Secure storage-Confidentiality and authentication

Ryad Benadjila, Louiza Khati, Damien Vergnaud

Secure storage - Confidentiality and Authentication

Ryad Benadjila[a], Louiza Khati[a], Damien Vergnaud[b]

[a] *ANSSI, France*
[b] *Sorbonne Université, CNRS, LIP6, F-75005 Paris, France and Institut Universitaire de France*

Abstract

Secure disk storage is a rich and complex topic and its study is challenging in theory as well as in practice. In case of loss or theft of mobile devices (such as laptops and smartphones), the threat of data exposure is important and a natural security objective is to guarantee the confidentiality of the data-at-rest stored in such devices (e.g. on disks or solid-state drives). Classical approaches to encrypt data may have a severe impact on performance if the underlying architectural specificities are not considered. In particular, it is usually assumed that an encryption scheme suitable for the application of disk encryption must be length preserving. This so-called "full disk encryption" method provides confidentiality but does not provide cryptographic data integrity protection. It indeed rules out the use of authenticated encryption where an authentication tag is concatenated to the ciphertext. Moreover, authenticated encryption requires storing tags, and latency is added due to extra read/write accesses and tag computations. We present a comprehensive study of full disk encryption solutions and compare their features from a security perspective. We additionally present threat models for authenticated disk encryption and present a systematized analysis of the techniques usable in these settings (which has, up to now, received little attention from the research community). We finally review the current state-of-the-art of incremental cryptography and provide new insights for its use in secure disk storage contexts.

Keywords:
Full Disk Encryption, Authentication, Incremental Cryptography

1. Introduction

The proliferation of mobile electronic devices has made it critical to protect the on-device data, as mobile phones or laptops are easily lost or stolen. When this occurs, extremely valuable and sensitive stored information might be at risk. An important security objective is therefore to ensure confidentiality of such on-device data. Mobility implies strong constraints on the methods to be used for secure disk storage. The encryption algorithm has to be computationally and space-efficient with respect to the storage for read/write accesses to the data.

It is usually assumed that an encryption scheme suitable for the application of disk encryption must be length preserving. This so-called *Full Disk Encryption (FDE)* method is implemented in a large panel of devices, from compact smartphones to larger and more powerful ones such as servers. FDE is known to guarantee the confidentiality in the disk at a low level (e.g. sector level). Obviously, reading and writing operations should suffer as little as possible from performance degradation that occurs when one deploys FDE. This requires FDE to have random access to every single sector on the drive and impacts different layers in a device: from the disk memory cells to the operating system.

FDE is now a mainstream technology that provides confidentiality but unfortunately does not provide cryptographic data integrity protection. However, if data confidentiality is necessary, it is perhaps even more critical to prevent data alteration. Up to now, most disk encryption solutions rely on the so-called *poor-man's authentication* to ensure integrity, meaning if an attacker alters some encrypted data on the disk, the corresponding plaintext will be scrambled unpredictably, yielding an alert in the operating system or application layer. However, this approach is not cryptographically viable and a stronger integrity mechanism should be provided (e.g. some applications silently ignore errors when parsing files). This protection comes with a cost: authenticity-oriented cryptosystems store extra tags with additional latency to access and compute them. Hence, it is important to analyse if a cryptographic mechanism guarantees data authenticity with minimal performance impact and minimal storage.

This expository work aims at giving a state-of-the-art of cryptographic constructions used in FDE and the rationale behind their design (storage constraints and performance goals). We also formalize new models where, in addition to confidentiality, two levels of data authentication at a low-level

are possible.

Organization of the paper. This paper is organized as follows. In section 2, data at rest threats are analysed to present adversary models and their relevance to FDE. Disk storage technologies and their constraints are described. Section 3 exposes the challenges of FDE regarding these constraints while keeping decent performance. Section 4 explores the possibility to add data authentication. Two strategies are detailed: local and global authentications. The first one only offers a block-level authentication but does not prevent replay attacks (restoring a former version of the disk content). The second one protects the entire disk even against replay attacks by using specific authentication schemes that have the property to be *incremental*: tags are updated instead of being re-computed from scratch. The implications of these authentication schemes in terms of storage and update time are analysed and discussed.

2. Data Protection and Data Storage

2.1. Data Protection

The need for user data protection has become critical, emphasized by smartphones and cheap outsourced storage democratization. Mobile devices can be left unattended, sent for repair, sold, given to legal authorities during checks (e.g. in airports), lost, or stolen. In all these common situations, an untrusted party can have access to the user data stored in the device (even when switched off).

In this paper, **a device** refers to a system that runs autonomously with *a persistent memory to protect*. FDE concerns data at rest stored in the disk only. Data is encrypted before being stored in the disk and decrypted after being loaded from the disk which means that the disk should always store encrypted data[1]. Data is said to be encrypted and decrypted "on-the-fly".

Threat Models. All these devices store some amount of data that needs to be protected in confidentiality, and different solutions exist depending on the model (architecture and adversary). In the following, we gather all these scenarios in three main attack categories according to how the adversary accesses the disk.

[1]The strategy used for the initial encryption step depends on the FDE tool.

1. **Single passive access to the disk.** When a device is stolen or lost, the user expects data confidentiality insurance: anyone except him should not learn any information about clear data. The adversary has access to the encrypted data and aims at discovering information about the corresponding plaintexts.

2. **Multiple passive access to the disk.** It encloses -1- but here the adversary is more powerful as they can make copies of the disk at different times. The adversary can analyse these copies to track modifications. Later, we will see that even with a strong FDE encryption mode, an adversary is able to guess exactly which parts have been changed between copies.

3. **Active access to the disk.** This adversary encloses -1- and -2- with the ability to tamper with the disk data. These attacks are also known as "active attacks" [26]. As an active attack, we can think about silent data corruption [4] even if it is not an intentional attack as it can be due to random hardware failures. This issue is usually solved by non-cryptographic checksums.

A possible scenario for this latest model is an adversary that succeeds to perform malicious modifications in the disk without the user realizing it. They can also tamper with the disk and can give it back to the user who does not notice data corruption. An example is a copy-paste attack where a disk is shared between a user 1 and a user 2: the malicious user 1 copies data from the user 2 part of the disk and pastes it in its own part. As in many FDE tools, a unique key is used for the entire disk (more details in section 3.1); user 1 could be able to decrypt user 2's data.
In addition to data confidentiality, the user wants to be able to check the **authenticity** of the data they is using. A specific tampering attack is what we can call a "downgrade attack" or a "replay attack": the adversary snapshots the entire disk at time t and waits for a specific moment to restore it. The desired property to thwart this is a so-called **temporal data authentication**: *at any moment*, the user wants to be sure that the data they is manipulating now is the data stored during the previous legitimate manipulation.

For each category, an adversary can perform **online attacks** which means that they has physical access to the device. As a consequence, they can attack the disk content but also the other components (e.g. hardware parts). Otherwise, if it performs **offline attacks**, we suppose that they has no access

to the entire device but only to the raw disk memory content, i.e. storage memory cells.

2.2. From Disk Storage to Data Encryption

We provide a simplified description of a system as a basis for discussing the FDE problematic and analyzing the differences between FDE products.

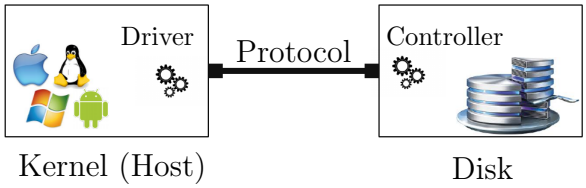

Figure 1: View of components involved in disk encryption.

First, let us define the taxonomy of the FDE-related components. At the highest level, we consider systems running operating systems (OS) and hosting applications. The core component of the OS is its kernel. In a nutshell, the BIOS is the first piece of code executed at power on, loading an on-disk bootloader that runs the kernel that handles the memory and the peripherals after boot time. It manages data stored in the disk through different virtualization layers as shown in Fig. 2. From an end-user perspective, data is abstracted in the form of files aggregated in filesystems that can be stored in volumes or partitions (a volume can be split among different partitions). The kernel handles physical disks through partitions with specific drivers in charge of low-level read/write operations. As shown in Fig. 1, a dedicated protocol conveys read/write commands at the physical layer. Such commands are limited to a fixed-size data unit called a sector. The low-level driver is consequently in charge of sector-based read and write operations. The Parallel ATA (PATA) and Serial ATA (SATA[18]) standards are examples of such protocols commonly used as classical disk interfaces. The physical disk contains a controller and non-volatile memory units to store data sectors. The controller is usually a dedicated micro-controller with embedded firmware that deals with the commands received from the kernel driver and manages the internal non-volatile memory depending on the disk technology.

Disks. The older and well-known technology is the Hard Disk Drive (HDD), read and written by a mechanical arm handled by the disk controller. Each drive has one or more disk platters allowing to store and retrieve a sector. The controller communicates with a driver using a protocol that enables write

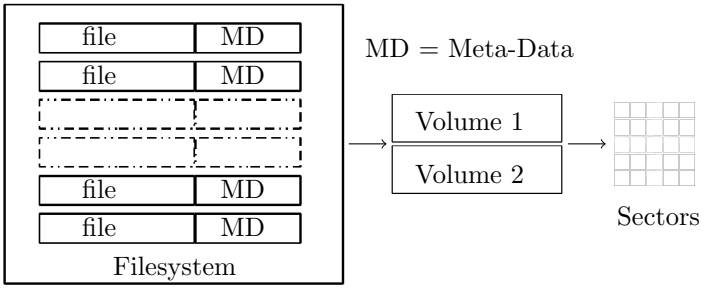

Figure 2: Simplified logical levels: from files to sectors

and read commands depending on sector numbers, also called Logical Block Addresses (LBA).

