys are condensed in a small area.

DEFY (Peters et al. 2015) also provides secure deletion, complementary to its deniability. It leverages all-or-nothing transform (AONT), a cryptographic function which can ensure that a missing portion of a message will render the entire message irrecoverable. In this way, DEFY can efficiently achieve secure deletion by only removing a small portion of the data being deleted. Braga et al. (Braga and Colito 2014) proposed two secure deletion approaches for Android phones, one of which is based on encryption. They modified the key management of EncFS (Wang et al. 2012), an encrypted file system, to ensure that every file is encrypted with a unique key and a random IV. The removal of the unique key and IV makes the file irrecoverable.

Table 4 summarizes the features offered by different secure deletion approaches for flash-based storage systems, including overwriting-based and encryption-based approaches. These approaches may also have different deletion granularity. Generally, the scrubbing-based secure deletion approaches (Wei et al. 2011; Qin et al. 2013; Sun et al. 2008; Reardon et al. 2012), which only targets at the invalid data, and the encryption-based secure deletion approaches (Lee et al. 2008, 2010, 2011; Reardon et al. 2012), which only need to delete keys, have relatively low performance overheads compared to the overwriting-based approaches. A major advantage of encryption-based approaches is that, deleting a small key usually can be much more efficiently achieved compared to deleting data which are large in size.

### A new direction for secure deletion

Secure deletion is used to securely dispose data once they become obsolete. This requires a security deletion guarantee that the adversary should neither recover the deleted data, nor learn anything about them ("Secure deletion" section). The question is, can we achieve the secure deletion guarantee by simply deleting the data themselves? The answer is unfortunately no. Conventional secure deletion approaches rely on either overwriting or encryption to make the deleted data inaccessible. However, the past

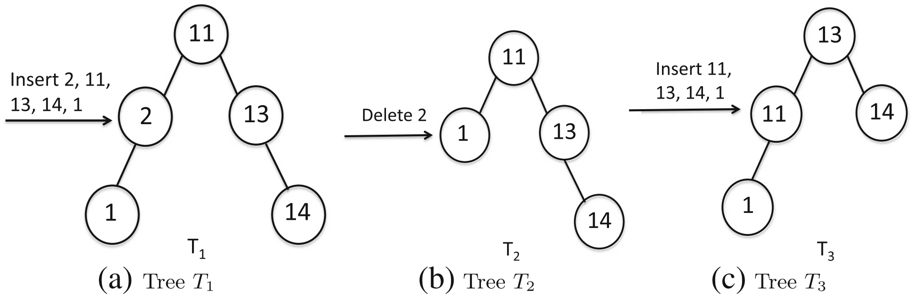
**Table 4** Comparison of features offered by secure deletion approaches for flash-based storage systems

| Scheme | Method | Layer | Deletion granularity |
|---|---|---|---|
| Scrubbing (Wei et al. 2011) and SmSD (Qin et al. 2013) | Overwriting-based | Controller (FTL) | A physical page |
| Sun et al. (Sun et al. 2008) | Overwriting-based | File system (YAFFS) | Single file |
| Zero overwriting (Reardon et al. 2012) | Overwriting-based | File system (YAFFS) | Single file |
| Trim (Intel Corporation: Intel Solid-State Drive Optimizer 2009) and TrueErase (Diesburg et al. 2012) | Overwriting-based | File system to device driver | Single file |
| Purging and ballooning (Reardon et al. 2012) | Overwriting-based | Application | All the invalid data |
| Braga et al. (Braga and Colito 2014) for unencrypted files | Overwriting-based | Application | All the invalid data |
| SAFE (Swanson and Wei 2010) | Based on overwriting and encryption | Controller | The entire storage medium |
| Lee et al. (2008, 2010) | Encryption-based | File system (YAFFS) | Single file |
| Lee et al. (2011) | Encryption-based | File system (YAFFS) | Single file |
| DNEFS (Reardon et al. 2012) | Encryption-based | File system (UBIFS) | Single file |
| DEFY (Peters et al. 2015) | Encryption-based | File system (YAFFS) | Single file |
| Braga and Colito (2014) for encrypted files | Encryption-based | File system (EncFS) | Single file |

existence of the deleted data may leave artifacts in the layout at all layers of a computing system (Bajaj and Sion 2013b; Chen and Sion 2015) or create side effects on the other data which have not been deleted (Bajaj and Sion 2013a), and the adversary can potentially take advantage of those structural artifacts or side effects to learn sensitive information about the deleted data (Bajaj and Sion 2013a, b; Chen and Sion 2015).

