Masaryk University
Faculty of Informatics

Authenticated and Resilient Disk Encryption

Doctoral Thesis

Milan Brož

Brno, Fall 2018

Masaryk University
Faculty of Informatics

Authenticated and Resilient Disk Encryption

Doctoral Thesis

Milan Brož

Brno, Fall 2018

Declaration

Hereby I declare that this thesis is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Milan Brož

Advisor: prof. RNDr. Václav Matyáš, M.Sc., Ph.D.

Abstract

Full Disk Encryption (FDE) is a security feature that is present in many systems – from mobile devices to encrypted data in the cloud. Transparent on-the-fly encryption of storage devices operates directly on disk sectors layer and can be divided into the encryption of data by a symmetric encryption algorithm and key management. We analyzed both areas on real use cases and proposed new concepts that improve security considerably.

FDE was perceived as a security feature that cannot provide data integrity protection and focused on confidentiality only. To disprove this common claim, we implemented authenticated encryption that provides both confidentiality and integrity protection.

Our approach is based on per-sector authentication tags that are stored in configurable software-defined metadata areas on the same disk. On top of this metadata store, we implemented authenticated encryption that utilizes state-of-the-art authenticated encryption algorithms. Our open-source solution was accepted into the mainline Linux kernel and contains an extension of the dm-crypt disk encryption driver and our newly designed per-sector metadata store dm-integrity driver.

As our analysis revealed, the Linux kernel did not provide suitable authenticated ciphers for disk encryption applications. As a complement to our work (Master thesis supervised by the author of this thesis), implementations of AEGIS and MORUS authenticated ciphers, as CAESAR crypto competition finalists, were submitted to the mainline Linux kernel.

The second part of our work focused on key management that is an inseparable part of the security of any FDE system. Many disk encryption applications rely on a password-based key derivation. Processing a password is intentionally expensive, in order to increase attacker costs and limit possible dictionary and brute force attacks. With the introduction of memory-hard password hashing functions, we significantly improve resistance to such attacks on long-term FDE password-protected keys.

We analyzed Password Hashing Competition finalists and proposed requirements for a password-based key derivation function in the context of disk encryption.

We also designed and implemented a new version of the LUKS (Linux Unified Key Setup) key management scheme that uses the Argon2 memory-hard key-derivation function. This new format also supports configuration of authenticated disk encryption to deploy the results our work in a real environment easily.

Keywords

authenticated encryption, AEAD, Linux, disk encryption, FDE, dm-crypt, dm-integrity, Password Hashing Competition, PHC, storage security, Linux Unified Key Setup, LUKS, LUKS2, TrueCrypt, VeraCrypt

Contents

Preface

I started to contribute to the Linux disk encryption in the year 2008, almost ten years ago. Since then, the number of users dramatically increased and today almost every Linux distribution has a disk encryption option as one of the main configuration choices.

With the increasing popularity and more powerful systems, we started to face many problems. Users want encryption to be almost invisible and with a minimal impact to the storage performance. Use cases vary from high-end data servers to small and slow embedded devices. On the other hand, data stored on disks need to survive an increasing power of parallel and GPU dictionary attacks for years to come. Deployment of various optimizations, hardware support, and new cryptographic algorithms is a never-ending process.

The primary goal of this thesis came with an idea to alter the focus from the performance improvements to the research of new security challenges. As my experience shows, an active project improves performance in time anyway.

The initial focus of this thesis was on the disk encryption key management and research of new possibilities to increase resistance to dictionary and brute force attacks without any special hardware features. My goal was to investigate new memory-hard algorithms and use them for key derivation. This part is covered in Chapter 5 of this thesis.

During the review of my thesis proposal in 2015, Peter Schwabe (as a reviewer) came with an interesting idea to also focus on authenticated encryption and try to disprove a common myth that it cannot be used for disk encryption (due to the length-preserving nature and also for performance reasons). There have been several theoretic ideas already and some proofs-of-concept that needed hardware modifications.

Based on this challenge, I have decided to strictly focus on the goal to provide a practical implementation of authenticated encryption for generic storage already present in existing systems. I present our solution in Chapter 4.

The real security of a system is always based not only on security and proper use of underlying crypto primitives, but also on implementation details. A lab-only prototype cannot reveal essential issues of real systems.

Being in a unique position as a maintainer of a widely used open-source disk encryption project, I took the risk to contribute our ideas there. The time-consuming effort and coordination work to accept the real code was an academically invisible part of this PhD thesis, yet with a considerable practical impact.

Acknowledgements

I thank all users and contributors of cryptsetup and dm-crypt project that helped me to fix all my mistakes, provide necessary feedback and for trust in the software we are developing.

I would like to thank Ric Wheeler for his generous support of my work, my colleagues at Red Hat Czech for help to introduce our ideas into reality, Mikuláš Patočka for discussions and implementation of complex kernel code, Peter Schwabe for the idea to focus on the authenticated encryption and Arno Wagner for discussions and feedback.

Finally, thanks to my thesis advisor Vashek Matyáš. Without his support, this thesis would never be finished.

1 Introduction

As requirements for security and privacy grow, encryption becomes an essential part of most systems involved in the processing of sensitive data. With the availability of processors that can perform data encryption with throughput even higher than the throughput of high-end storage devices, the possibility to encrypt the whole storage device on-the-fly becomes a reality not even for mobile and embedded devices, but also for encryption of data stored in so-called cloud systems.

Full Disk Encryption (FDE) was one of the first concepts of encrypted storage. Based on a simple idea to encrypt disk sectors directly with symmetric block ciphers, disk encryption started to be widely used immediately once hardware managed to provide enough performance. FDE found a way directly to hardware (in the form of self-encrypting drives) or to software implementations like BitLocker in Windows OS, dm-crypt in Linux or FileVault in MacOS.

The idea of a transparent layer that is just inserted into existing data processing chain has some limitations. The encryption itself has no information about data context it encrypts. It means that empty sectors are encrypted the same way as sectors containing data. Another limitation is that the encryption key is (in most implementations) the same for all encrypted sectors. A user with the key can access the whole decrypted (plaintext) storage. Disk encryption is not thus intended for multi-user systems where we need some userspace isolation. While encryption can be implemented more effectively on higher layers (filesystem or application), the popularity of FDE is not decreasing.

Despite the apparent simplicity of FDE, evolution shows that even FDE can be implemented insecurely. Wrongly used encryption modes [1], insecure initialization vectors [2] or bad key management schemes are, unfortunately, widespread.

1.1 Motivation and Contribution

The motivation for our work is to propose an improvement for full disk encryption with the goal to increase security. Our focus area is application of recent development of cryptographic algorithms. Many concepts used in disk encryption are also directly applicable for filesystem encryption (instead of sectors we have filesystem blocks).

Based on experience with development and maintenance of Linux disk encryption, we strongly focus on practical usability of our proposals. Contributions of our work are summarized in the following list:

- Review and analysis (based on our independent implementation) of TrueCrypt and VeraCrypt disk encryption formats.
- Design and implementation of authenticated sector-based disk encryption that provides both confidentiality and data integrity protection under the defined threat model.
- Implementation of universal per-sector metadata store that can be used to build data integrity protection for the block device in Linux.
- Definition of requirements for password-based key derivation function for use in the disk encryption key management.
- Analysis of Password Hashing Competition candidates from the practical usability point of view and with the focus on our requirements.
- Extension of LUKS (Linux Unified Key Management) scheme of Argon2 memory-hard function.
- Use of AEGIS and MORUS authenticated encryption ciphers from the CAESAR cryptographic competition finalists for the Linux kernel cryptographic API.