The physical composition of newer Solid State Drives (SSD) is totally different: they are flash memory devices gradually replacing the HDDs with improved performance. Similar to magnetic drives, they are logically sector-addressable devices. The flash memory cells of an SSD are hierarchically organized as a set of flash chips called packages, which are further divided into dies, planes, blocks[2] and pages. Every page consists of one or more sectors and is the smallest unit that can be written. The Flash Translation Layer (FTL) stores the mapping between LBAs and Physical Block Addresses (PBAs). The FTL implements an abstraction layer to be compatible with systems using HDD: the driver reads and writes sectors using LBA. Disk interfaces such as PATA or SATA have different physical link properties, speed, and communication protocols but they all enable access to sector granularity. The driver and other components from the OS manage data and optimize read and write access.

Physical and Logical sectors. The physical sector is the sector used internally in the disk and the logical sector is the one presented to the host by the disk controller. The kernel asks to write at least one logical sector in an **atomic** operation from its point of view: the host cannot ask to write less than one sector. Historically, HDDs had 512-bytes sectors that were presented to the host: at the time, logical and physical distinctions did not exist. To read and write disk data efficiently, the OS used a unit called a

[2]These blocks should not be confused with the blocks of the blockcipher, nor with the "block" (actually "sector") in the term Logical Block Address (LBA).

page[3] that is a virtual unit composed usually by eight logical sectors. A page is the smallest unit the OS can read and write and, its size is a multiple of the logical sector size. OS storage page size is commonly chosen to match the virtual memory page size (usually 4096 bytes): this allows optimizing file system transfers between disk and RAM as well as caching. Nowadays, disk manufacturers are able to produce disks with larger sectors on par with the OS page size to increase low-level performance. This heterogeneous situation regarding logical and physical sectors sizes explains why the Advanced Format standard has been created to specify various compatibility modes.

3. Full Disk Encryption

3.1. Challenges

Encryption can be implemented at different levels from file to sector-based encryption (see Fig. 2). File encryption aims at encrypting independently each file usually without encrypting the corresponding meta-data. With filesystem encryption, the files and the meta-data (including the internal hierarchy) are encrypted (see ecryptfs [50] for instance). Volume encryption, partition encryption and disk encryption use a block-oriented encryption known as Full Disk Encryption (FDE). Blocks refers to sectors as previously introduced. Equivalent qualifiers are "Low-level full disk encryption" or "whole disk encryption" [51]. FDE advantageously ensures unconditional and transparent encryption of all the data stored in the disk.

FDE COST. Disk memory access is time-consuming: this is a critical issue for manufacturers and academic research. Huge efforts have been put into optimizing memory accesses to achieve tremendous advances in memory latency. Memory subsystems are tuned as much as possible with cache memories and fast SRAM memory chips to speed up computer capabilities. Adding an encryption layer could affect **performance** drastically. FDE framework developers try to integrate encryption that takes into account this optimization work. A consequence is that FDE is **length-preserving**: all the sectors are encrypted **independently** without additional data; otherwise sector (un)alignment management becomes a problem. Moreover most of FDE implementations aim at being compatible with a maximum number

[3]It is different from SSD pages.

of systems. A large panel of FDE frameworks tries to defeat different adversaries by implementing encryption at different layers and by managing the key differently.

Key management. To decrypt data, the user has to provide a secret; usually a password or a pin code. The secret will unlock the decryption key, which is then used to decrypt the whole disk. More complex key managements[4] are also implemented in some devices, but in this paper, we consider the classical FDE encryption schemes where the whole disk is encrypted with a unique key. This choice is mainly due to shared disk usage, password modification, and backup. The location of the key management, the kernel driver or the disk controller, is a way to define two families of FDE products: the first one called Self-Encrypted Disks (SEDs) corresponds to stand-alone systems performing encryption/decryption inside the disk by the controller and the other corresponds to software implementations performing encryption/decryption on the host side. Although password derivation and key management are obviously crucial topics for disk security, we consider them out of the scope of the current article as we only focus on FDE itself after the main key has been recovered.

Standard Encrypted Disks. A standard disk (either HDD or SSD) usually aims only at storing data without encryption. Software FDE solutions like dm-crypt [1] (Linux), Bitlocker [20] (Windows), FileVault [15] (MacOS), Truecrypt/Veracrypt(multi-OS) [3] perform encryption through the OS and store the encrypted data in the standard disk. Even though such implementations can take advantage of hardware-accelerated instructions such as AES-NI, we refer to them as software-based solutions as opposed to dedicated hardware solutions where most of the FDE logic is performed in dedicated controllers. In software-oriented solutions, the standard disk does not manipulate raw user secrets neither the raw cryptographic key but the host does. An adversary can take advantage of this by targeting the system RAM: these are out-of-scope attacks. We only focus on adversaries with access to the standard disk and aim at breaking confidentiality.

Self-Encrypted Disks. In SEDs, encryption is performed by a dedicated hardware in the disk itself. The SED is supposed to be unlocked only

[4]Disk encryption key management is a large subject that will not be fully discussed in this paper: only the necessary elements are given.

by the legitimate user. The user authentication is performed directly on a physical interface like a pinpad or through a trusted computer. Encryption/decryption is performed on the device and the encryption key should never leave it [41].

Software/Hardware Implementation. Software implementations are easy to develop and maintain contrary to hardware ones. But hardware implementations have the advantage of efficiency, reliability [5] and are potentially more resistant to attack vectors such as side-channel leakage. This explains why SEDs are usually found at a higher price compared to software-based solutions. Different SEDs were analysed in [40] and the authors exploited flaws in the firmware of the micro-controllers handling the hardware encryption. This study disproves the common belief that hardware FDE solutions are more secure than software-based ones. Building a secure FDE solution is however not straightforward: it is a complex process and involves versatile skill sets. The resistance of the cryptographic mechanism, the key derivation function, the usage of hardware keys are not sufficient: the encryption and the firmware implementation are also part of this mechanism and none of them should be neglected. Fully controlling the design and production chain, which is the case for SEDs and some smartphones, can help but can affect the implementation neatness and consistency.

Performance. The performance of an FDE solution can encompass the execution time and for some specific usages power consumption. The execution time depends essentially on the cryptographic algorithm, the way the tool is implemented, the number of the read and write operations. The cryptographic algorithm efficiency includes among others its structure (number of basic operations), its parallelization capabilities for encryption and decryption, the key size. An implementation can be more or less optimized to speed up the execution time. And a dedicated hardware implementation should have a smaller execution time than its software version. Obviously, read and write operations on the disk should be minimized.
It is also desirable to limit the power consumption of the system for cost-efficiency reasons. For smartphones, it is crucial because their usability depends on battery life, and too much power dedicated to cryptographic computations is not an interesting deal. FDE mechanisms should therefore seek energy consumption limitation [25].

3.2. FDE and Cryptography

As argued above, writing and reading a sector have to be as quick as possible which implies that encryption/decryption delays have the same requirement. This is why each sector is encrypted **independently** and encryption relies on blockciphers for their time **performance**. A secure blockcipher reveals no information about the plaintext knowing the ciphertext as long as the key is randomly sampled and kept secret. Today, the most used blockcipher is AES (128 bits block length). The sector length is a multiple of the block length and a secure encryption mode has to be used since using sequentially the blockcipher on the sector content will lead to insecure encryption (ECB mode). A full disk encryption mode uses the same key [26, 35] to encrypt the whole disk and it can be the case that the key is composed of sub-keys or used to derive sub-keys.

A widespread FDE mode is CBC-ESSIV [24] for "Cipher Block Chaining-Encrypted Salt-Sector IV". This mode is a CBC mode where the Initialization Vector (IV) is derived from the sector number. In CBC mode, the IV is required for decryption and it is stored as a first block in the ciphertext which is not possible for length preserving encryption. That is why, in CBC-ESSIV, for each sector, the IV is the encryption of the sector number s under a different key k' and the plaintext blocks are encrypted using the key k. This mode is parallelizable for decryption only and it is secure for an adversary belonging to the threat models -1- and -2-. This mode is less and less used and it should not be adopted because of its vulnerability to malleability attacks. *A malleability attack* consists in applying a transformation on a ciphertext block i and knowing the impact on the plaintext. To be able to perform a malleability attack, the adversary must have active access to the disk (threat model -3-). For CBC, flipping the j-th bit on the ciphertext block i will lead to decrypting the plaintext block $i + 1$ correctly but with the j-th bit also flipped as shown in figure 3. The i-th plaintext, which is 128 bits for AES, is randomized. This attack has severe impacts: a plaintext bit can be flipped at any position which gives the power to the adversary to change the plaintext block the way they wants. Practical attacks are demonstrated by Lell in [38] on dm-crypt. In the BitLocker solution, the elephant diffuser component [20], which is a keyed diffuser, is applied to the entire sector plaintext to mix all plaintext bits, and then CBC-ESSIV is used for encryption. The diffuser makes the malleability attack on CBC impractical. Nowadays, the standard for storage devices [31] specifies XTS mode for "XEX Tweakable blockcipher with ciphertext Stealing" which is based on XEX for

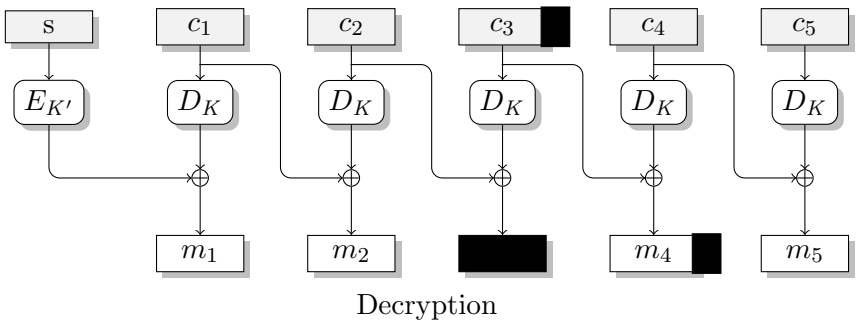

Figure 3: CBC-ESSIV malleability attack.