To justify the impact of the past existence of the deleted data, we use a balanced binary search tree (BST) (Chen et al. 2016). We first create a balanced BST by inserting five nodes in the order of 2, 11, 13, 14, 1, and obtain tree $T_1$ (Fig. 3a). We then delete node 2, obtaining tree $T_2$ (Fig. 3b). However, if we directly create the balanced BST by inserting nodes in the order of 11, 13, 14, 1, we will obtain tree $T_3$ (Fig. 3c). This example indicates that, due to the past existence of node 2, $T_2$ and $T_3$ have different structures. This also indicates that, although node 2 have been deleted, its structural artifacts remain in the data organization. Therefore by having access to tree $T_2$, the adversary may suspect the past existence of the sensitive data which have been deleted, and tries to partially or fully recover them.

We also describe some concrete attack scenarios which can take advantage of structural artifacts (Jia et al. 2016; Chen et al. 2016). Commodity NAND flash-based block devices usually adopt a log-structured writing technique, in which flash blocks as well as pages within a block are allocated sequentially (Min et al. 2012). As shown in Fig. 4, A, B, C, and D are the data being written to NAND flash, and each occupies a flash page. To securely delete C, the user has two options: 1) We perform a scrubbing (Wei et al. 2011) over the corresponding flash page. The scrubbing technique however, will convert this page to a



(a) Tree $T_1$ (b) Tree $T_2$ (c) Tree $T_3$

**Fig. 3** An example from (Chen et al. 2016) showing why structural artifacts matter. $T_1$, $T_2$, and $T_3$ are balanced BSTs. **a** Tree $T_1$, **b** Tree $T_2$, and **c** Tree $T_3$

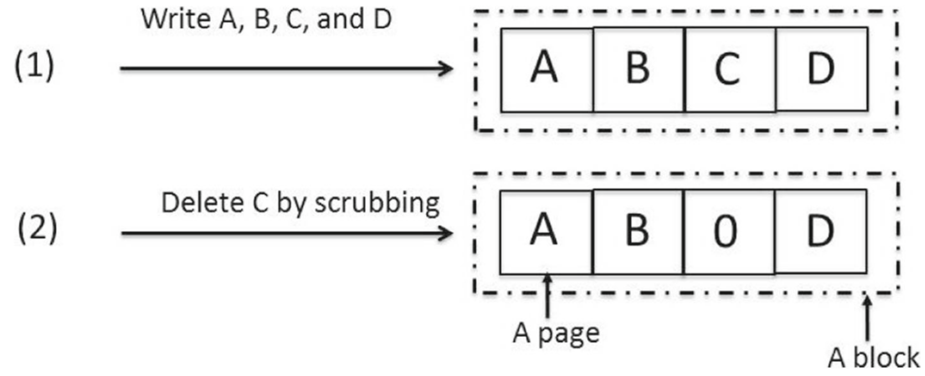


**Fig. 4** Scrubbing-based secure deletion

page with all "0" bits (i.e., a zero page). By having access to the storage state after deletion of C, the adversary will notice the zero page and suspect that a past deletion has been performed on it. 2) We encrypt A, B, C, and D with different keys, and delete key for C (Reardon et al. 2012). However, the adversary can also find out that C has been deleted since it cannot be successfully decrypted to plaintexts which are semantically meaningful. A further attack will be performed to fully or partially recover C by taking advantage of the correlation between C and its neighboring data (Chen et al. (2016) provides a concrete attack scenario which can take advantage of the structural artifacts to recover bitcoin transactions).

To remove impacts of the past existence of the deleted data (including both structural artifacts and side effects), a few recent works investigate undetected secure deletion, truly secure deletion, history independence, as well as untraceable deletion.

**Undetectable secure deletion.** Jia et al. (2016) tried to achieve the secure deletion guarantee by hiding the deletion history. Intuitively, if the deletion history is concealed from the adversary, he/she should not be able to find out whether there was a deletion in the past, eliminating his/her possibility in recovering the deleted data or learning anything about the deleted data. Based on this key observation, Jia et al. investigated a novel security notion for NAND flash-based block devices, i.e., undetectable secure deletion, to achieve two security properties: 1) Data to be deleted are completely removed from NAND flash memory, which ensures that the adversary cannot have access to the data once they have been deleted; 2) The deletion history is concealed from the adversary, which ensures that the adversary cannot gain any knowledge about whether there was a deletion in the past.