1.2 Thesis Structure

The structure of the thesis is as follows:

- Chapter 2 provides an introduction to disk encryption and related technologies important for our research. It describes current state-of-the-art in disk encryption and background for issues we are focusing on in the following chapters.
- Chapter 3 follows on the previous chapter and, based on our independent implementation, analyses evolution and known issues in TrueCrypt (and VeraCrypt) disk encryption system. Our work in this area was published in [3]. Many of problems with TrueCrypt design were motivations for our research described in later chapters.
- Chapter 4 introduces a novel approach to implementation of full disk encryption system that provides not only confidentiality, but also data integrity protection. The idea to research a possible practical authenticated FDE is based on a Peter Schwabe's note in the thesis proposal review. Our solution is integrated in the mainline Linux kernel and available in many Linux distributions. Our results were published in [4, 5] and as an extended version in [6].

- Chapter 5 presents our effort to find a new password-based key derivation function to be later used in a new LUKS2 disk encryption key management system. This chapter is based on our practical evaluation of Password Hashing Competition [7] candidates. Our analysis is published in [8], based on our technical report [9].
- Chapter 6 introduces basic ideas behind evolution of the LUKS [10] key management system into a more extensible LUKS2 format and as such, it is a practical output of our work described in previous chapters.
- Chapter 7 concludes our research on practical disk encryption systems and mentions possible future directions in this research area.
- Appendix A includes the LUKS2 on-disk format definition, as used in the reference cryptsetup [11] implementation and presents a practical documentation guide for developers.
- Appendix B provides a list of academic publications co-authored by the author of this thesis.

2 Related Work

2.1 Full Disk Encryption

Full disk encryption (FDE) as transparent on-the-fly encryption of storage devices operates directly on the disk sectors. This means that it can be implemented inside an operating system (or even inside a storage device itself) without the need to adapt higher layers like filesystem or applications. FDE does not rule out encryption on higher layers. Independent encryption on various layers can still complement one another very well.

This layer separation is an important factor in designing secure and reliable storage systems, but also requires a very responsible design of all involved components. In theory, the layer separation is ideal and FDE could be completely transparent for the layers below and above. In reality, security and performance of an FDE system is directly influenced by the underlying storage architecture and design should take into account relevant physical and operational limits. This leads to many inevitable trade-off decisions (and often to design mistakes deflating security).

From a high level perspective, FDE just provides transparent access to a storage device using another virtual device layered on top of the encrypted one. The top level device contains clear data (plaintext) and applications can access this virtual device as a normal storage. The encryption layer then encrypts and decrypts sectors on-the-fly, so that the underlying device contains only encrypted data (ciphertext).

The whole underlying storage device is encrypted and if the device is properly wiped before initial use, the whole underlying disk should contain only encrypted data, statistically not distinguishable from a random noise.

Once a user activates the plaintext device (usually by providing a password or using some security token-based unlocking mechanism) the encryption layer is activated and plaintext device is made accessible to the system. Management of encryption keys, in particular the techniques for generating keys and storing them securely, is a critical part of the system.

2.2 Disk Encryption on Different Layers

FDE can be implemented in various layers, beginning with the hardware itself and ending with full software implementation running solely on the host processor. The problems to solve in various layers are usually quite similar, but implementations differ in utilizing various kinds of hardware.

The hardware level FDE is processed directly inside the drive. There is a lot of products using this kind of hardware FDE with various marketing names. Examples are Hitachi (*bulk data encryption*), Seagate (*full disk encryption*) or Toshiba (*self-encrypting drives*). There is also the *OPAL* storage specification [12] that tries to create a common standard for manufacturers of these drives.

Another hardware-based FDE is the *Chipset FDE*. This is a common stock drive without encryption capabilities, combined with a special hardware interface (bridge) placed between the drive and disk interface that provides encryption. The only difference from self-encrypting drives is that the encryption bridge is clearly (both logically and physically) separated from the drive. Example of such systems are almost all external USB drives with hardware-based encryption (because the USB Bridge is an additional interface, combining with encryption functionality seems to be a cost effective solution for manufacturers).

The approach we are mainly interested in is a *software-based FDE*. The encryption runs directly on the host system processor (either directly as a generic sub-procedure or utilizing special instructions like Intel AES-NI). There can be also a special crypto processing units integrated like IBM System z9 – CP Assist for Cryptographic functions (CPACF).

Schematic concept of areas related to FDE security is illustrated in Figure 2.1.

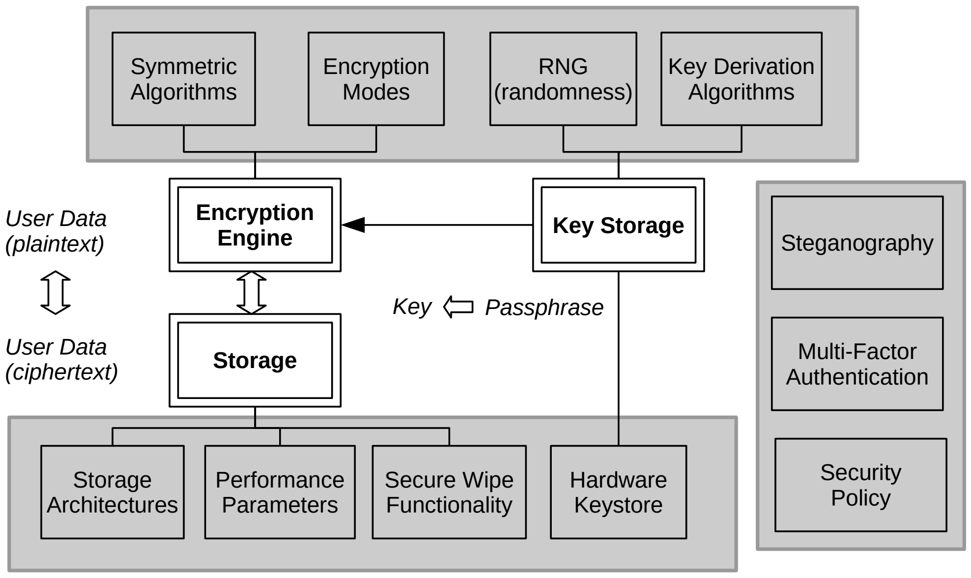

Figure 2.1: A schematic concept of FDE architectural design domains.

2.3 Common Threat Models

The common FDE threat model [13, 14, 15] covers the situation when the encrypted storage media content (containing ciphertext only) is accessible to an attacker while the storage device itself (or virtual image) is not in use (it is powered down or detached from the system in a secure state). The attacker can have direct physical access to the media with the encrypted content.

In other words, FDE protects data on storage from unauthorized access in situations when the device is lost, seized or when the attacker is able to remotely create an encrypted disk image snapshot (e.g., for a virtual machine in a datacenter or in a cloud environment). Sometimes the attacker can have access to some plaintext of stored data (like cached images from web pages or known files from a common OS installation).

In general, FDE does not fully protect in situation where the attacker has repeated access to the storage media and is able to create multiple snapshots of ciphertext in time. FDE also does not protect data when an authorized user is using the system (the system is powered on and the encryption layer is active), including various standby modes where the RAM contains plaintext data or even encryption key.

While most threat models are not explicitly covering situations where the device is in use, an FDE system design could prevent or complicate attacks for specific attack vectors outside our threat model boundary. An example is erasing the encryption key from RAM in the standby mode or implementations of encryption algorithms that do not store keys in memory [16].

Hardware-based FDE (either self-encrypted drives or chipset-based encryption) have other attack vectors [13, 17], yet the threat model is in principle the same.

While the threat models presented above provide *confidentiality* (plaintext is available only to a user with the secret key), they do not cover *integrity protection*. Integrity protection assures that all attempts to tamper the data by an attacker without secret key knowledge are detected (in other words it proves that data are not modified).