"Xor-Encrypt-Xor" designed by Rogaway [46]. *A tweakable blockcipher* is a blockcipher that takes not only the classical inputs, a key, and a plaintext, but also a tweak and outputs the ciphertext. A tweak is a public value and in the case of FDE, it is the sector number, hence there is no need to store this value. This trick allows having a length preserving mode to encrypt each sector. A modification on a sector can be seen by a passive adversary (threat model -2-): all unchanged plaintext blocks at the same position will have the same ciphertext blocks and the modified ones will be different. Then it provides a **"spatial" security** in the sense that each encrypted sector block looks random for the adversary as long as there is no repetition of plaintext block at the same position. If the adversary is active and modifies sector content (ciphertext), they is limited to tamper with one block only that corresponds to 128 bits in the case of AES usage. For example, they can flip one bit in the sector, after decryption the corresponding plaintext will look random (see figure 4) but all the other plaintext blocks will be decrypted normally. This property makes a tampering attack harder but does not prevent it. Moreover, XTS has the advantage to be parallelizable for encryption and decryption. Moreover, it uses only one AES call for each plaintext block.

Wide Tweakable Block Ciphers (WTBC) go further in limiting the tampering ability of the adversary but require more AES calls per block. A WTBC is based on a standard blockcipher that processes input blocks through multiple passes in order to simulate a blockcipher over the input size. This is why they are known to be slow [20] and much less used than XTS for example. In the FDE context, their block size is the sector size so an adversary, belonging to threat model -3-, corrupting one bit of the ciphertext

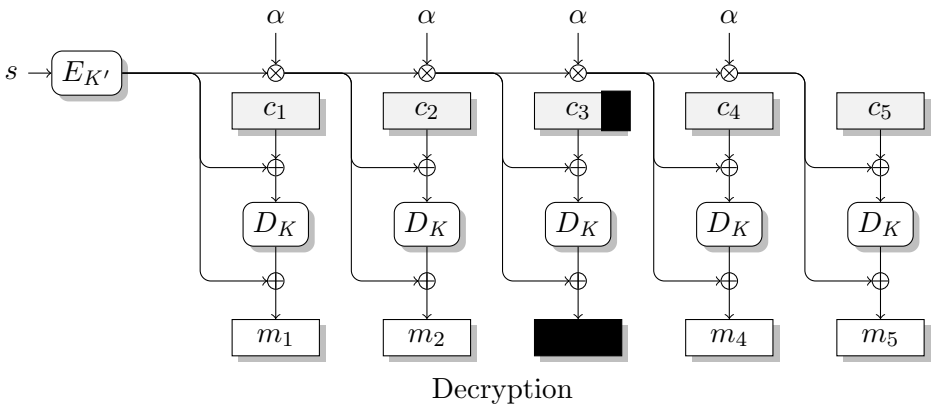

Figure 4: XTS malleability attack.

will lead to randomizing the entire plaintext sector (illustrated in figure 5). EME2 [27, 32], and XCB [39, 32] are examples of WTBC. Recently, a new WTBC, Adiantum [16], was introduced by Biggers and Crowley that is more efficient than XTS when there is no cryptographic accelerator.

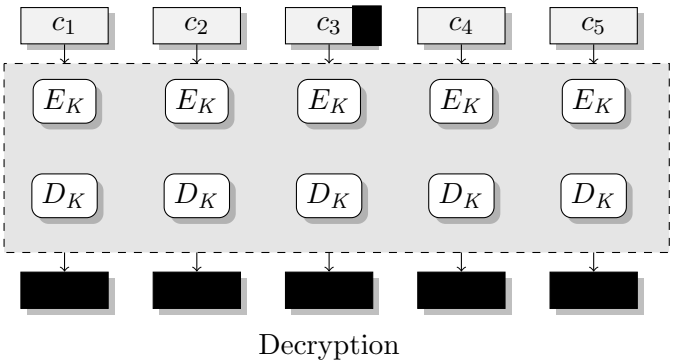

Figure 5: WBTC malleability attack.

4. Data Authentication

In the context of FDE the strongest notion of integrity is the *poor man's authentication* which means that if an adversary modifies the ciphertext (here a sector content), one can hope that this modification will lead to a random

change in the corresponding plaintext in such a way that the system or the user will detect this tampering. Relying on this method to protect disk content authenticity is not enough as an adversary can easily take advantage of the disk encryption algorithm malleability. Lell [38] implemented an attack against a volume encrypted using CBC-ESSIV as shown in figure 3. To be able to detect all the illegitimate data modifications, **data authentication** is required in addition to encryption. A Message Authentication Code (MAC) takes as input a message and returns a tag. This primitive is a keyed algorithm usually based on blockciphers [22] like CBC-MAC and CMAC or hash functions [23] like the HMAC family. Authentication for disk content is not a new subject [30, 12]; it is the purpose of the standard [33] and was the aim of different recent projects: the dm-integrity framework [11], dm-x [13] and StrongBox [19]. Having data authentication is out of FDE scope due to storage of additional data [48, 35, 11], which is the reason why the definition of new models are required. These new models bring new challenges: how should we store tags, which algorithms minimize the added computational latency (e.g. additional to encryption only latency), and additional data storage. The purpose of this section is to clarify the models where data authentication is possible and to provide solutions and their analysis.

4.1. Local Authenticity

A naive solution is to compute a tag over all the disk content using a MAC and only one tag is stored for the whole disk. However, then each time a sector is read or written, this tag has to be verified or re-generated which means processing all disk sectors; this is too costly. A better and natural mechanism (at the expense of space) is to compute a **local** tag for each sector to keep independence between each sector which costs to store a tag per sector. We call this per sector data authentication **local** or **spatial authentication**.

ADE Model. The possibility to store tags for each data sector gives a model that diverges from the FDE model we called **the Authenticated Disk Encryption (ADE) model**. Depending on how these tags are stored (see § 4.1), we can consider storing more than the local tags to make the cryptographic primitive stronger. That is why from here local tags refer to additional data stored for data protection including cryptographic tags, IV, ... In this model, the adversary has access to the disk several times and can modify its content: data sectors and tag sectors. This adversary is part of the

threat model -3-, their goal is to break confidentiality or to build a forgery for at least one data sector (e.g. data sector d and the corresponding tag τ). They wins if data confidentiality of a sector content is broken or if for the data sector number s, they builds a fresh data sector d^* and its corresponding tag τ^* where verification succeeds. A mechanism secure in the ADE model does not cover replay attacks [52]: an adversary having a copy of the disk (or a part of it) at the time t can replace the disk content at any moment after that time (replay attacks in theft with recovery described in § 2.1).

Authenticated Encryption (AE). To perform AE, we can choose among generic compositions or authenticated encryption.

Generic compositions [8]. There are three classical generic compositions: MAC-Then-Encrypt, MAC-and-Encrypt, or Encrypt-Then-MAC to achieve confidentiality and authentication with two keys. These keys can be derived from a master key or generated independently. These solutions store exactly the same amount of extra data (without taking into account the keys) and the choice between them has to be made according to security and calculation efficiency. From a security point of view, the composition Encrypt-Then-MAC is known to be secure if the underlying primitives are secure[5]. For example, XTS-AES can be composed with HMAC-SHA-2 as in [11], but the simple approach will lead to **sector reordering attacks**. As the MAC of a sector does not take as input the sector number, sector reordering will not be detected: an adversary changing the location of a sector together with its tag will obtain a valid MAC verification. The standard for authenticated encryption for storage devices [33] specifies that additional data for each sector should be added and the MAC ensures its authenticity.

Authenticated encryption schemes (AE). They are specific schemes that provide confidentiality and authenticity at the same time by design. They take usually a single key and use derivation function(s) to obtain the required key material. For a given plaintext, it outputs a ciphertext and a tag. As they are dedicated to this purpose, they can be more efficient. In 2009, some schemes were standardized among them CCM [21, 33], OCB [21] and GCM [21, 33] with AES as blockcipher. AES-GCM is a widely deployed AE scheme: it is used in protocols such as SSH [34], TLS [45], IPSec [54] and storage [33]. A few years ago, the CAESAR competition [9] was or-

[5]This is not straightforward as explained in [42], some care must be taken.

ganized with the goal to identify a portfolio of authenticated ciphers that offer advantages over AES-GCM and are suitable for widespread adoption. Specifically, robustness and tolerance against nonce misuse [53] (a missing feature of AES-GCM) have been one of the leitmotivs for the emergence of seven finalists ACORN, AEGIS, Ascon, COLM, Deoxys-II, MORUS, and OCB (see [9] for details).

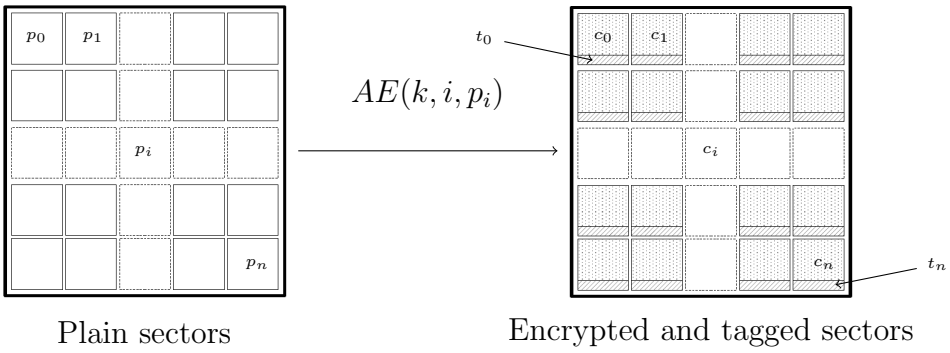

Figure 6: Disk Encryption and local authentication tag within each sector. Dashed boxes represent encrypted data and hatched boxes tags.

Local Tag Storage:. Different ways of storing local tags *on the disk* can be implemented and depending on the chosen strategy; the seek time can be more or less large.