To realize undetectable secure deletion, Jia et al. proposed NAND Flash Partial Scrubbing (NFPS), an undetectable secure deletion scheme for NAND flash-based block devices. Having observed that deleting data from flash with a full scrubbing (Wei et al. 2011) or a block erasure may provide the adversary a clue that there was a deletion in the past, Jia et al. proposed to perform a partial scrubbing, i.e., partial page reprogramming and partial block erasure, to only modify a portion of the bits in the page/block storing the deleted data, avoiding producing a zero page or an all-"1" block.

**Truly secure deletion.** NFPS (Jia et al. 2016) aimed to conceal the past existence of the deleted data in the NAND flash memory. However, it still cannot sanitize the structural artifacts introduced by the deleted data. Chen et al. (2016) investigated another novel security notion, namely, truly secure deletion, which can ensure the sanitization of both the data and the structural artifacts.

To achieve truly secure deletion, Chen et al. (2016) proposed TedFlash, a truly secure deletion scheme for NAND flash-based block devices. In TedFlash, the data of every write to NAND flash will be placed to an empty location which is randomly selected. Note that as the random placement of data is independent and does not affect the placements of any other data, TedFlash can eliminate the structural artifacts brought by each write. Most importantly, the random placement technique is exclusively feasible for NAND flash, because: 1) Random seeks on flash memory are as efficient as sequential seeks. 2) The random placements can distribute data evenly among flash, naturally achieving a good wear leveling.

**History independence.** History independence is proposed to prevent historic information about the pattern of access to a data structure from being leaked through its representation when observed by an adversary (Chen and Sion 2015). History independence ensures that by having access to a storage state, the adversary is not able to identify the operation sequence which leads to this state. In other words, given two operation sequences leading to the same storage state: one sequence has a delete operation (e.g., delete *D*) and its corresponding insert operation (e.g., insert *D*), and the other sequence does not have the aforementioned delete and insert operation, the adversary will not be able to differentiate which operation sequence led to this storage state. Therefore, history independence ensures that after having removed a data record, the storage state is somehow equivalent to a state that the deleted record never exists, naturally achieving the secure deletion guarantee. Implicitly, history independence ensures that no structural artifacts will be introduced. Otherwise, the memory representation of each storage state cannot be "independent" of the operation sequence leading to it.

Bajaj and Sion (2016; Bajaj and Sion 2013b) designed history independent file system (HIFS), the first approach which can provide history independence for file storage over mechanical hard disks. HIFS uses a history independent hash table to allocate file data to the underlying block device in a history independent manner. Most significantly, they can simultaneously achieve history independence and preserve locality to improve I/O performance. However, HIFS does not work for flash-based storage systems ("The architecture of a storage system" section), since flash memory usually has its own internal software layer (e.g., FTL) which is used to transparently handle its special characteristics by using a special history dependent block placement technique.

Chen et al. thus proposed HiFlash (Chen and Sion 2015), aiming to achieve history independence in flash-based block devices. To achieve history independence, HiFlash enforces a bijection between block device and flash memory. Specifically, HiFlash always places data records, which are written to the same block-device locations, to the same flash locations, regardless of their write patterns. However, by introducing a bijective mapping between block device and flash memory, HiFlash can only remove structural artifacts introduced by the software component staying between the block device and the flash memory (e.g., FTL). It is unfortunately not able to remove the structural artifacts introduced from the upper layers. Therefore, it strongly relies on the assumption that the upper layer has eliminated the structural artifacts, which is not necessarily true.

**Untraceable deletion.** Bajaj and Sion (2013a) introduced untraceable secure deletion, aiming to remove side effects of the data being deleted (i.e., the impacts of the past existence of the deleted data on the data which are still preserved in the computing devices). The corresponding design, Ficklebase, is to achieve untraceable secure deletion in the context of relational databases. In Ficklebase, once a tuple is expired, all its side-effects will be removed via versioning and query rewriting.

Table 5 summarizes the secure deletion approaches which handle impacts of the past existence of deleted data, including both structural artifacts and side effects. The approaches can also securely remove data themselves, and may be incorporated into different layers of a storage system.