Disk encryption is widely used in corporations and the government sector, where it provides a basic protection to data theft for portable devices. Many of these users are strictly bound by various standards and requirements for cryptography use. Design of some disk encryption systems is done with these limitations in mind and sometimes leads to constructions that are used just to fulfill these requirements. The best known standard is the Federal Information Processing Standard (FIPS) 140 [18] that defines requirements for the US government cryptography use. The standard is published by the National Institute of Standards and Technology (NIST) that also drives a validation for cryptography products (modules). There is a requirement to use validated cryptography modules for unclassified information in US government and related organizations. In the European Union, there exists a similar recommendation for cryptography use, published by the European Union Agency for Network and Information Security [19].

2.4 Storage Architecture

The low-level underlying storage architecture develops in time and not only enables growth in storage sizes, but also introduces new issues and challenges. A *storage* here means a device intended for persistent data store. In real systems it is a disk, either rotational or flash memory based, available in an operating system as a block device. The atomic accessible unit of this device is indeed *block* (consisting of one or more low-level disk *sectors*), but to avoid confusion with an encryption block we will use sector as the description for an atomic unit of a block device.

In the future we can expect a massive use of persistent byte-addressable memory. Persistent byte-addressable storage has similar characteristics as a random access memory (RAM), just it is non-volatile (it keeps content after power-off). While it is not yet widely available, methods how to integrate it in operating systems are already discussed [20]. It is possible that there will be a transitioning period during which storage devices will be accessed through an emulation interface of a legacy block-based storage [21].

2.4.1 Sectors

Operating systems use usually 64-bit linear address for sectors on block devices. Common sector sizes are 512 bytes or 4096 bytes (enterprise storage can use up to 64-kilobyte sectors) [22].

Some enterprise drives intended for use in disk arrays provide a special space for parity checksums (typically 520-byte sector consisting of 512 bytes of data and 8 bytes for checksum) [23, 24].

2.4.2 Data Alignment

A good example of a performance-related attribute is the alignment of filesystem blocks to sectors (in flash memory storage even to internal memory pages). There can be multiple virtual storage layers so the alignment of all layers becomes a critical part of the storage performance (misalignment usually means increasing of input/output operations for the drive since it must split the writing area between one or more underlying sectors). If we add a transparent encryption layer between any of these layers, we should respect all restrictions to avoid a significant performance decrease.

2.4.3 Rotational Disks

Drives with rotational media (disks), called hard disk drives (HDDs), are still very common persistent storage devices. A modern HDD usually consists of several rapidly rotating platters and an arm with read-and-write heads. The data are stored by magnetizing ferromagnetic material on the platter surface.

Downside of rotational media is mainly caused by the mechanical nature of accessing sectors. Most notable is the *seek time* (time that an arm needs to move to a proper location on platters). There are also hybrid drives (drives combining rotational platters with a smaller cache on non-volatile flash memory technology).

The non-rotational memory can be presented outside as a separate drive (dual-drive) or it is integrated into an internal drive caching strategy, these drives are called solid-state hybrid drives (SSHDs).

2.4.4 Flash Memory Based Storage

The flash memory is a non-mechanical and non-volatile memory. The core of flash memory are cells (NOR or NAND gates) that can persistently store information. For single-level cell (SLC) it is one bit, multi-level cell (MLC) can recognize multiple levels of electrical charge and store more than one bit in one cell. Also, the types of flash chips differ according to the intended use (by accessing times, erase times), different kind of chips are used in memory cards and in fast SSDs.

Flash memory eliminates unpredictable seek time. The main downside of flash memory is the *rewriting procedure*. While cells can be read randomly (in byte or word granularity), they must be erased only by larger blocks. Rewriting a sector implies erasing a larger block followed by write of the changed sector (and all remaining data in the whole block).

Misalignment or write operations with internal flash block boundaries can significantly decrease data throughput.

Various optimizations can be used to mitigate a slow erase procedure. A drive usually contains an additional spare area to keep pre-erased blocks.

Also, there is an interface that informs the drive that particular sectors are no longer used and can be erased in advance. This operation is called TRIM command. Filesystems use TRIM to maintain a high drive performance during the whole life-time.

Unused sectors (after TRIM) contain detectable pattern. In combination with FDE, an attacker can detect unused space from ciphertext analysis. A pattern can also leak some additional information. Figure 2.2 demonstrates an example where we can recognize the filesystem type just from a ciphertext high level view [25].

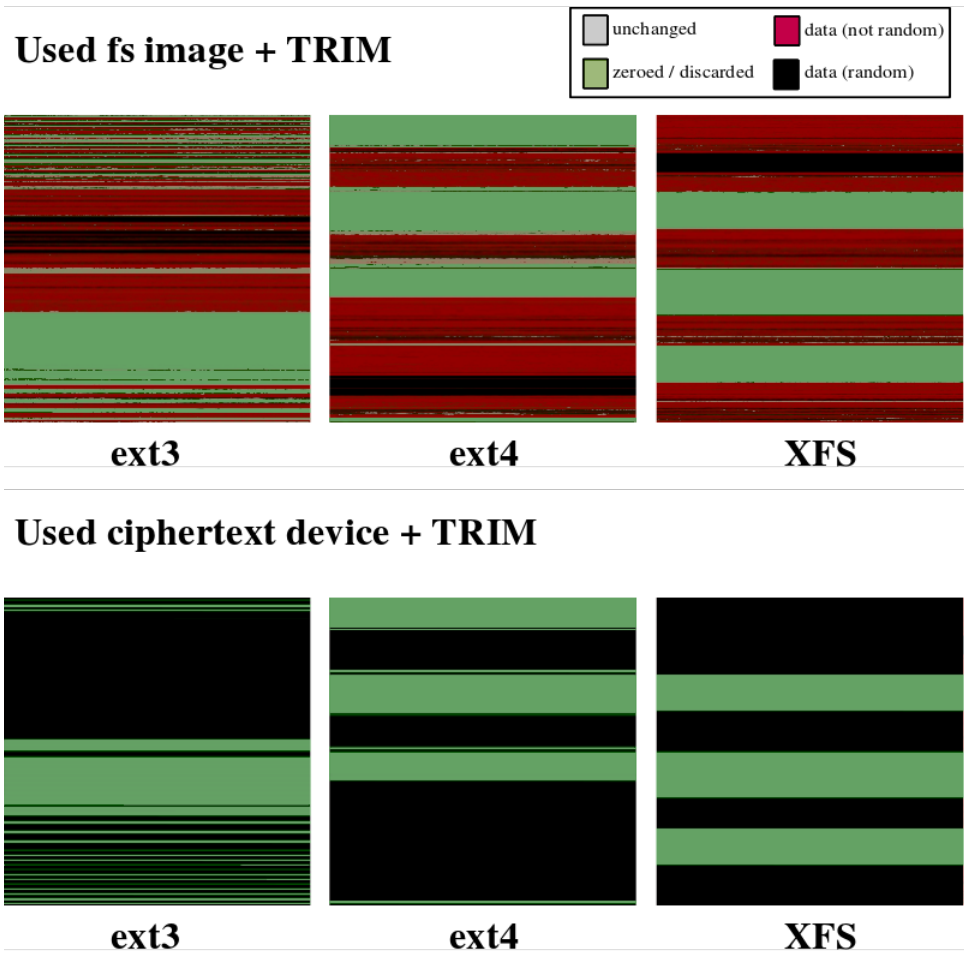

Figure 2.2: TRIM ciphertext device patterns [25].

The number of block erase counts is limited by the used physical technology (memory wear). Drives use *wear-leveling* that distributes write operations uniformly across all cells to maximize flash memory life-time.

The sector addressing used for an operating system block device is different than the internal representation of a flash storage. The addressing layer is called flash translation layer (FTL). FTL can be implemented inside the drive (most of SSDs and flash disks) or can be implemented or accessed in software (in specialized flash-friendly filesystems). In the latter case, there must be a low-level interface to allow a software access to the internal physical layer.

2.4.5 Network Attached Storage and Cloud Services

Storage attached remotely can provide direct access to a block device (the sector operations are routed to a storage controller). The *iSCSI device* server is a good example. Large enterprise storage can also use a special dedicated storage area network (SAN). The Network Attached Storage (NAS) usually means IP-based file-sharing service [26].