Location. Local tags storage solutions can be: (1) to extend physical sector size; (2) to implement virtual sector smaller than physical sector (illustration in figure 6) or (3) to keep the same sector size and dedicate some of them to store tags.
Solution (1) was proposed in [14] by arguing that it is feasible for manufacturers to produce disks with bigger sector sizes because some space is already reserved for checksums. In this case, the only additional step is to modify the device mapper or the device controller to take into account authentication, but this solution can not be implemented in all deployed disks. Changing physical sector size is expensive so the majority of manufacturers will wait for a standard. Solution (1) relies on manufacturers.
Solution (2) avoids waiting for a new disk format; it is compliant with existing disks. The counterpart is implementing a virtualization layer. A basic virtualization strategy is to split the physical sector into two parts: data and

local tags. An example is the FreeBSD disk encryption GELI [17]: 480 bytes of data and a 32 bytes for the tag are stored in a 512 bytes physical sector. Solutions (3) breaks atomicity contrary to solutions (1) and (2). If for any reason, some sectors are written and the corresponding local tags are not (or the other way around), the authentication check will fail. If a write operation on the disk is interrupted, the previous sector content should be recovered otherwise authenticity check will fail. An example is dm-integrity: it implements interleaved meta-data sectors [10] where the meta-data includes local tags. A fixed number n of consecutive sectors are processed, and the corresponding meta-data is stored in the next sector (see figure 7). The user can choose the number n. The software dm-integrity allows the possibility to manage recovery on write failure with **journals**: for example, it can save an old data content of unfinished sector write. Enabling journals is an option and experiments with dm-integrity show that enabling them has severe impacts on performance. To be more specific, Milan Broz [10] shows that adding local tags to AES-XTS encryption with HMAC-SHA256 in dm-integrity decreases the read and write throughputs by 20% on average. And adding journalisation divides these numbers by 2!

Discussion. Disk sectors are now dedicated to storing actual data (D_d sectors) and local tags (D_{la} sectors) for solution (3), but this additional data estimation cost is applicable for all solutions. We have $D_s = D_d + D_{la}$ where D_s is the total number of sectors. Let τ_s be the tag size and S_s the sector size. The maximum number of tags a sector can store is $TS = S_s/\tau_s$. Then we have $D_{la} = D_d/TS$ and $D_d = (TS \times D_s)/(TS + 1)$ which means that the data sectors represent $(100 \times\ TS)/(TS + 1)$ percent of the disk and local tags represent $100/(TS + 1)$ percent. For a disk with a sector size of 4096 bytes ($S_s = 2^{12}$ Bytes) where 1 Terabyte (2^{40} Bytes) of actual data are stored, we have $D_d = 2^{40-12}$ sectors. Let us take the example of the following generic composition: the encryption primitive is AES-XTS encryption where the tweak is the sector number and the MAC primitive is HMAC-SHA-256. The HMAC is computed over the concatenation of the ciphertext and the sector number to avoid reordering attacks. The local tag size per sector data is 32 Bytes which corresponds to the tag produced by HMAC-SHA-256 ($\tau_s = 2^5$ Bytes). Then a sector stores 128 local tags ($TS = 2^7$) which gives $D_{la} = 2^{28-7}$ sectors (8 GB). Additional data represent only 0,8% of the disk but this estimation does not consider journals.

Data authentication protecting against replay attacks is a desirable prop-

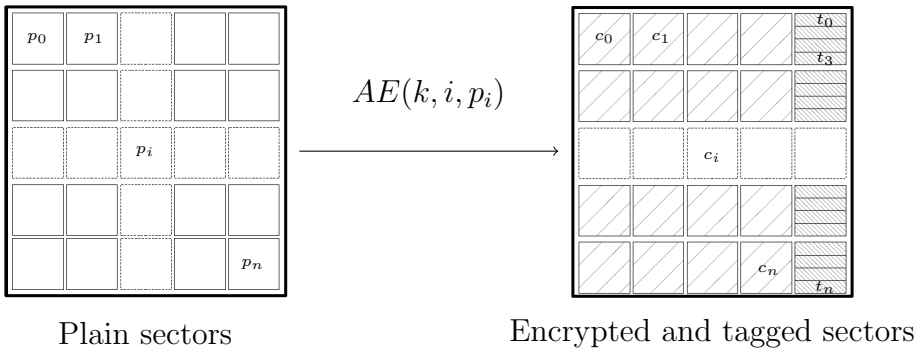

Figure 7: Interleaved Meta-data. Large dashed boxes represent encrypted data and small dashed boxes meta-data.

erty; we can call this security notion **temporal authenticity** or **global authenticity**. The next section aims at analyzing how we can get such property and at estimating its cost.

4.2. Global Authentication and Incremental Cryptography

Global authenticity is a strong security notion that aims at protecting all the local tags from tampering. A simple solution is to store all these tags in *persistent secure memory*, which means the adversary cannot tamper with it.

Secure Memory (SM). Obviously, such a component is not an accessible part of the disk otherwise downgrade attacks are still applicable. Now, we will assume that such memory exists otherwise replay attacks cannot be prevented in the context of full disk encryption [30, 11] and also in the context of secure cloud storage with block-level encryption [43, 13]. Disk and cloud storage are similar from a theoretical point of view in the sense that the server can be seen as a big remote disk. These two use cases are different; nonetheless, the data authenticity problem for data stored in a physical disk or a distant disk (cloud) can be solved similarly. For disk protection, we can consider a Secure Element (which usually embeds a small and persistent memory) as a SM but it can store only a limited amount of data typically a few kilobytes to a few megabytes [49]. Whereas for cloud storage, the SM can be associated with a dedicated client local storage where it is possible to store data that cannot be tampered with by the server. In this case, the SM can be a whole

disk partition that the local OS protects from any server access, yielding a few gigabytes of "secure memory".
Once again, the naive idea is to compute a global tag over all the local tags. Then reading a sector will lead to re-computation over all the local tags which is too much time-consuming.

FADE Model. We have a different model deviating from the ADE one called the **Fully Authenticated Disk Encryption (FADE)** where the adversary has the possibility to perform any modification on the disk. It breaks data authentication if it succeeds to build a forgery for a sector s different from the last legitimate one. In this model, the adversary can attempt replay attacks.

Global MACs. To be secure in the FADE model, a global MAC is needed, and it should have specific properties to be effective.
-**Security**: It has to guarantee the authenticity of the local tags.
-**Speed**: It has to be fast for tag generation and verification, minimize cryptographic operations, minimize the added read/write access to the disk; tag generation/verification must be relative to a data unit.
-**Locality**: If the verification fails, the index of tampered sectors should be easy to find.
-**Minimal storage in the disk**: If additional data has to be stored in the disk it should be as short as possible to save space in the disk.
-**Minimal storage in secure memory**: A global tag has to be stored in secure memory that has limited storage space.

Incremental Cryptography. Incremental cryptography was introduced by Bellare, Goldreich, and Goldwasser in [6] and it is an attractive feature that enables to efficiently update a cryptographic output like a ciphertext, a signature, or an authentication tag after modifying the corresponding input. A MAC can be incremental regarding some update operations like replacing, deleting, or inserting a block in the input. In our case, the input is the concatenation of the sector local tags computed with a classical MAC e.g, not an incremental MAC then for each modification in a sector; the local tag will be recomputed, and the global tag (output of incremental MAC) will be updated.

This efficiency comes from the computation of independent MACs over all the inputs (the values p_i in Fig. 8) and the resulting output blocks are combined to obtain the global tag τ. This independent processing seems to

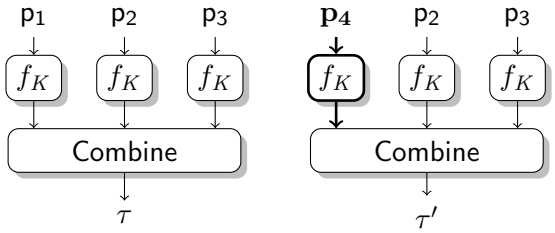

Figure 8: Overview of incremental replace operation. The function f_K denotes the MAC. The inputs τ_1, τ_2, τ_3 give the tag τ. After replacing τ_1 by τ_4, the tag τ is updated with τ'.

fit the independence of the disk sectors: as the sectors are modified independently, we can imagine a root tag τ for the whole disk that is updated for each sector modification without recomputing the root tag from scratch.

A disk is a fixed number of sectors, and local tags depend on the sector number so having an incremental MAC regarding the replace operation only is sufficient. The chaining Xor-Scheme [6, 36], Xor-MAC [7], GMAC [37] and Merkle tree [6] are incremental MACs. These algorithms use a keyed function f_K[6] and, they are compared in Table 1 with regard to the computational time of their tag generation algorithm, the replace update operation, and the tag verification algorithm. The table gives the number of calls to f_K for a disk composed of n sectors which means that the input of the tagging algorithm is the concatenation of the sector local authentication tags. In the following, *log* is for binary logarithm. For n inputs, the Xor-Scheme and the Xor-MAC have a constant time replacing operation (respectively 4 and 2 calls to f_K) and the storage cost is only a tag size (their description can be found in Appendix A.). The main drawback is the verification cost: to verify the authenticity of 1 sector tag; it requires n calls to f_K. Merkle tree is the only incremental scheme that has the locality property and a trade-off between replacing and verifying operation delays. The counterpart is that it requires more storage space, but it is not necessary to store the entire tree. In the next section, the Merkle tree storage will be discussed in detail.

4.3. Merkle Tree

A Merkle tree is an authentication data structure where leaves are the values to protect, and each node is the MAC of the concatenation of its

[6]To be more precise, f_K is a pseudo-random function.

	Tag	Replace	Verify	Storage	L
Chaining XS	n	4	n	1	N
Xor-MAC	$n+1$	2	$n+1$	1	N
Merkle tree	n	$log(n)$	$log(n)$	n	Y

Table 1: Operation and Storage costs of incremental MACS for n-block input. The column L is for *Locality property.*

children. In the following, Merkle trees are binary trees which means that each node is the MAC of its two children. This construction is used to ensure data authenticity at block level [43, 30, 28, 2], in cloud storage context [29, 13] and at file level [47] due to its incremental and locality properties.