## Future directions

This section provides an overview of the promising research directions for both PDE and secure deletion.

**Achieving plausible deniability and secure deletion in a single system.** Most of the existing systems either provide deniability or achieve secure deletion. However, data confidentiality should be simultaneously ensured during and after the lifetime of the data, because: First, by recovering the data being deleted, the adversary can achieve a similar gain comparable to compromising confidentiality of the data being stored; Second, if the confidentiality of the data cannot be ensured during their lifetime, secure deletion (i.e., ensuring confidentiality of the data after their lifetime) turns meaningless since the adversary has already obtained the data before they are "securely" removed. Therefore, we expect a system which can achieve both plausible deniability and secure deletion. Simply combining the existing PDE and secure deletion may be problematic, since secure deletion may require a fine-grained encryption mechanism, and plausibly deniable encryption is not necessarily designed as fine-grained. In addition, when pre-processing data for secure deletion purpose, careful consideration may be needed to avoid bringing in any deniability compromises. The only attempt for this type of system is DEFY (Peters

**Table 5** Comparison of features offered by secure deletion approaches designed to handle impacts created by past existence of the deleted data

| | NFPS (Jia et al. 2016) | TedFlash (Chen et al. 2016) | HIFS (Bajaj and Sion 2013b; Bajaj et al. 2016) | HiFlash (Chen and Sion 2015) | Ficklebase (Bajaj and Sion 2013a) |
|---|---|---|---|---|---|
| Method to sanitize data | Overwriting | Encryption | Overwriting | Overwriting | Encryption |
| Method to sanitize past impacts | Partial scrubbing | Random placement technique | History independent hash table | One-one mapping | Versioning, query rewriting |
| Layer | Controller (FTL) | Controller (FTL) | File system | Controller (FTL) | Application |

et al. 2015), but DEFY itself may suffer from deniability compromise (Jia et al. 2017).

**Providing confidentiality guarantee for light-weight computing devices.** Computing devices nowadays are turning more and more light-weight. Wearable devices like smart watches and smart glasses, IoT devices like smart home hubs or smart plugs, are increasingly popular today. Those light-weight computing devices are usually equipped with limited computational power. However, both the PDE and the secure deletion usually require expensive encryption operations, and thus cannot directly fit the use of light-weight devices. This can be mitigated by either outsourcing part of the expensive computation to the third-party cloud providers (without confidentiality compromise) or reducing the level of security to improve performance.

**Eliminating deniability compromise and data leakage.** The existing PDE/secure deletion systems mainly focus on external storage, and may neglect the security leakage in other sources like memory. By having obtained snapshots of the victim computing device, the adversary may capture the memory state, and may compromise deniability or recover the data being deleted by performing forensic analysis. Therefore, processing of the sensitive data should be conducted in an isolated memory region, which cannot be learned by the adversary. This isolated memory region can be created by using trusted execution environment (e.g., Intel SGX (Intel: Intel software guard extensions 2017)).

**Secure against quantum computing.** Both PDE and secure deletion may rely on encryption (e.g., AES-128, XTS-AES), which is not necessarily quantum resistant. The development of quantum computers seems to be expedited recent years (Wikipedia: Timeline of quantum computing 2018). Therefore, there will be a need to ensure that the PDE and the secure deletion are secure against quantum computing, which requires carefully checking the existing cryptography primitives being used and replacing those which are vulnerable to quantum computing with the ones that are quantum resistant.

## Conclusion

In this survey, we summarize the existing approaches for protecting data confidentiality against snapshot adversaries. Our survey covers techniques for both PDE and secure deletion, which can ensure data confidentiality during and after their lifetime, respectively. We also discuss a few promising future directions.

**Acknowledgment**
This work was partially supported by the National Key Research & Development Program of China (Grant No. 2017YFC0822704) and National Natural Science Foundation of China (No. 61602476, No. 61772518 and No. 61602475).

**Authors' contributions**
All authors read and approved the final manuscript.

**Competing interests**
The authors declare that they have no competing interests.

## Publisher's Note

**Author details**

[1]Data Assurance and Communication Security Research Center, Chinese Academy of Sciences, Beijing, China. [2]State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. [3]School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China. [4]School of Information Systems, Singapore Management University, Singapore, Singapore. [5]Department of Computer Science, Michigan Technological University, Houghton, USA.