For the cloud services, the situation is even more complicated. Various distributed systems can provide access to the storage on various levels, including distributed block storage.

For performance reasons, there can be various intermediate layers that provide *thin-provisioning* (allocating only used sectors from a common storage pool), *deduplication* services (the same data in various sectors are stored only once), *snapshots* (to access data snaphost in time using copy-on-write logic) or *cache* layers.

2.4.6 Secure Wipe

The safe way to wipe data on a storage device is an important factor in evaluating security of an FDE system. We need to use secure wipe in several contexts.
Secure wipe is an operation that will:

- fill the whole storage (or part of it) with data not distinguishable from a random noise;
- wipe the exact area of storage to destroy specific information (area with the encrypted key);
- re-write the exact area in such a way that no copies of previous area content exist.

There can exist an algorithm how to implement some of the secure wipe operation types for a specific storage configuration. Yet for the generic storage (with all possible configurations) the secure wipe would be very hard (or even impossible) to implement correctly.

For HDDs, Gutmann analyzed secure wipe and suggested some methods [27], followed by analyzing the same problem for semiconductor devices [28]. The author notes that described algorithms are no longer valid for modern HDDs with ultra-high densities and different technology [27].

For solid state drives, an effective wipe method must bypass FTL and clear directly physical flash blocks. Many drives do not expose interfaces for such an operation. An alternative is to use the special TRIM command (if available), which should guarantee that all following reads of the area specified will return a zeroed data (defined as the deterministic read zero after TRIM – DZAT [29]).

If we want to sanitize the whole drive, there are more methods. The drive itself can support *erase unit* command. An alternative is to wipe the whole drive twice (to be sure that the hidden area is overwritten as well). Unfortunately, as analyzed in [30], sanitizing is effective only for some drives.

2.4.7 Security Impact

The storage architecture can directly influence security of the system. For example, modern *solid state drives* (SSDs) contain processors capable of highly sophisticated operations. The need of a complex flash sector storage management also implies that various garbage collector and optimization processes are running inside these drives. These processes are running invisibly in the background to provide optimal performance and life-time optimizations (*wear-leveling*). The drive shuffles the sectors to keep the drive performance optimal. Unfortunately, it also means that with these drives it is impossible to ensure that erasing a sector really destroys the information stored there. There can be even more copies of the same sector internally, as a side effect of drive optimization algorithms.

The sector reallocation problem is not new. For rotational media, sector reallocation exists as well (if a drive detects a physically defective area it reallocates sectors to a special reserved area). Yet what is an exceptional procedure (usually caused by a media error) for rotational drives becomes a normal operating mode for SSDs.

2.4.8 Performance

Performance of a storage system is very important factor in the storage industry, influencing many decisions including security. For FDE, we can see that fast chips (operating in the ECB mode) were massively used [1], the design proposal of BitLocker [31] discussed performance as a key factor. Recently, a new fast (but controversial) cipher for mobile device encryption was introduced [32], just be to be retracted few months later [33] and replaced with not yet widely analyzed HPolyC lenght-preserving encryption [34].

Another example is a low iteration count for key-derivation algorithms where the intent is to calculate a key quickly in all situations and on all supported hardware systems. It leads to very efficient dictionary attacks. One typical case is the TrueCrypt key derivation analyzed in Chapter 3.

2.5 Cryptography and Limiting Factors

High performance is an important factor for real-world usability of FDE. The storage encryption layer uses *symmetric block ciphers* for data encryption that can provide the needed performance. While the block size of block ciphers is usually smaller than the sector size, block ciphers must be operated in a *cipher mode*.

The encryption layer preserves sectors as independently accessible units. We can define it as a length-preserving encryption, therefore there is no space available for any additional context information inside encrypted sectors (sector size is not increased).

The same plaintext data located in different sectors must be encrypted to different ciphertexts. To fulfill this requirement without using any additional space for context information, we have to use implicit distinctive *per-sector* information. Most of FDE systems depend on the *sector number* (offset from the block device start). The sector number is a visible information to an attacker and it is usually transformed into an *initialization vector* by an additional operation.

The symmetric key used for sector-level encryption is called *volume key* or Media Encryption Key (MEK). It is either derived from a user passphrase by a *password-based key derivation function* (PBKDF) or generated by a *random number generator* (RNG). The generated key can be stored using different encryption methods in some special metadata area (on the disk or on another device) or stored on a special device (token).

2.5.1 Sector Encryption Algorithms

FDE uses block ciphers for encryption of sectors in a block encryption mode. The history of proposing encryption modes is rich [14, 35]. We will focus on currently used or proposed modes only.

Some systems use a variety of block ciphers according to user selection, but most of them have the Advanced Encryption Standard (AES) [36] as the default.

The first important issue is on what granularity we can localize a change in plaintext based only on snapshots of ciphertext. In an ideal situation it is the whole sector (as it is the best possible situation in the random sector access model). The ideal encryption produces ciphertext change diffused through the whole ciphertext sector for every possible change in the plaintext sector.

To provide such a granularity, we can use a block cipher with a large-enough block [37, 31]. With increasing sectors sizes, this solution is not applicable in reality. Also, the performance of these ciphers is not adequate for storage encryption. Most of the block ciphers today use a small block (like 128 bits in AES).

Another option is to use an additional independent operation before the encryption step (and symmetrically after the decryption step) that preprocesses the plaintext so the final encryption meets the whole-sector change requirement. This approach was used in BitLocker [31] with a diffuser operation called *Elephant*. It consists of two mixing operations that work in opposite directions and are applied sequentially. There is also an additional per-sector key applied to the ciphertext.

While the BitLocker design allowed to use encryption both with and without the Elephant diffuser, in the new versions of Windows operating systems one can use only the mode without it [38].

For sector encryption we do not need to solve any padding (variable size of plaintext) because storage sector size is always a multiple of the encryption block size.

2.5.2 Narrow Encryption Modes

Traditional cipher modes, more specifically called narrow-block modes due to the comparison to sector size, are the basic building blocks for a practical use of block ciphers. In the context of FDE, a very detailed analysis of many available modes is in the works of Fruhwirth [14] and Rogaway [35].

The electronic code-book (ECB) mode [37] is the simplest application of a block cipher to a sector. The plaintext (sector) is split into blocks of the block cipher size and these blocks are encrypted without any additional tweaks. We mention the ECB mode because it is the basic building block for other constructions, a direct use of the ECB mode would be very weak and vulnerable to known-plaintext pattern matching or reply attacks. Unfortunately, there are millions of drives with HW FDE that use the ECB mode [1].

A very common mode for sector encryption is the cipher-block chaining (CBC) mode. CBC is probably the most analyzed and most widely used mode (it was invented in 1976 and it is still used in many applications). The principle of the CBC mode is the exclusive or (XOR) applied between the previous ciphertext block and the following plaintext block. For the first block (where there is no previous ciphertext block), the *initialization vector* (IV) is used instead. The initialization vector must be unique for every sector. The IV is tweaked by the sector number and must not be predictable by an attacker. Fruhwirth documented the whole series of CBC issues in a special chapter of his work [14].

IEEE P1619 Security in Storage Working Group (SISWG) adopted the new narrow-block mode XTS (XOR-encrypt-XOR with ciphertext stealing) specifically designed for storage encryption [39, 40]. The standard itself defines AES as the given block cipher, but XTS mode can be used with any block cipher.

The XTS mode uses two keys. The first key is used to generate an IV and the second key is used for the encryption. The XTS mode eliminates the problem with an unpredictable IV internally, so the sector number can be used directly as a tweak value. Unlike the CBC, XTS is fully parallelizable (after the initial IV calculation all blocks can be processed in parallel). The concept is based on the XEX construction initially described by Rogaway in [41].