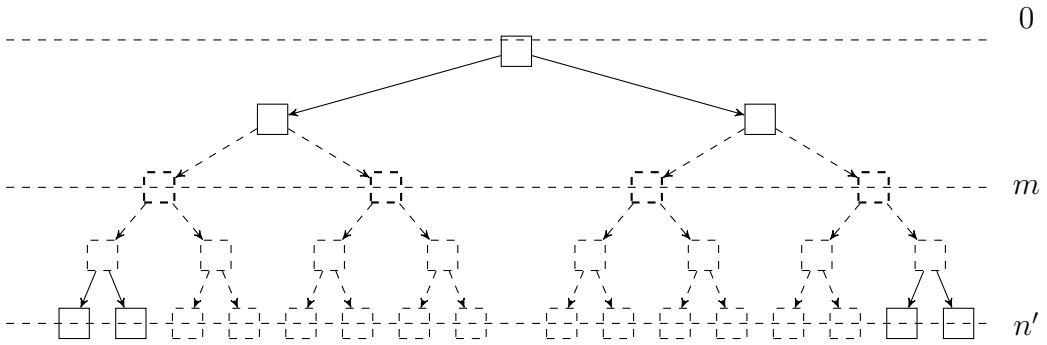

Figure 9: Perfect Merkle tree of n' levels. Nodes of level m only are stored ($m \leq n'$).

For a disk of n sectors, Merkle tree leaves are the n local tags where $n = 2^{n'}$.
This estimation is given for a perfect binary tree; the number of actual data is 2^n where n is an integer. In this paper, all the Merkle tree storage costs are computed according to a number of leaves (tags) equal to a power of 2 e.g. where $n = 2^{n'}$. Let L be the output size of the MAC algorithm f_K used to compute the Merkle tree nodes. Then storing an entire Merkle tree takes $(2^{n'} - 1)L$ bits.
There are different memories where it can be stored: the SM, the disk, and the RAM. But in any case, a copy of the trusted root has to be stored in the SM. This value must not be tampered with in order to recompute the path from the sector local tag to the root and check the obtained root and the

stored one. To simplify the analysis, we distinguish two ways of storing the tree: either it is entirely stored in the same memory which is said to be a *stand-alone storage*, or the tree is split into different memories which is said to be an *hybrid storage*. It is possible to store some strategic nodes and to recompute the corresponding subtrees when needed.

Nowadays RAM modules can store a large amount of data; typically between 4 GB and 16 GB. In addition to current computation data, the RAM could store a part of the Merkle tree. In the following, an estimation of the update/verify times of a sector and the storage cost in the different memories is given depending on the storage configurations. Some of the following solutions are suitable for disk protection but once again it depends on the device: for a laptop, the disk size is in the range of gigabyte or terabyte, the RAM size is rarely larger than 32 GB, and is in average around 8 GB. The secure memory is usually only a few kilobytes. The **access latencies** of the different memories are also a parameter to take into account.

Cost	$S^{\emptyset,\emptyset}_{\mathsf{tree}}$	$S^{\emptyset,\emptyset}_{\mathsf{tags}}$	$S^{\emptyset,\emptyset}_{\mathsf{root}}$	$S^{\mathsf{tree},\emptyset}_{\mathsf{root}}$	$S^{\emptyset,\mathsf{tree}}_{\mathsf{root}}$	$S^{\mathsf{tree},\mathsf{tree}}_{\mathsf{root}}$
Up/Ver	n'	1	$2^{n'}-1$	n'	n'	n'
RAM	-	-	-	$2^{n'}-1$	-	$2^{n'}-1$
Disk	0	0	0	0	$2^{n'}-1$	$2^{n'}-1$
SM	$2^{n'}-1$	$2^{n'}$	1	1	1	1

Table 2: Estimation of Stand-alone storage of Merkle tree with n actual data sectors where $n = 2^{n'}$. Up/Ver stands for update and verify cost.

Stand-alone Storage:.

Table 2 gives 5 possibilities for Merkle tree storage where it is entirely stored in the SM, the RAM, or the disk. We denote them $S^{b,c}_{a}$ where a denotes what is stored in the SM, b what is stored in the RAM and c what is stored in the disk.

- $S^{\emptyset,\emptyset}_{\mathsf{tree}}$: The entire Merkle tree is stored in the SM and for each read and write operation, the update/verify operation on the Merkle tree costs n' calls f_K. This solution does not use disk and RAM storage but the update/verification has a reasonable cost. If it is possible to store $2^{n'}-1$ tags then it should be possible to store $2^{n'}$ which corresponds to solution $S^{\emptyset,\emptyset}_{\mathsf{tags}}$. In this case, all the local tags can be stored in the SM and as it is assumed that this memory is tamper-proof, the global authentication scheme is no longer needed: the local

tags are stored directly in SM. Then the Merkle tree is no longer needed. If the SM is big enough, this solution should be considered otherwise the following possibilities, where only a small value (the root) is stored in SM, are more suitable.

- $S^{\emptyset,\emptyset}_{\mathsf{root}}$: The only value stored is the Merkle tree root and it is stored in the SM. The storage on the disk, the SM, and the RAM is minimized but the update/verify cost is the worst: the entire Merkle tree has to be recomputed for each update/verify operation. For this reason, this solution seems to be unsuitable for disks.

- $S^{\mathsf{tree},\emptyset}_{\mathsf{root}}$: This solution considers a context where the RAM is big enough to store the entire Merkle tree and operates in rated conditions. The access time to Merkle tree nodes in the RAM is smaller than in the disk but the counterpart is that the Merkle tree will not be maintained when the system is switched off. It has to be recomputed each time the system is switched on. This solution could be interesting if saving disk storage is a priority. In fact, it stores only the root in SM and no additional data in the disk. The update/verify operation cost is reasonable but a re-computation delay due to the volatility of the RAM is added.

- $S^{\emptyset,\mathsf{tree}}_{\mathsf{root}}$: This solution has the same settings as $S^{\mathsf{tree},\emptyset}_{\mathsf{root}}$ except that the Merkle tree is not stored in RAM but in the disk. It has the advantage to remove the re-computation delay but this time the additional storage in the disk is the entire Merkle tree size. Moreover, the time to read and write a node will be larger as it is stored in the disk. Unsurprisingly, this solution could be implemented in an optimized manner by some developers as solution $S^{\mathsf{tree},\mathsf{tree}}_{\mathsf{root}}$.

- $S^{\mathsf{tree},\mathsf{tree}}_{\mathsf{root}}$: This solution combines solutions $S^{\mathsf{tree},\emptyset}_{\mathsf{root}}$ and $S^{\emptyset,\mathsf{tree}}_{\mathsf{root}}$: the tree is stored in the disk and also in the RAM. Each time the device is switched on, the Merkle tree is copied from the disk to the RAM which avoids the re-computation delay of solution $S^{\mathsf{tree},\emptyset}_{\mathsf{root}}$. This solution is the most efficient from a speed point of view.

For a reasonable amount of data, solution $S^{\emptyset,\emptyset}_{\mathsf{tags}}$ can be considered for cloud storage as few gigabytes can be stored on the client-side but it seems unpractical for a stand-alone disk. Solution $S^{\mathsf{tree},\mathsf{tree}}_{\mathsf{root}}$ stores only the root in the SM and is the quickest solution described above which makes it the most suitable for a stand-alone disk.

Discussion. Table 3 gives the Merkle tree storage cost in RAM, in the disk, and in the SM with the same settings as in section 4.1. As explained in the analysis of $S^{\emptyset,\emptyset}_{\mathsf{root}}$, the update and verify operations cost a large amount of calls to the function f_K. The same problem will emerge each time the device is switched on for solution $S^{\mathsf{tree},\emptyset}_{\mathsf{root}}$ due to storage in RAM. Solutions $S^{\emptyset,\emptyset}_{\mathsf{tree}}$ and $S^{\emptyset,\emptyset}_{\mathsf{tags}}$ are not suitable for the rather small-sized SM we discussed in section 4.2. Unsurprisingly, solution $S^{\mathsf{tree},\mathsf{tree}}_{\mathsf{root}}$ seems to be the most suitable for disk protection. In these settings, 8 GB are needed to store local tags and 8 GB for global storage then for solutions $S^{\emptyset,\mathsf{tree}}_{\mathsf{root}}$ and $S^{\mathsf{tree},\mathsf{tree}}_{\mathsf{root}}$, the FADE mechanism costs 1,5% of the entire disk.

Cost	$S^{\emptyset,\emptyset}_{\mathsf{tree}}$	$S^{\emptyset,\emptyset}_{\mathsf{tags}}$	$S^{\emptyset,\emptyset}_{\mathsf{root}}$	$S^{\mathsf{tree},\emptyset}_{\mathsf{root}}$	$S^{\emptyset,\mathsf{tree}}_{\mathsf{root}}$	$S^{\mathsf{tree},\mathsf{tree}}_{\mathsf{root}}$
Up/Ver	28	1	268435455	28	28	28
RAM	-	-	-	8 GB	-	8 GB
Disk	0	0	0	0	8 GB	8 GB
SM	8 GB	8 GB	32 B	32 B	32 B	32 B

Table 3: Estimation of Stand-alone storage of Merkle tree where $n = 2^{28}$ and $L = 32$ Bytes

Hybrid Storage:. Table 4 presents variants where the Merkle tree nodes of level m, where $m \leq n'$, are stored. We denote them $H^{b,c}_a$ where a denotes what is stored in the SM, b what is stored in the RAM and c what is stored in the disk.