Received: 19 January 2018 Accepted: 17 April 2018
Published online: 05 June 2018

**References**

Amazon: New SSD-Backed Elastic Block Storage (2016). Retrieved December 17, 2017, from https://aws.amazon.com/blogs/aws/new-ssd-backed-elastic-block-storage/

Anderson R, Needham R, Shamir A (1998) The steganographic file system. In: International Workshop on Information Hiding. Springer. pp 73–82

Apple: Use filevault to encrypt the startup disk on your mac (2017). Retrieved December 17, 2017, from https://support.apple.com/en-us/HT204837

Assange J, Weinmann RP, Dreyfus S (2012) Rubberhose: Cryptographically deniable transparent disk encryption system. Retrieved December 17, 2017, from https://web.archive.org/web/20120716034441/http://marutukku.org/

Bajaj S, Chakraborti A, Sion R (2016) Practical foundations of history independence. Information Forensics and Security. IEEE Transactions on 11(2):303–312

Bajaj S, Sion R (2013) Ficklebase: Looking into the future to erase the past. In: Data Engineering (ICDE) 2013 IEEE 29th International Conference on. IEEE. pp 86–97

Bajaj S, Sion R (2013) Hifs: History independence for file systems. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM. pp 1285–1296

Ballard Spahr LLP: State and local governments move swiftly to sue equifax (2017). Retrieved December 17, 2017, from https://www.natlawreview.com/article/state-and-local-governments-move-swiftly-to-sue-equifax

Bauer S, Priyantha NB (2001) Secure data deletion for linux file systems. In: Usenix Security Symposium. Volume 174

Blass EO, Mayberry T, Noubir G, Onarlioglu K (2014) Toward robust hidden volumes using write-only oblivious ram. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM. pp 203–214

Boneh D, Lipton RJ (1996) A revocable backup system. In: USENIX Security Symposium. pp 91–96

Braga A, Colito A (2014) Adding Secure Deletion to an Encrypted File System on Android Smartphones. In: SECURWARE 2014, The Eighth International Conference on Emerging Security Information, Systems and Technologies. pp 106–110

Breeuwsma M, De Jongh M, Klaver C, Van Der Knijff R, Roeloffs M (2007) Forensic data recovery from flash memory. Small Scale Digit Device Forensic J 1(1):1–17

Cai Y, Luo Y, Ghose S, Mutlu O (2015) Read disturb errors in mlc nand flash memory: Characterization, mitigation, and recovery. In: Dependable Systems and Networks (DSN), 2015 45th Annua IEEE/IFIP International Conference on. IEEE. pp 438–449

Canetti R, Dwork C, Naor M, Ostrovsky R (1997) Deniable encryption. In: Annual International, Cryptology Conference. Springer. pp 90–104

Cbsnews: Apple's celebrity icloud leak probably has mundane causes (2014). Retrieved December 17, 2017, from https://www.cbsnews.com/news/apples-celebrity-icloud-leak-probably-has-mundane-causes/

Chakraborti A, Chen C, Sion R (2016) Poster: Datalair: A storage block device with plausible deniability. ACM

Chakraborti A, Chen C, Sion R (2017) Datalair: Efficient block storage with plausible deniability against multi-snapshot adversaries. Proc Priv Enhanc Technol 3:175–193

Chang B, Cheng Y, Chen B, Zhang F, Zhu WT, Li Y, Wang Z (2018) User-friendly deniable storage for mobile devices. Comput Secur 72:163–174

Chang B, Wang Z, Chen B, Zhang F (2015) Mobipluto: File system friendly deniable storage for mobile devices. In: Proceedings of the 31st Annual Computer Security Applications Conference. ACM. pp 381–390

Chang B, Zhang F, Chen B, Li Y, Zhu WT, Tian Y, Wang Z, Ching A (2018) Mobiceal: Towards secure and practical plausibly deniable encryption on mobile devices. In: Proceedings of the 48th IEEE/IFIP International Conference on Dependable Systems and Networks

Chen B, Jia S, Xia L, Liu P (2016) Sanitizing data is not enough!: towards sanitizing structural artifacts in flash media. In: Proceedings of the 32nd Annual Conference on Computer Security Applications. ACM. pp 496–507

Chen B, Sion R (2015) Hiflash: A history independent flash device. arXiv preprint arXiv:1511.05180