Independent encryption of blocks in the XTS mode implies that a simple change in plaintext propagates only to one block, so the granularity of a change is just one cipher block. The annex of the IEEE P1619.2 standard [40] mentions that the mode offers better protection against ciphertext manipulations than the CBC. The public comments for the NIST XTS document [42] discuss various problems in the XTS mode. One observation is that a limited propagation of a small plaintext change provides an attacker with a possibility to easily collect many cipher blocks with the same tweak and key. It can be used to perform fine-grained ciphertext manipulation attacks.

A simple ciphertext manipulation (and also demonstration that length-preserving encryption does not provide integrity protection) is illustrated in Figures 2.3 and 2.4 [43]. Here we intentionally modified an AES-XTS encrypted device in a specific pattern that appeared in the plaintext with block granularity as a pseudo-random noise.

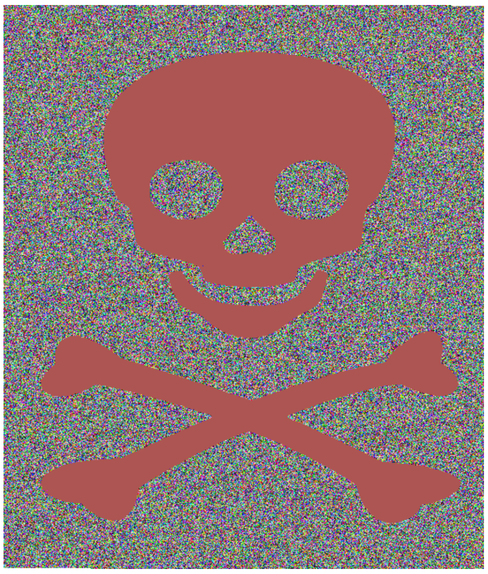
Figure 2.3: Pattern-modified ciphertext [43].

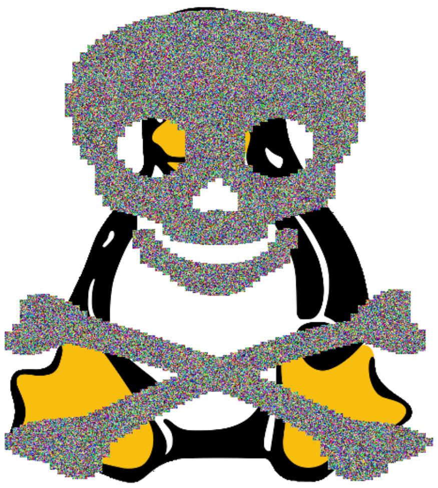
Figure 2.4: Pattern-decrypted plaintext [43].

2.5.3 Initialization Vectors

The IV in the CBC mode can be *fixed* (which causes similar problems as using the ECB mode), it can be a simple *counter* (sector number) or a *generated IV* with the sector number as one input parameter [44]. The *counter IV* is sometimes denoted as *plain IV*. The counter IV must never overflow, otherwise one IV is used for more sectors.

To remedy the problem of non-secret IV, we can define a simple function calculating the IV based on other attributes not known to the attacker.

An example of such a design is the encrypted salt-sector IV (ESSIV) [14]. The IV is generated by encryption of the sector number. It is encrypted using the same block cipher as used for sector encryption, just with the key derived from the volume key using a hash function.

If we have n as a sector number (padded by zeros to the required length), K as a volume key, K_{iv} as the derived key, E encryption function of the block cipher and H as a hash function (where hash size can be used as key size for E) then $ESSIV$ is defined as

$$
\begin{aligned}
K_{iv} &= H(K), \\
IV_n &= E(K_{iv}, n).
\end{aligned}
\tag{2.1}
$$

ESSIV does not need any additional information, the K_{iv} is simply derived from the volume key.

Another example is the IV calculation used in BitLocker [31], where the IV is calculated simply as encryption of the sector number with the same key as used for data encryption

$$IV_n = E(K, n). \tag{2.2}$$

Yet another example that uses additional encryption key, is TrueCrypt in the CBC mode. Here the IV is calculated as

$$IV_n = (n||n) \oplus K_{iv}, \tag{2.3}$$

where || is a simple concatenation operation, $\oplus$ is the XOR operation and the key K_{iv} is provided externally.

This IV construction is known to be insecure (with a properly aligned consecutive sectors we know which bits in n will change and with a properly formatted first block in the sector we can create a special detectable pattern in the ciphertext) [45].

Illustration of IV importance is provided in Figure 2.5, where we used the ECB mode encryption and in Figure 2.6, where we used the XTS mode with a constant IV for all sectors.

2.5.4 Wide Encryption Modes

There are several wide-block encryption modes that operate sector-wise (the whole sector changes on a random plaintext change).

As an example, *EME* (ECB-mix-ECB) is a cipher mode developed by Halevi and Rogaway [46]. It consists of 5 stages with 2 encryption phases and 3 complete data traversals. The IEEE 1619.3 standard proposal (not updated for several years) contains a draft of the EME2 block mode [47].

Another wide-block mode CMC (CBC-mask-CBC) [48] by the same authors is based on the CBC mode (reusing possible existing CBC implementation). Unlike EME, CMC is not parallelizable.

Due to performance reasons (time cost of at least a two-pass encryption) and patent status, wide-block modes are not yet widely used in sector encryption applications.

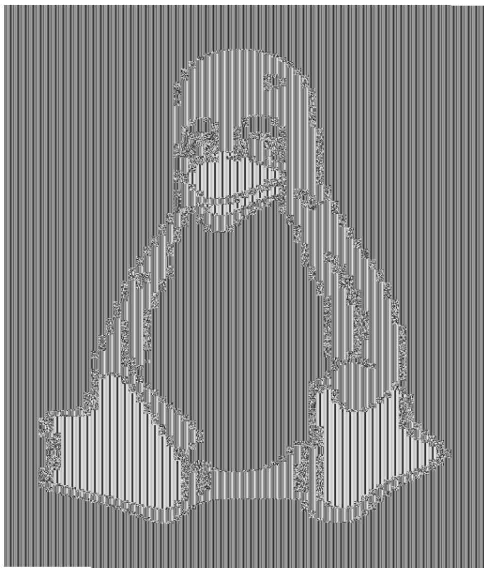
Figure 2.5: Patterns in ECB mode. [43].

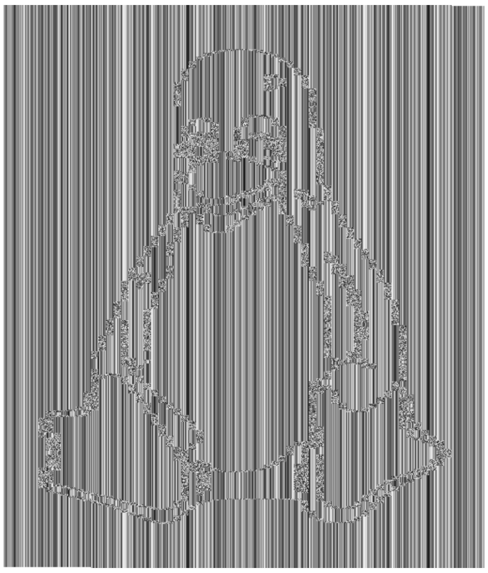
Figure 2.6: Patterns in XTS with a constant IV [43].

A very recent addition is HPolyC [34] mode based on XChaCha12 that claims to provide a very fast mode for disk encryption with wide encryption mode characteristic.

2.6 Key Storage and Management

Encryption key is the secret that should guarantee confidentiality in FDE systems. Initially, the volume key is generated by using a random number generator (RNG) or it is derived from a passphrase. Key management should provide a reliable way for a key manipulation while still providing enough comfort for the user.

There are many hardware devices helping to solve this problem – tokens, smartcards, Trusted Platform Modules (TPMs). For example, the BitLocker key scheme allows one to unlock the same device by a passphrase combined with a TPM or a token (multi-factor authentication) or alternatively using a recovery password [15].