The performance depends on the choice of m (see Tab. 4). The optimum value of m depends on many factors: the disk, the SM and the RAM sizes, the CPU, the cryptographic algorithms etc. Due to this variety of factors, this

Cost	$H^{\emptyset,\mathsf{top}}_{\mathsf{root}}$	$H^{\emptyset,\emptyset}_{\mathsf{level\text{-}m}}$	$H^{\emptyset,\mathsf{subtrees}}_{\mathsf{level\text{-}m}}$	$H^{\mathsf{subtrees},\emptyset}_{\mathsf{level\text{-}m}}$	$H^{\mathsf{subtrees},\mathsf{top}}_{\mathsf{root}}$
Up/Ver	$2^{n'-m} - 1 + m$	$2^{n'-m} - 1$	$n' - m$	$n' - m$	n'
RAM	-	-	$2^{n'} - 2^{m+1}$	-	$2^{n'} - 2^{m+1}$
Disk	$2^{m+1} - 1$	0	0	$2^{n'} - 2^{m+1}$	$2^{m+1} - 1$
SM	1	2^m	2^m	2^m	1

Table 4: Estimation of Hybrid storage of Merkle Tree with n actual data sectors where $n = 2^{n'}$ and $m \leq n'$. Up/Ver stands for update and verify cost.

analysis is only theoretical: it provides insights about the main tendencies of each implementation strategy. It would be nonetheless interesting to test these solutions in different devices to adjust the level m and compare their real performances.

- $H_{\mathsf{root}}^{\emptyset,\mathsf{top}}$: This solution minimizes the storage in SM by storing the Merkle tree root only. The top of the Merkle tree, from the root to level m, is stored in the disk. For each update/verify operation, recomputing the corresponding subtree costs $2^m - 1$ calls to f_K and the update/verify of the top of the tree costs m calls to f_K. If a verification fails, it will not be possible to find exactly which sector was tampered with, in the best case, only the node at level m can be given as an information to the user.

- $H_{\mathsf{level\text{-}m}}^{\emptyset,\emptyset}$: The nodes at the level m are stored in the SM. Here, these nodes do not need data authenticity check, they are assumed to be authentic as they are stored in the SM. For each update, the needed subtree is recomputed which costs $2^{n'-m} - 1$ calls to f_K. Here, the SM has to store $2^m L$ bits. Depending on the value m and the SM size, it might be possible for a disk. For instance, if the SM can store 1 MB and the $L = 256$ bits then the maximum value of m is 12.

- $H_{\mathsf{level\text{-}m}}^{\emptyset,\mathsf{subtrees}}$: The nodes of the level m are stored in the SM and the m subtrees are stored in the RAM. Each update operation costs going through the subtree, which requires $n' - m$ calls. Storage in the RAM involves a re-computation time when the device is switched on.

- $H_{\mathsf{level\text{-}m}}^{\mathsf{subtrees},\emptyset}$: The nodes of the level m are stored in the SM and all the subtrees are stored in the disk. Unlike, the previous solution, accessing tree nodes in the disk takes more time.

- $H_{\mathsf{root}}^{\mathsf{subtrees},\mathsf{top}}$: The Merkle tree is split between all the memories: the root in SM, the top in the disk and the corresponding subtrees in the RAM. Because of the RAM storage, a delay is added each time the device is switched off. The update/verify time is acceptable.

Solution $H_{\mathsf{root}}^{\emptyset,\mathsf{top}}$ seems to be the most suitable for disk encryption, the SM storage is reduced to the Merkle tree root, the upper part is stored in the disk and some layers have to be recomputed. These solutions seem to remain acceptable for disk encryption as long as the value m gives decent storage

Cost	$H_{\mathsf{root}}^{\emptyset,\mathsf{top}}$	$H_{\mathsf{level\text{-}m}}^{\emptyset,\emptyset}$	$H_{\mathsf{level\text{-}m}}^{\emptyset,\mathsf{subtrees}}$	$H_{\mathsf{level\text{-}m}}^{\mathsf{subtrees},\emptyset}$	$H_{\mathsf{root}}^{\mathsf{subtrees},\mathsf{top}}$
Up/Ver	8207	8191	13	13	28
RAM	-	-	8 GB	-	8 GB
Disk	2 MB	0	0	8 GB	2 MB
SM	32 B	1 MB	1 MB	1 MB	32 B

Table 5: Evaluation of the Hybrid storage of Merkle tree with $n = 2^{28}$, $m = 15$, $L = 32$ Bytes.

cost in the SM and also in the RAM. Solutions, where update/verify time is minimized, seem to be better as this latency has a direct impact on the disk performance.

Discussion. Table 5 gives an estimation for the value $m = 15$ with the same settings than in the discussion for stand-alone storage. The value m was chosen to have a maximum storage in SM equal to 1 MB. We can note that the update and verification time is shorter for solutions $H_{\mathsf{level\text{-}m}}^{\emptyset,\mathsf{subtrees}}$ and $H_{\mathsf{level\text{-}m}}^{\mathsf{subtrees},\emptyset}$ whereas it is quite long for solutions $H_{\mathsf{root}}^{\emptyset,\mathsf{top}}$ and $H_{\mathsf{level\text{-}m}}^{\emptyset,\emptyset}$. Storage in RAM in solutions $H_{\mathsf{level\text{-}m}}^{\emptyset,\mathsf{subtrees}}$ and $H_{\mathsf{root}}^{\mathsf{subtrees},\mathsf{top}}$ adds a re-computation delay that should be limited as much as possible. Solution $H_{\mathsf{level\text{-}m}}^{\mathsf{subtrees},\emptyset}$ seems to be the best solution if the SM can store 1 MB otherwise $H_{\mathsf{root}}^{\mathsf{subtrees},\mathsf{top}}$ has to be considered. The storage cost of the entire FADE mechanism for $H_{\mathsf{level\text{-}m}}^{\mathsf{subtrees},\emptyset}$ is close to solutions $S_{\mathsf{root}}^{\emptyset,\mathsf{tree}}$ and $S_{\mathsf{root}}^{\mathsf{tree},\mathsf{tree}}$, it represents about 1,5% of the disk but the update and verify time is better.

Cost	$H_{\mathsf{root}}^{\emptyset,\mathsf{top}}$	$H_{\mathsf{level\text{-}m}}^{\emptyset,\emptyset}$	$H_{\mathsf{level\text{-}m}}^{\emptyset,\mathsf{subtrees}}$	$H_{\mathsf{level\text{-}m}}^{\mathsf{subtrees},\emptyset}$	$H_{\mathsf{root}}^{\mathsf{subtrees},\mathsf{top}}$
Up/Ver	32	7	3	3	28
RAM	-	-	6 GB	-	6 GB
Disk	2 GB	0	0	6 GB	2 GB
SM	32 B	1 GB	1 GB	1 GB	32 B

Table 6: Evaluation of the Hybrid storage of Merkle tree with $n = 2^{28}$, $m = 25$, $L = 32$ Bytes.

In section 4.2, the SM was limited to a few megabytes and corresponds to realistic figures of the current state-of-the-art of Secure Elements. Other technologies embedding more secure memory have raised industry attention in

the last years. Examples of such solutions are those belonging to the Trusted Execution Environment (TEE) ecosystem, with TrustZone, Intel SGX, and so on. In the TEE paradigm, a Secure Element is "emulated" in the form of a sandboxed execution mode (the "Secure World") of a general-purpose processor, yielding more computing power and storage space when compared to classical Secure Elements. Even though the storage space of such a solution is usually shared with the so-called "Non Secure World", more and more SoC vendors embed non-volatile memory dedicated to the TEE with hardware security isolation insurance. In such components, the SM considered in our models could reach hundreds of megabytes to gigabytes of internal storage, and Table 6 gives figures in a case where 1 gigabyte can be stored in the SM. Software implementations of FADE using TEE as an SM have been proposed in [28]: the Secure Block Device Library (SBDL) uses CMAC and Merkle trees to bring confidentiality and integrity to Trusted Applications data at-rest storage. Temporal integrity is ensured whenever the underlying SoC provides physical secure storage with non-tampering properties: the Merkle tree root, as well as the master encryption key, are then stored inside it.
In Tables 2, 3, 4, 5 and 6, the path of the Merkle tree (or a part of it for hybrid storage) stored explicitly in RAM is updated and verified for each read and write operation. As attacks in RAM are out of scope, the verification path can be omitted to speed up the read operation of a sector. Doing so for the write operations is more tedious: after some number of writes, several paths in the tree are updated. These updates lead to refreshing the tree in SM (which can be reduced to the root or more nodes depending on the chosen solution) just before powering down the device. In case of failure (device out of battery for instance), updating the tree in SM is essential, otherwise, the integrity check will fail for all the sectors lately updated. Hence, this optimization seems suitable for SEDs embedding emergency batteries, leaving enough time to perform such updates.

5. Conclusion

Today, products implementing disk encryption are widespread. Most of them use the classical FDE mode of operation, namely XTS. Adding efficient data authenticity in the strong model (FADE) with tight constraints is an important yet unsolved practical challenge. Developers and researchers have begun to address it since a few years with preliminary solutions (see figure

Protection	Name	Description
FDE	Bitlocker[20]	Elephant diffuser AES-CBC-ESSIV encryption
	Veracrypt[3]	AES-XTS (or other encryption algorithms with XTS mode)
	dm-crypt[1]	AEC-CBC-ESSIV or AES-XTS
	FileVault[15]	AES-XTS
	CRYHOD[44]	Proprietary
ADE	GELI[17]	Per sector metadata Data+tag in 8 sectors (4096 bytes) 8+1 native sectors used
	dm-integrity[11]	Arbitrary-sized metadata per sector (tweakable)
FADE	SBDL[28]	Configurable AEAD+CMAC+Merkle tree Trusted storage (in TEE) usage for root hash and key
	dm-x [13]	Configurable AEAD+Merkle tree Trusted storage (trusted VM, TPM)
	StrongBox [19]	Configurable AEAD+Merkle tree Trusted storage (TEE)

Table 7: Implementations with their security level

7). The global authentication mechanism is new from a cryptographic perspective, and the question is whether we can do better than Merkle trees. From an architectural point of view, the introduction of a secure memory in laptops as well as in smartphones seems to be a strong requirement to have global authentication. This is out of scope in this paper, but there is a crucial question for the deployment of FADE solutions: when secure memory are available, are they accessible to standard FADE software for embedding authentication data (such as a root of a Merkle tree), or are they kept private to manufacturers proprietary implementations? The next challenging step would be to implement FADE mechanisms with the Merkle tree in different devices with different configurations: stand-alone storage, hybrid storage, and in this case we should find the trade-off for the value m.