Congress US (1996) Health Insurance Portability and Accountability Act. Retrieved November 9, 2017, from http://www.hhs.gov/ocr/privacy/index.html

Congress US (1999) Gramm-Leach-Bailey Act. Retrieved November 9, 2017, from http://www.gpo.gov/fdsys/pkg/PLAW-106publ102/pdf/PLAW-106publ102.pdf

Czeskis A, Hilaire DJS, Koscher K, Gribble SD, Kohno T, Schneier B (2008) Defeating encrypted and deniable file systems: Truecrypt v5. 1a and the case of the tattling os and applications. In: HotSec

Durak B (2006) wipe - Linux man page. Retrieved December 17, 2017, from https://linux.die.net/man/1/wipe

Diesburg S, Meyers C, Stanovich M, Mitchell M, Marshall J, Gould J, Wang AIA, Kuenning G (2012) Trueerase: Per-file secure deletion for the storage data path. In: Proceedings of the 28th, Annual Computer Security Applications Conference. ACM. pp 439–448

Dürmuth M, Freeman DM (2011) Deniable encryption with negligible detection probability: An interactive construction. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer. pp 610–626

Frèche G, Bloch MR, Barret M (2017) Polar codes for covert communications over asynchronous discrete memoryless channels. Entropy 20(1):3

Garfinkel SL, Shelat A (2003) Remembrance of data passed: A study of disk sanitization practices. IEEE Secur Priv 99(1):17–27

Geambasu R, Kohno T, Levy AA, Levy HM (2009) Vanish: Increasing data privacy with self-destructing data. In: USENIX Security Symposium. pp 299–316

GEEP EDS LLC: Darik's Boot and Nuke (2017). Retrieved December 17, 2017, from http://www.dban.org/

Goldreich O, Ostrovsky R (1996) Software protection and simulation on oblivious rams. J ACM (JACM) 43(3):431–473

Gutmann P (1996) Secure deletion of data from magnetic and solid-state memory. In: Proceedings of the Sixth USENIX Security Symposium, San Jose, CA. Volume 14

Halderman JA, Schoen SD, Heninger N, Clarkson W, Paul W, Calandrino JA, Feldman AJ, Appelbaum J, Felten EW (2009) Lest we remember: cold-boot attacks on encryption keys. Commun ACM 52(5):91–98

Han J, Pan M, Gao D, Pang H (2010) A multi-user steganographic file system on untrusted shared storage. In: Proceedings of the 26th Annual Computer Security Applications Conference. ACM. pp 317–326

Howlader J, Basu S (2009) Sender-side public key deniable encryption scheme. In: Advances in Recent Technologies in Communication and Computing, 2009. ARTCom'09. International Conference on. IEEE. pp 9–13

Hu J, Yan S, Zhou X, Shu F, Wang J (2017) Covert Communication in Wireless Relay Networks. In: GLOBECOM 2017 - 2017 IEEE Global Communications Conference. pp 1–6

Ibrahim MH (2009) Receiver-deniable public-key encryption. IJ Netw Secur 8(2):159–165

Incits: Scsi storage interfaces (2016). Retrieved December 17, 2017, from http://www.t10.org/index.html

Intel: Intel software guard extensions (2017). Retrieved December 19, 2017, from https://software.intel.com/en-us/sgx

Intel Corporation: Intel Solid-State Drive Optimizer (2009). Retrieved December 17, 2017, from http://download.intel.com/design/flash/nand/mainstream/IntelSSDOptimizerWhitePaper.pdf

Jagdmann D (2015) srm - Linux man page. Retrieved December 17, 2017, from https://www.mankier.com/1/srm

Jia S, Xia L, Chen B, Liu P (2016) Nfps: Adding undetectable secure deletion to flash translation layer. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. ACM. pp 305–315

Jia S, Xia L, Chen B, Liu P (2017) Deftl: Implementing plausibly deniable encryption in flash translation layer. In: Proceedings of the 24th ACM conference on Computer and communications security. ACM

Joukov N, Papaxenopoulos H, Zadok E (2006) Secure deletion myths, issues, and solutions. ACM Work Storage Secur Survivability:61–66

Joukov N, Zadok E (2005) Adding secure deletion to your favorite file system. In: Third IEEE International Security in Storage Workshop (SISW'05). IEEE. p 8