A common way for authentication is still a password (or passphrase) and it seems that this is not going to change in the next years [49]. Moreover, software-based encryption works with common drives that do not contain any special encryption and key storage capabilities.

2.6.1 Key Hierarchy and Passwords

The encryption key can be stored on the encrypted device itself (or on different but common storage media) in a special area (metadata area or header) encrypted with a different key. There is a *key-hierarchy* (or *key-wrapping*), one key is stored as data for another encryption. An example of a simple key-hierarchy system is TKS1 defined by Fruhwirth for the LUKS encryption system [50]. Key hierarchy systems allow one to change a passphrase without a change to the volume encryption key. Validity of a decrypted candidate volume key can be verified by using a fingerprint (LUKS, BitLocker) or by searching for a known signature (TrueCrypt, VeraCrypt) in the decrypted metadata area.

The on-disk metadata can contain a visible signature (in this case it is clearly detectable that the device uses encryption) or it can be encrypted in such a way that without a key it should not be possible to detect that an encryption is in place.

2.6.2 Deniable Encryption

FDE can be used to build *deniable encryption* storage systems. Deniable encryption [51] (also named plausibly deniable encryption) is often a simple steganographic technique for hiding an encrypted device inside a so-called decoy device. This allows users to provide decoy keys (in a plausible manner) when someone seized their device, but the confidential data are still stored on the hidden device.

An example of a plausibly deniable encrypted device is the hidden disk in TrueCrypt (or a hidden operating system that is just an extension of this basic concept) and is analyzed in Chapter 3. Another recent design of a plausible deniable storage on a mobile platform (Android) is *Mobiflage* [52]. Its design supports more hidden (or decoy hidden) volumes.

2.6.3 Password-Based Key Derivation Functions

When the volume key is stored in a key-hierarchy system, the unlocking key (the key that encrypts the volume key) can be derived from a user passphrase by a password-based key derivation function (PBKDF) [53].

PBKDF is an algorithm that produces an output directly usable as a cryptographic key from a secret password or passphrase (and from other public attributes). The main goal of a PBKDF is to increase complexity of key search attacks (dictionary attacks to password). The password or passphrase is provided by the user and as such has usually bad randomness characteristic and low entropy. (While here we mention password or passphrase, PBKDF can have another key as an input.)

For the FDE case, unlocking can be an offline process (common use case is a fully encrypted laptop, the user has to provide his password before the network is even initialized). PBKDF runs always on the user's machine and is limited by the available resources there. The attacker can use any resource he can afford. The attack cost has also an economic significance – if the attack requires a lot of energy, it can slow down the attacker significantly [54].

The most used PBKDF is PBKDF2 [55] that can be defined as

$$K = PBKDF2(PRF, password, salt, iterations, K_{length}), \tag{2.4}$$

where K is a derived key of length K_{length}. *PRF* is a pseudo-random function (HMAC based on a hash function), *password* is the secret information provided by the user and *salt* is a public cryptographic salt with the purpose to avoid building dictionary of precomputed hashes (rainbow tables). The *iterations* parameter specifies how many times the process repeats until K is provided.

Increasing iterations implies increasing work (and time) needed for deriving the key. PBKDF2 is designed to be not internally parallelizable. The iteration attribute should be set to as high as it can be tolerated for the environment and acceptable for the user [53].

The main downside of PBKDF2 is a constant (and small) memory footprint during the whole operation. This allows an attacker to use massive parallel systems like graphics processing units (GPUs) or application-specific integrated circuits (ASICs) to speed up dictionary search [56].

The problems with no longer suitable PBKDF2 lead to proposals of new PBKDF algorithms. The first one was scrypt [57, 58]. Scrypt introduces the concept of *sequential memory-hard* algorithms (which try to use as much memory as possible for a given number of operations and also that it cannot be computed on parallel systems more efficiently). The goal is to make the parallel attacks more expensive. We analyze several algorithms in Chapter 5 in more detail.

2.7 Data Integrity

Cryptographic integrity protection combined with encryption of stored data (and metadata) in sectors is discussed in Chapter 4. Cryptographic integrity protection can be also implemented separately (it does not provide confidentiality then).

One such solution is *dm-verity* used in secure boot for Google's Chromebooks and Android systems [59]. This layer has a pre-computed tree of hashes (Merkle tree) of fixed-size blocks (the device itself is read-only in normal operation). The top-level hash (root hash) is verified by a public key stored on the device. While dm-verity provides integrity protection only (anyone can recompute hash tree, there is no secret key), the top level hash is signed. This operation adds authentication (only a user with the valid private key can sign the root hash).

3 TrueCrypt and VeraCrypt Evolution – Case Analysis

One step in our research is to analyze and describe certain known problematic areas of existing full disk encryption design and propose modification to the identified problems.

TrueCrypt [60] (and follow-up VeraCrypt [61]) provides a very valuable insight into the history of software-based FDEs. TrueCrypt is one of the widest-known open source tools providing FDE. It is available for multiple operating systems (OS) including Windows, Mac OS, and Linux, letting you share encrypted storage among these systems. Development of TrueCrypt was abruptly stopped in May 2014 and one of the existing clones named VeraCrypt de-facto continued where TrueCrypt was abandoned.

Our analysis is based on own, independent reimplementation of the TrueCrypt/VeraCrypt format that is integrated into cryptsetup and dm-crypt projects [11, 62] and is used in almost all Linux distributions.

Our most important implementation goal was to disregard the original source code (we did not want to use it even for reference), to avoid either tainting our code with hidden problems or violating the original license. We wrote our TrueCrypt format-handling code from scratch, using only the available documentation, inspired partly by *tc-play* [63] another independent open source implementation. The following format description is based on our implementation. Although we intended our implementation only for Linux, our comments apply for all supported architectures.

3.1 TrueCrypt Containers and Linux

Implementing a disk encryption system involves two major tasks: providing a safe way to store symmetric encryption keys and implementing the disk encryption layer. Whereas the application usually handles key storage, disk encryption can be an integral part of an OS. The authors of the original TrueCrypt for Linux used several approaches.

The first release supporting Linux (version 4.x) used its own Linux kernel module to perform encryption. Version 5.x used outside-the-kernel (user-space) disk encryption (which degraded performance notably).

Starting with version 6.0, releases have used dm-crypt [62], a native Linux kernel disk encryption module. Windows systems also employ a separate system loader that implements only essential functions enabling system initialization from an encrypted system disk.

A common complaint about TrueCrypt (for all supported OSs) is its implementation of the cryptographic primitives (hash and block ciphers). In contrast, Linux native encryption module and configuration utility cryptsetup [11] show that using common cryptographic libraries is not hard at all.

A TrueCrypt container consists of a data area (usually a whole disk or partition) containing both encrypted data and the header. The header, which is also encrypted, contains metadata: the key and other information employed for user data encryption. User data are stored in the container's remaining space.

The header is encrypted with the same algorithm as the data, but with a key derived from a user-provided password and salt (random data visibly stored in the header). Because the header is encrypted, no one should be able to detect a volume existence unless he knows the password.

There are three basic encrypted-volume types – a *standard* volume, a *hidden* volume (another volume hidden in the decoy encrypted volume), and an *OS volume* (encryption of the bootable operating system partition).

The hidden volume resides in a decoy volume's unused file system areas. A decoy OS contains a decoy encrypted volume; the secret OS with the secret data is in a hidden volume. A hidden volume aims to provide *plausible deniability*. It lets you reveal the encryption key to a decoy encrypted (outer) volume while the real data remain secure in the decoy's unused file system areas (encrypted with different key).

A hidden OS hides the entire OS. A decoy OS contains a decoy encrypted volume; the secret OS with the secret data is in a hidden volume. System encryption requires a special boot loader, which, according to the provided password, loads the decoy or hidden OS[64]. An illustration of volume types use can be seen on Figure 3.1.