Appendix A. Incremental MACs

Here is the description of two incremental MACs : the Xor-MAC and the Xor-Scheme (the fixed version). These constructions are of interest due to their incremental properties: once computed, a tag can be updated efficiently.

NOTATIONS. we let $\{0,1\}^*$ denote the set of finite binary strings.
A pseudorandom function family (PRF) is a function family which behaves like a random function for a computationally bounded adversary (e.g., input and output behavior).
The description of cryptographic scheme CS is given as a tuple

$$\mathsf{CS} := (\mathsf{A}, \mathsf{B}, \ldots, \mathsf{C})$$

where each element can be a parameter of the scheme or an associated algorithm. The notation $\mathsf{CS.A}$ refers to the parameter or algorithm A of the scheme CS.

Appendix A.1. Description of the Xor-MAC

There are two versions of the XMAC construction: the random based one denoted XMAC-R and the nonce (or counter) based one denoted XMAC-C depending on how the value v is defined.
Then XMAC is defined as follows

$$\mathsf{XMAC\text{-}C/R} = (\mathsf{KS}, \mathsf{BS}, \mathsf{DS}, \mathsf{NS/RS}, \mathsf{kg}, \mathsf{tag}, \mathsf{upd}, \mathsf{ver}).$$

- The XMAC construction is based on a pseudorandom function family $\mathsf{F} = (\mathsf{KS}, \mathsf{Dom}, \mathsf{Rng}, \mathsf{eval})$ such that $\mathsf{F} : \mathsf{F.KS} \times \mathsf{F.Dom} \to \mathsf{F.Rng}$. It follows that $\mathsf{F.Dom} = \{1\} \times \{0,1\}^p \times \mathsf{XMAC.BS}$ where $\{0,1\}^p$ is the position space and $\mathsf{F.Rng} = \{0,1\}^{t\ell}$ where $t\ell$ is the tag size.

- The **key generation algorithm** $\mathsf{XMAC.kg}$ is a (probabilistic) algorithm that takes no input and returns a key $K \in \mathsf{XMAC.KS}$ such that $\mathsf{XMAC.KS} = \mathsf{F.KS}$.

- The **tagging algorithm** $\mathsf{XMAC.tag}^1$ takes as inputs the key $K \in \mathsf{XMAC.KS}$, a document $m \in \mathsf{XMAC.DS}$ and outputs a tag t. A value v prepended by the bit 0, such that $v \in \mathsf{XMACR.RS}$ for the randomized version and such that $v \in \mathsf{XMAC.NS}$ for the counter-based version, is given as input to F and each document block $m_i \in \mathsf{XMAC.BS}$ prepended

by $1||i$ is given as input to the pseudorandom function F_K then the sum of the corresponding outputs τ and the value v is returned as the tag $t = (v, \tau)$.

- The **Verification algorithm** XMAC.ver takes as inputs the key $K \in$ XMAC.KS, the document m and the tag $t = (v, \tau)$. It re-computes the value τ from the inputs m and v then it returns true if this value is equal to the input t and false otherwise.

- The **Update algorithm** XMAC.upd takes as inputs the key $K \in$ XMAC.KS, the document m, the operation $op \in$ OpCodes such that OpCodes $= \{\mathtt{R}, \mathtt{I}, \mathtt{D}\}$, the set of argument arg and the tag t. The argument arg is composed by the position i where the block value has to be inserted, deleted or replaced and the new document block $x \in$ XMAC.BS for the insert and delete operations or ε if it is a delete operation: $\text{arg} = \langle i, x \rangle$. The I and D operations can be performed only for the last position: as each document block is processed with it block position, it is not possible to insert or delete a block efficiently.

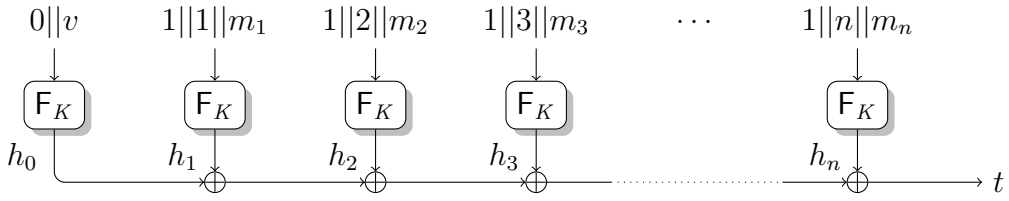

Figure A.10: Description of the XMAC where v is a random value or a nonce.

Appendix A.2. Description of the Xor-Scheme

The XS is a probabilistic scheme, we have a random space XS.RS from which the random data blocks are randomly sampled. It is defined as follows:

$$\mathsf{XS} = (\mathsf{KS}, \mathsf{BS}, \mathsf{DS}, \mathsf{RS}, \mathsf{kg}, \mathsf{tag}, \mathsf{upd}, \mathsf{ver}).$$

- XS is based on a pseudorandom function family $\mathsf{F} = (\mathsf{KS}, \mathsf{Dom}, \mathsf{Rng}, \mathsf{eval})$ such that $\mathsf{F} : \mathsf{F.KS} \times \mathsf{F.Dom} \rightarrow \mathsf{F.Rng}$. It follows that $\mathsf{F.Dom} = \mathsf{XS.RS}^2 \times \mathsf{XS.BS}^2$ and $\mathsf{F.Rng} = \{0,1\}^{t\ell}$ where $t\ell$ is the tag size.

- The **key generation algorithm** $\mathsf{XS.kg}$ is a probabilistic algorithm that takes no input and returns a key $K \in \mathsf{XS.KS}$ such that $\mathsf{XS.KS} = \mathsf{F.KS}^2$.

- The **tagging algorithm** $\mathsf{XS.tag}$ takes as inputs the key $K \in \mathsf{XS.KS}$, a document $m \in \mathsf{XS.DS}$ and outputs a tag $t := (r, t)$. Similarly to the XS scheme, for each document block $m_i \in \mathsf{XS.BS}$, a random block value $r_i \in \mathsf{XS.RS}$ is randomly sampled and its bit length is denoted $\mathsf{XS.rl}$. The concatenation of these values is denoted $R_i := m_i||r_i$. Each couple (R_{i-1}, R_i) is processed by the function F_{K_1}. The concatenation of the last value R_n and the number of blocks n encoded as an ℓ-bit block is processed by a pseudorandom permutation function F_{K_2} with a different key K_2. Then the output values denoted h_i are bitwise Xored to give the value t.

- The **Update algorithm** $\mathsf{XS.upd}$ takes as inputs the key $K \in \mathsf{XS.KS}$, the document the operation $op \in \mathsf{OpCodes}$ where $\mathsf{OpCodes} = \{\mathtt{I}, \mathtt{D}\}$, the set of argument arg and the tag t. The argument arg is composed by the position i where the block value has to be inserted or deleted and the new document block $x \in \mathsf{XS.BS}$ to insert or ε if it is a delete operation: $\text{arg} = \langle i, x \rangle$.

- The **Verification algorithm** $\mathsf{XS.ver}$ takes as inputs the key $K \in \mathsf{XS.KS}$, the document m and the tag $t := (r, t)$. It re-computes the value τ from the inputs r and m. It returns true if this value is equal to the input t and false otherwise.

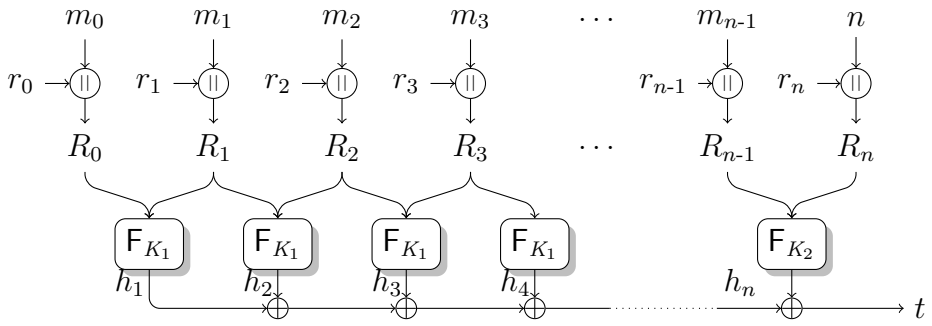

Figure A.11: Description of the fixed Xor-Scheme

References

[1] *dm-crypt: Linux device-mapper crypto target*, `https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt`, 2020.

[2] *dm-verity: Linux device-mapper block integrity checking target*, `https://source.android.com/security/verifiedboot/dm-verity`, 2020.

[3] *Veracrypt*, `https://www.veracrypt.fr/en/Home.html`, 2020.

[4] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder, *An analysis of data corruption in the storage stack*, TOS **4** (2008), no. 3, 8:1–8:28.

[5] Hagai Bar-El, *Security implications of hardware vs. software cryptographic modules*, Discretix White Paper (2002).

[6] Mihir Bellare, Oded Goldreich, and Shafi Goldwasser, *Incremental cryptography and application to virus protection*, 27th Annual ACM Symposium on Theory of Computing (Las Vegas, NV, USA), ACM Press, May 29 – June 1, 1995, pp. 45–56.

[7] Mihir Bellare, Roch Guérin, and Phillip Rogaway, *XOR MACs: New methods for message authentication using finite pseudorandom functions*, Advances in Cryptology – CRYPTO'95 (Santa Barbara, CA, USA) (Don Coppersmith, ed.), Lecture Notes in Computer Science, vol. 963, Springer, Heidelberg, Germany, August 27–31, 1995, pp. 15–28.

[8] Mihir Bellare and Chanathip Namprempre, *Authenticated encryption: Relations among notions and analysis of the generic composition paradigm*, Advances in Cryptology – ASIACRYPT 2000 (Kyoto, Japan) (Tatsuaki Okamoto, ed.), Lecture Notes in Computer Science, vol. 1976, Springer, Heidelberg, Germany, December 3–7, 2000, pp. 531–545.

[9] Daniel Bernstein, *Caesar finalists*, 2018.