Klonowski M, Kubiak P, Kutyłowski M (2008) Practical deniable encryption. SOFSEM 2008: Theory Pract Comput Sci:599–609

Lee B, Son K, Won D, Kim S (2011) Secure data deletion for usb flash memory. J Inf Sci Eng 27(3):933–952

Lee C, Sim D, Hwang JY, Cho S (2015) F2fs: a new file system for flash storage. Usenix Conf File Storage Technol:273–286

Lee J, Heo J, Cho Y, Hong J, Shin SY (2008) Secure deletion for nand flash file system. In: Proceedings of the 2008 ACM symposium on Applied computing. ACM. pp 1710–1714

Lee J, Yi S, Heo J, Park H, Shin SY, Cho Y (2010) An efficient secure deletion scheme for flash file systems. J Inf Sci Eng 26(1):27–38

LSoft Technologies Inc: Active@ KillDisk (2017). Retrieved December 17, 2017, from http://www.killdisk.com/

McDonald AD, Kuhn MG (2000) Stegfs: A steganographic file system for linux. In: Information Hiding. Springer. pp 463–477

Memory Technology Devices: UBIFS (2015). Retrieved December 17, 2017, from http://www.linux-mtd.infradead.org/doc/ubifs.html

Meng B, Wang Jq (2009) A receiver deniable encryption scheme. In: International Symposium on Information Processing (ISIP '09)

Meng B, Wang J (2010) An efficient receiver deniable encryption scheme and its applications. JNW 5(6):683–690

Microsoft: Bitlocker (2017). Retrieved December 17, 2017, from https://docs.microsoft.com/en-us/windows/device-security/bitlocker/bitlocker-overview

Min C, Kim K, Cho H, Lee SW, Eom YI (2012) Sfs: random write considered harmful in solid state drives. In: FAST. p 12

O'Neill A, Peikert C, Waters B (2011) Bi-deniable public-key encryption. Adv Cryptol–CRYPTO 2011:525–542

Pang H, Tan KL, Zhou X (2003) Stegfs: A steganographic file system. In: Data Engineering, 2003. Proceedings. 19th International Conference on. IEEE. pp 657–667

Pang H, Tan KL, Zhou X (2004) Steganographic schemes for file system and b-tree. IEEE Trans Knowl Data Eng 16(6):701–713

Perlman R (2005) The Ephemerizer: Making Data Disappear. In: Journal of Information System Security. Citeseer

Perlman R (2005) File system design with assured delete. In: Third IEEE International Security in Storage Workshop (SISW'05). IEEE. p 6

Peters TM, Gondree MA, Peterson ZN (2015) DEFY: A deniable, encrypted file system for log-structured storage. In: 22th Annual Network and Distributed System Security Symposium. NDSS

Peterson ZN, Burns RC, Herring J, Stubblefield A, Rubin AD (2005) Secure deletion for a versioning file system. In: FAST. Volume 5. pp 143–154

Plumb C (2010) shred 1 - Linux man page. Retrieved December 17, 2017, from https://linux.die.net/man/1/shred

Qin Y, Tong W, Liu J, Zhu Z (2013) Smsd: A smart secure deletion scheme for ssds. J Converg 4:30–35

Kissel R, Scholl M, S S, Li X (2006) Guidelines for Media Sanitization. Retrieved December 17, 2017, from http://ws680.nist.gov/publication/get_pdf.cfm?pub_id=50819

Reardon J, Basin D, Capkun S (2013) Sok: Secure data deletion. In: Security and Privacy (SP), 2013 IEEE Symposium on. IEEE. pp 301–315

Reardon J, Capkun S, Basin D (2012) Data node encrypted file system: Efficient secure deletion for flash memory. In: Proceedings of the 21st USENIX conference on Security symposium, USENIX Association. pp 17–17

Reardon J, Marforio C, Capkun S, Basin D (2012) Secure deletion on log-structured file systems. ASIACCS 2012

Reardon J, Ritzdorf H, Basin D, Capkun S (2013) Secure data deletion from persistent media. In: ACM Sigsac Conference on, Computer & Communications Security. pp 271–284

Robust Flash Storage: YAFFS (2002). Retrieved December 17, 2017, from http://www.yaffs.net/yaffs-overview

Roche DS, Aviv A, Choi SG, Mayberry T (2017) Deterministic, stash-free write-only oram. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ACM. pp 507–521