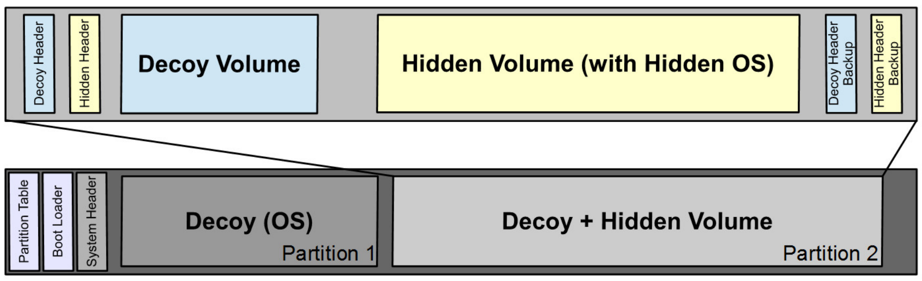

Figure 3.1: TrueCrypt encrypted volume types – example of hidden operating system.

3.2 On-disk Header Format

The header is at a fixed position, different for each volume type. All headers use the same internal format.

A TrueCrypt header is a sector consisting of 512 bytes. Since version 6.0, the reserved area for the header has increased (to 128 Kbytes). However, the remaining space seems to be used only to properly align the data area for disks with larger block sizes. As our implementation proved, only this one sector contains the required data for complete data decryption.

A header starts with 64 bytes of randomly generated salt. Each header salt is different, even if the header is encrypted with the same password as another header. Next comes an area containing volume metadata information and keys. This area is encrypted by an algorithm from a known algorithm list. You determine the used algorithm by trying all known variants with the provided password and salt. You validate a properly decrypted header using the checksum; the header will start with the text string *TRUE* for TrueCrypt or with *VERA* for VeraCrypt. Binary format is otherwise the same for both TrueCrypt and VeraCrypt headers.

A standard TrueCrypt header is in the storage device's first sector (with a backup near the device end). A hidden header is near the device end or (since version 6.0) in the middle of the extended standard header (squeezed into an unused data alignment area). For system encryption, the header cannot be in the system partition (there is no additional space), so it is in an unused space between that partition and the partition table.

3.2.1 Key Derivation Algorithms

To derive the key from the provided passphrase, you use the password-based key derivation function PBKDF2 [55]. The maximum password length is 64 bytes. During volume formatting, users can select the hash functions required for the key derivation functions. TrueCrypt supports the hash functions SHA-1 (obsoleted in version 5.0), SHA-512, RIPEMD160, and Whirlpool.

To slow down brute-force and dictionary attacks, the key derivation function uses a sequential iterated function. In TrueCrypt, the iteration count is fixed to (only) 1000 for newer algorithms or 2000 for SHA-1 and RIPEMD160.

VeraCrypt authors tried to solve the problematic low iteration count for PBKDF2 by introducing two changes. One change is adding new hash algorithms for PBKDF2 with increased (but still fixed) iteration count to 500000 (or 200000 for OS encryption header) for SHA-512, Whirlpool and SHA-256 or to 655331 (327661 for OS) for RIPEMD160.

The latter change is an optional user-configurable iteration count named Personal Iterations Multiplier (PIM). PIM is a number that the user must provide together with the unlocking passphrase. Both changes significantly increase unlocking time for a container on a slower hardware.

3.2.2 Encryption Types

TrueCrypt has supported many encryption types. Encryption algorithms include AES competition finalists AES-256 [36], Serpent [65], and Twofish [66]; older versions include Blowfish, CAST5, TripleDES, and IDEA. These were removed later in VeraCrypt. Some older algorithms are only for existing containers, not new ones. TrueCrypt no longer supports IDEA, owing to licensing problems. VeraCrypt also added Camellia [67] and Kuznyechik [68] ciphers.

3.2.3 Encryption Modes

The encryption always works with 512-byte sectors even if the underlying device can have a larger physical sector. Block ciphers support only 64 or 128-bit encryption blocks, so the sector must be encrypted in parts using a block encryption mode. Over time, the encryption mode has switched from CBC (cipher-block chaining) to LRW (Liskov, Rivest, and Wagner) to XTS (XOR-encrypt-XOR with ciphertext stealing).

3.2.4 CBC Mode and Whitening

In CBC, TrueCrypt added whitening (originally called sector scrambling), which combined user data (sector content) with other data derived from a secret key. This process calculated the whitening value using multiple CRC32 (32-bit cyclic redundant check) operations over the sector number, seeded by the secret key. It then repeatedly applied the calculated per-sector value to the whole sector. We believe this is because it aimed to complicate chosen plaintext attacks.

For disk encryption, two sectors with the same data must not produce the same pattern in the encrypted data. So, the encryption tweaks every sector by using sector-context-specific information called the initialization vector (IV). In CBC, TrueCrypt calculated the IV by a simple XOR (exclusive OR) operation from another secret key and the sector number.

This design had serious issues. First, even though the IV was derived from a secret key, it is partially predictable (you could predict the next IV in some cases because the sector numbers were consecutive). Second, whitening was just a linear transformation (not using a cryptographically secure one-way function). This could lead to *watermarking attacks* [45] – special pattern detection from ciphertext. Such attacks successfully revealed the existence of hidden volumes (just from ciphertext analysis) and probably directly led to CBC deprecation in TrueCrypt version 4.1.

3.2.5 LRW and XTS Modes

CBC was replaced with LRW as (at that time) a candidate for the IEEE 1619 standard for storage encryption. TrueCrypt used it with an extension for 64-bit block ciphers (required for Blowfish).

LRW used a tweak key, a special secondary key that solved the predictable IV problem so that the per-sector context could be the sector number. The tweak key was independent of the encryption key; it mixed context information (the IV) directly in the encryption mode. The IV was the sector position – the predictable sector number combined with the tweak key.

TrueCrypt version 5.0 switched to XTS, defined in the IEEE 1916-2007 standard. XTS uses two independent keys: one for the main encryption and the other for a tweak key (used for the same purpose as in LRW). The IV is the sector number.

3.2.6 Chained Ciphers

Chains of ciphers (instead of just one cipher) appeared in TrueCrypt 3.0. When one cipher is broken (and no longer provides privacy), another still encrypts the volume. This approach obviously assumes that ciphers do not weaken if they are in a chain. There is also a notable performance penalty.

A chain can comprise two or three cipher algorithms: Blowfish (later replaced with Twofish), AES, or Serpent. The ciphers order should be probably the same as the order of the keys stored in the header, but TrueCrypt uses reverse notation for some chains. VeraCrypt introduced new chains with Camellia and Kuznyechik.

Ciphers with different block sizes could cause a problem (Blowfish has smaller blocks). So, for CBC, TrueCrypt defined the *inner CBC mode*. This mode applied the block cipher mode even inside a chain to prepare a full block as input for another cipher in the chain. If all ciphers had the same block size, TrueCrypt employed the *outer CBC mode* and directly used one cipher's block output as input for another cipher. Whitening (and the IV) applied only once in the chain. In LRW and XTS, the chain simply redirects one cipher's output as input to another. This is equivalent to stacking separate encrypted devices.

3.3 Encryption Keys

There are two sets of encryption keys. One is generated from the header salt and password and is for decrypting the on-disk header. The other is stored in the header and contains the real encryption keys.

There are three keys $K_{1,2,3}$ for encryption algorithms, corresponding to the three possible ciphers in a chain. (If there is only one cipher or a two-cipher chain, the remaining keys or key is omitted.) These keys are independent and generated from RNG.