[10] Milan Broz, *Authenticated and resilient disk encryption*, Ph.D. thesis, Masaryk University, 2018.

[11] Milan Broz, Mikulás Patocka, and Vashek Matyás, *Practical cryptographic data integrity protection with full disk encryption*, ICT Systems Security and Privacy Protection, 2018, pp. 79–93.

[12] Kevin R. B. Butler, Stephen E. McLaughlin, and Patrick D. McDaniel, *Disk-enabled authenticated encryption*, IEEE 26th Symposium on Mass Storage Systems and Technologies, MSST 2012, Lake Tahoe, Nevada, USA, May 3-7, 2010, 2010, pp. 1–6.

[13] Anrin Chakraborti, Bhushan Jain, Jan Kasiak, Tao Zhang, Donald E. Porter, and Radu Sion, *dm-x: Protecting volume-level integrity for cloud volumes and local block devices*, Proceedings of the 8th Asia-Pacific Workshop on Systems, Mumbai, India, September 2, 2017, 2017, pp. 16:1–16:7.

[14] Debrup Chakraborty, Cuauhtemoc Mancillas-López, and Palash Sarkar, *Disk encryption: Do we need to preserve length?*, Cryptology ePrint Archive, Report 2015/594, 2015, `https://eprint.iacr.org/2015/594`.

[15] Omar Choudary, Felix Gröbert, and Joachim Metz, *Security analysis and decryption of filevault 2*, Advances in Digital Forensics IX - 9th IFIP WG 11.9 International Conference on Digital Forensics, Orlando, FL, USA, January 28-30, 2013, Revised Selected Papers, 2013, pp. 349–363.

[16] Paul Crowley and Eric Biggers, *Adiantum: length-preserving encryption for entry-level processors*, IACR Transactions on Symmetric Cryptology **2018** (2018), no. 4, 39–61.

[17] Pawel Jakub Dawidek, *GELI(8)*, `https://www.freebsd.org/cgi/man.cgi?query=geli`, 2014.

[18] Dell Computer Corporation, Hewlett Packard Corporation,Hitachi Global Storage Technologies, Inc., Intel Corporation, Maxim Integrated Products, Seagate Technology, Western Digital Corporation , *Serial ATA International Organization, Serial ATA Revision 3.0*, Tech. report, 2009.

[19] Bernard Dickens III, Haryadi S Gunawi, Ariel J Feldman, and Henry Hoffmann, *Strongbox: Confidentiality, integrity, and performance using stream ciphers for full drive encryption*, Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, 2018.

[20] Niels Ferguson, *AES-CBC + Elephant diffuser: A Disk Encryption Algorithm for Windows Vista*, `http://www.microsoft.com/en-us/download/details.aspx?id=13866`, 2006, p. 22.

[21] International Organization for Standardization, *Information technology – Security techniques – Authenticated encryption*, Standard, Geneva, CH, 2009.

[22] ———, *Information technology – Security techniques – Message Authentication Codes (MACs) – Part 1: Mechanisms using a block cipher*, Standard, Geneva, CH, 2011.

[23] ———, *Information technology – Security techniques – Message Authentication Codes (MACs) – Part 2: Mechanisms using a dedicated hash-function*, Standard, Geneva, CH, 2011.

[24] Clemens Fruhwirth, *New methods in hard disk encryption*, Master's thesis, Vienna University of Technology, 2005.

[25] Aaron Fujimoto, Peter Peterson, and Peter L. Reiher, *Comparing the power of full disk encryption alternatives*, IGCC, IEEE Computer Society, 2012, pp. 1–6.

[26] Kristian Gjøsteen, *Security notions for disk encryption*, ESORICS 2005: 10th European Symposium on Research in Computer Security (Milan, Italy) (Sabrina De Capitani di Vimercati, Paul F. Syverson, and Dieter Gollmann, eds.), Lecture Notes in Computer Science, vol. 3679, Springer, Heidelberg, Germany, September 12–14, 2005, pp. 455–474.

[27] Shai Halevi and Phillip Rogaway, *A parallelizable enciphering mode*, Topics in Cryptology – CT-RSA 2004 (San Francisco, CA, USA) (Tatsuaki Okamoto, ed.), Lecture Notes in Computer Science, vol. 2964, Springer, Heidelberg, Germany, February 23–27, 2004, pp. 292–304.

[28] Daniel Hein, Johannes Winter, and Andreas Fitzek, *Secure Block Device–Secure, Flexible, and Efficient Data Storage for ARM TrustZone Systems*, Trustcom/BigDataSE/ISPA, 2015 IEEE, vol. 1, IEEE, 2015, pp. 222–229.

[29] Alexander Heitzmann, Bernardo Palazzi, Charalampos Papamanthou, and Roberto Tamassia, *Efficient integrity checking of untrusted network*

storage, Proceedings of the 2008 ACM Workshop On Storage Security And Survivability, StorageSS 2008, Alexandria, VA, USA, October 31, 2008, 2008, pp. 43–54.

[30] Fangyong Hou, Dawu Gu, Nong Xiao, Fang Liu, and Hongjun He, *Performance and consistency improvements of hash tree based disk storage protection*, International Conference on Networking, Architecture, and Storage, NAS 2009, 9-11 July 2009, Zhang Jia Jie, Hunan, China, 2009, pp. 51–56.

[31] IEEE, *IEEE Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices*, IEEE Std 1619-2007 (2008), 1–32.

[32] ———, *IEEE Standard for for Wide-Block Encryption for Shared Storage Media*, IEEE Std P1619.11619-2010 (2011).

[33] ———, *IEEE Standard for Authenticated Encryption with Length Expansion for Storage Devices*, IEEE Std 1619.1-2018 (2018), 1–45.

[34] Kevin Igoe and Jerome Solinas, *Aes galois counter mode for the secure shell transport layer protocol*, Tech. report, 2009.

[35] Louiza Khati, Nicky Mouha, and Damien Vergnaud, *Full disk encryption: Bridging theory and practice*, Topics in Cryptology – CT-RSA 2017 (San Francisco, CA, USA) (Helena Handschuh, ed.), Lecture Notes in Computer Science, vol. 10159, Springer, Heidelberg, Germany, February 14–17, 2017, pp. 241–257.

[36] Louiza Khati and Damien Vergnaud, *Analysis and Improvement of an Authentication Scheme in Incremental Cryptography*, Selected Areas in Cryptography - SAC 2018 (Calgary, Canada), August 2018.

[37] Tadayoshi Kohno, John Viega, and Doug Whiting, *CWC: A high-performance conventional authenticated encryption mode*, Fast Software Encryption – FSE 2004 (New Delhi, India) (Bimal K. Roy and Willi Meier, eds.), Lecture Notes in Computer Science, vol. 3017, Springer, Heidelberg, Germany, February 5–7, 2004, pp. 408–426.

[38] Jakob Lell, *Practical malleability attack against cbc-encrypted luks partitions*, Blog (2013).

[39] David A. McGrew and Scott R. Fluhrer, *The security of the extended codebook (xcb) mode of operation*, SAC 2007: 14th Annual International Workshop on Selected Areas in Cryptography (Ottawa, Canada) (Carlisle M. Adams, Ali Miri, and Michael J. Wiener, eds.), Lecture Notes in Computer Science, vol. 4876, Springer, Heidelberg, Germany, August 16–17, 2007, pp. 311–327.

[40] Carlo Meijer and Bernard van Gastel, *Self-encrypting deception: weaknesses in the encryption of solid state drives (ssds)*, (2018).

[41] Tilo Müller, Tobias Latzo, and Felix C Freiling, *Self-encrypting disks pose self-decrypting risks*, the 29th Chaos Communinication Congress, 2012, pp. 1–10.

[42] Chanathip Namprempre, Phillip Rogaway, and Thomas Shrimpton, *Reconsidering generic composition*, Advances in Cryptology – EUROCRYPT 2014 (Copenhagen, Denmark) (Phong Q. Nguyen and Elisabeth Oswald, eds.), Lecture Notes in Computer Science, vol. 8441, Springer, Heidelberg, Germany, May 11–15, 2014, pp. 257–274.

[43] Alina Oprea and Michael K. Reiter, *Space-efficient block storage integrity*, Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA, 2005.

[44] Primx, *CRYHOD*, Tech. report, 2018.

[45] Eric Rescorla, *Tls elliptic curve cipher suites with sha-256/384 and aes galois counter mode (gcm)*, (2008).

[46] Phillip Rogaway, *Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC*, Advances in Cryptology – ASIACRYPT 2004 (Jeju Island, Korea) (Pil Joong Lee, ed.), Lecture Notes in Computer Science, vol. 3329, Springer, Heidelberg, Germany, December 5–9, 2004, pp. 16–31.

[47] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin, *Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly*, Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society, 2007.

[48] Gopalan Sivathanu, Charles P Wright, and Erez Zadok, *Ensuring data integrity in storage: Techniques and applications*, Proceedings of the 2005 ACM workshop on Storage security and survivability, 2005, pp. 26–36.

[49] STMicroelectronics, *High-speed secure MCU with 32-bit ARM® SecurCore® SC300TM CPU with SWP, ISO, SPI, I2C and high-density Flash memory*, Tech. report, February 2017.

[50] IBM Linux Technology, *ecryptfs*.

[51] Alexander Tereshkin, *Evil maid goes after pgp whole disk encryption*, Proceedings of the 3rd international conference on Security of information and networks, 2010, pp. 2–2.

[52] Marten Van Dijk, Jonathan Rhodes, Luis FG Sarmenta, and Srinivas Devadas, *Offline untrusted storage with immediate detection of forking and replay attacks*, Proceedings of the 2007 ACM workshop on Scalable trusted computing, 2007.

[53] Serge Vaudenay and Damian Vizár, *Under pressure: Security of caesar candidates beyond their guarantees*, Cryptology ePrint Archive, Report 2017/1147, 2017, `https://eprint.iacr.org/2017/1147`.

[54] John Viega and D McGrew, *The use of galois/counter mode (gcm) in ipsec encapsulating security payload (esp)*, Tech. report, 2005.