Sarbanes USP, Oxley URMG (2002) Sarbanes-Oxley Act. Retrieved November 9, 2017, from http://www.sec.gov/about/laws.shtml#sox2002

Skillen A. (2013) Deniable storage encryption for mobile devices. PhD thesis. Concordia University

Skillen, A, Mannan M (2013) On implementing deniable storage encryption for mobile devices

Skillen A, Mannan M (2014) Mobiflage: Deniable storage encryptionfor mobile devices. IEEE Trans Dependable Secure Comput 11(3):224–237

Sourceforge: FreeOTFE A free "on-the-fly" transparent disk encryption program for PC and PDAs (2017). Retrieved December 19, 2017, Project website: https://sourceforge.net/projects/freeotfe.mirror/

Sourceware: Jffs2 (2003). Retrieved November 9, 2017, from https://www.sourceware.org/jffs2/

SQLite: Pragma statements (2017). Retrieved December 17, 2017, from http://www.sqlite.org/pragma.html#pragma_secure_delete

Stahlberg P, Miklau G, Levine BN (2007) Threats to privacy in the forensic analysis of database systems. ACM SIGMOD conf Manag data:91–102

Stefanov E, Van Dijk M, Shi E, Fletcher C, Ren L, Yu X, Devadas S (2013) Path oram: an extremely simple oblivious ram protocol. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. ACM. pp 299–310

Sun K, Choi J, Lee D, Noh SH (2008) Models and design of an adaptive hybrid scheme for secure deletion of data in consumer electronics. Consumer Electronics. IEEE Trans 54(1):100–104

Swanson S, Wei M (2010) Safe: Fast, verifiable sanitization for ssds. University of California-San Diego, San Diego, CA

Tang Y, Lee PP, Lui J, Perlman R (2012) Secure overlay cloud storage with access control and assured deletion. Dependable and Secure Computing. IEEE Trans 9(6):903–916

Team work systems: Advanced technology attachment (2017). Retrieved December 17, 2017, from https://teamws.com/glossary/ata-advanced-technology-attachment/

Thornber J Thin provisioning tools. https://github.com/jthornber/thin-provisioning-tools

Troncoso C, Diaz C, Dunkelman O, Preneel B (2007) Traffic analysis attacks on a continuously-observable steganographic file system. In: International Workshop on Information Hiding. Springer. pp 220–236

TrueCrypt: Free open source on-the-fly disk encryption software.version 7.1a (2012). Retrieved December 19, 2017, Project website: http://www.truecrypt.org/

Varun S, Shibin K, Anoop S, Vivek K (2017) Magikfs - The Steganographic Filesystem On Linux. Retrieved December 19, 2017, Project website: http://magikfs.sourceforge.net/

Wang Z, Murmuria R, Stavrou A (2012) Implementing and Optimizing an Encryption Filesystem on Android. In: IEEE International Conference on Mobile Data Management. pp 52–62

Wei MYC, Grupp LM, Spada FE, Swanson S (2011) Reliably erasing data from flash-based solid state drives. In: FAST. Volume 11. pp 8–8

Wikipedia: Timeline of quantum computing (2018). Retrieved December 19, 2017, from https://en.wikipedia.org/wiki/Timeline_of_quantum_computing

Yu X, Chen B, Wang Z, Chang B, Zhu WT, Jing J (2014) Mobihydra: Pragmatic and multi-level plausibly deniable encryption storage for mobile devices. In: Information Security - 17th International Conference, ISC 2014, Hong Kong, China, October 12-14 2014. Proceedings. pp 555–567

Zarras A, Kohls K, Dürmuth M, Pöpper C (2016) Neuralyzer: Flexible expiration times for the revocation of online data. In: Proceedings of the Sixth ACM on Conference on Data and Application Security and Privacy. ACM. pp 14–25

Zhao L, Mannan M (2015) Gracewipe: Secure and verifiable deletion under coercion. In: Network and Distributed System Security Symposium

Zhou X, Pang H, Tan KL (2004) Hiding data accesses in steganographic file system. In: Data Engineering, 2004. Proceedings. 20th International Conference on. IEEE. pp 572–583

Zuck A, Shriki U, Porter DE, Tsafrir D (2017) Preserving hidden data with an ever-changing disk. Work Hot Top Oper Syst