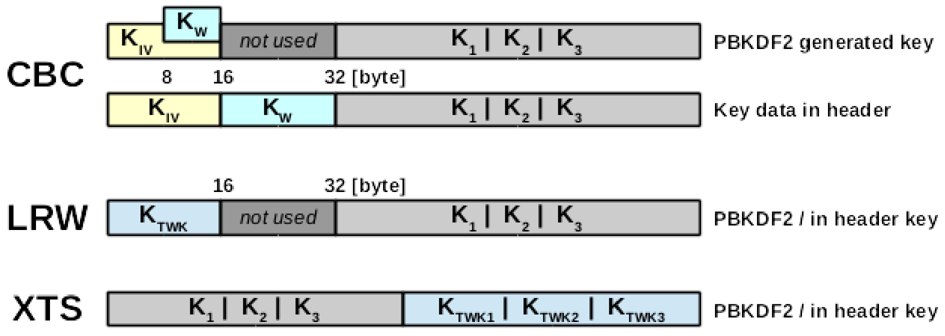

Figure 3.2: TrueCrypt Keys stored in volume header.

CBC also had a key K_{IV} used as a seed for IV calculation and a key K_W used as a seed for whitening. For some reason, the password-derived whitening key was only half of normal size (repeated twice) and was shared with the second part of the IV key. For keys decrypted from the header, these two keys were full length and independent. As we mentioned before, LRW also had a tweak key K_{TWK}, which was the same for all ciphers in a chain. XTS has independent tweak keys $K_{TWK1,2,3}$ as secondary keys for the ciphers. Keys storage is illustrated in Figure 3.2.

3.3.1 Keyfiles

You can supplement passwords with keyfiles. This approach uses CRC32 to mix the keyfile content (actually, just the first Mbyte of the keyfile) with the password. Mixing then uses a 64-byte pool, which limits the password to this size.

Keyfiles should ensure that users need both the password and keyfiles to unlock the volume. Contrary to the TrueCrypt security model, some attacks remove the keyfiles' influence, so that they can decrypt the header with just the password [45]. These attacks require specially prepared keyfiles.

3.4 Format Evolution

Initially, the TrueCrypt on-disk format design probably inherited some issues from the E4M (Encryption for the Masses) project. The algorithms used in TrueCrypt apparently evolved as new problems appeared. Most of TrueCrypt visible security problems were due to using its own (insufficiently secure) algorithms with CBC. These included problematic whitening and the partially predictable IV.

Since the current version uses XTS without whitening, these problems no longer exist. However, the keyfile-mixing algorithm could be problematic.

VeraCrypt tried to solve problems with low iteration count for PBKDF2, but at the cost of usability (unlocking is very slow or the user have to remember an additional PIM value).

In addition, modern flash-memory storage devices bring up security risks for disk encryption systems. One such problem is caused by wear leveling (arranging data writes such that no memory cell fails prematurely by exceeding the write-cycle limit). As data written to the same logical address are written to different physical locations, the old data content (in the original location) could remain on the device. The security problem here appears mainly if the user re-encrypts a header with a different password but an old header is still physically on the device. (This can happen even with other types of storage devices during bad-sector relocation but is the normal operating mode for flash-based devices.)

Another problem concerns TRIM, a special command that informs the device that a certain area of the medium is no longer in use. If TRIM is used with FDE, the disk's unused areas can be easily identified (the space contains detectable patterns). TRIM can also destroy hidden volumes (from the decoy OS viewpoint, this is unused space).

A trivial example of the introduced risk is that without TRIM, you cannot say which encrypted disk contains real data – all disks contain ciphertext (looking like random noise). However, with TRIM, you can easily identify empty disks without decryption.

The TrueCrypt security model does not allow for TRIM usage. On the other hand, TRIM can provide notable performance advantages and in some cases can be in compliance with the threat model. Leaking some side information such as the pattern of used sectors could be a marginal problem, yet the used data are encrypted.

Device wiping also becomes an issue. TrueCrypt always wipes the whole formatted device with random data. However, with hidden reserved flash space on a disk, this wiping might be insufficient. A simple workaround is to wipe the device twice to ensure that the reserved flash memory contains no residual data.

4 Authenticated Sector Encryption

A major shortcoming of current FDE implementations is the *absence of data integrity protection*. Confidentiality is guaranteed by symmetric encryption algorithms, but the nature of length-preserving encryption (a plaintext sector has the same size as the encrypted one) does not allow for any metadata that can store integrity protection information.

Cryptographic integrity protection is useful not only for detecting random data corruption [69] (where a CRC-like solution may suffice) but also for providing a countermeasure to targeted data modification attacks [70]. Currently deployed FDE systems provide no means for proving that data were written by the actual user. An attacker can place arbitrary data on the storage media to later harm the user. FDE systems such dm-crypt [14, 62] or BitLocker [31] simply ignore the data integrity problem, and the only way to detect an integrity failure is the so-called *poor man's authentication* (user can recognize data from garbage produced by the decryption of a corrupted ciphertext).

The aim of our work is to *demonstrate that we can build practical cryptographic data integrity and confidentiality protection* on the disk sector layer with acceptable performance and without the need for any special hardware. Our solution is an open-source extension of existing tools; we do not invent any new cryptographic or storage concepts. To achieve this goal, we implemented a per-sector metadata store over commercial off-the-shelf (COTS) devices. We provide an *open-source implementation as a part of the mainline Linux kernel*, where it is crucial to avoid proprietary and patented technology.

We focus on the *security of authenticated encryption and the algorithm-agnostic implementation*. Our main contributions are as follows:

- separation of storage and cryptographic parts that allow changing the underlying per-sector metadata store implementation without modifying the encryption layer,
- the concept and implementation of emulated per-sector metadata,
- algorithm-agnostic implementation of sector authenticated encryption in the Linux kernel and
- use of random initialization vector for FDE.

Storage security has often been perceived as an additional function that can easily be added later. The history of length-preserving FDE is a demonstration of this false idea. Although it is a simple application of cryptography concepts, some vendors deployed FDE not only with known vulnerabilities but also with incorrectly applied cryptography algorithms [1, 2].

4.1 Threat Model and Use Cases

Use cases for of FDE can be categorized to several situations, like

- *stolen hardware* (mobile device, laptop),
- *devices in repair*,
- resold *improperly wiped* devices,
- *virtual device* in a multi-tenant environment, or
- a *mobile device* storage.

In all of these scenarios, confidential data can leak out of the control of the owner. An analysis of the content of used drives [71] shows that encryption is often not used despite the importance of storage data encryption being recognized for a long time [72].

Type	**Description**
Random	An attacker with user skills, can investigate disk content. Focuses on hardware, to steal and sell it. Valuable data are simply a bonus. A typical representative for this category is a random thief.
Focused Opportunistic	An attacker can run advanced forensic tools and can use documented attacks (such as a brute-force password search). The goal is to obtain data, analyze them, and use them to make a profit. The attack is not targeted to a specific user, but if the recovered data allow such an attack, the attacker can switch to this type of attack. A typical representative is a computer technician with access to devices for repair. In some cases, the attacker can access the device repeatedly, but only in limited opportunistic occasions.
Targeted	A top skilled attacker that uses the best possible attack vectors, typically to focus on a specific user with exactly defined goals in advance. The user system is generally under full attacker control.

Table 4.1: Attackers.

Our threat model adds to all these scenarios the detection of *data tampering* on leaked (not-wiped) devices and expects that an attacker has limited ability to record device changes with access snapshots of the device in time. The model recognizes the *Focused Opportunistic* attacker defined in

Table 4.1. It does not protect data in situations where a device is in active (unlocked) state or an attacker can get the unlocking passphrase (and thus encryption key) directly. Our model expects COTS devices, it cannot rely on use of any of tamper-proof cryptographic devices like Hardware Security Modules (HSMs).

4.1.1 Attackers

We define three simplified types of attackers, as summarized in Table 4.1. The most common type of attacker for FDE is a *random* attacker. We define a *focused opportunistic* attacker type for sophisticated attacks that focus on the situation where the stolen device is returned to the user. In the *targeted* attacker case, FDE will not provide sufficient protection without additional countermeasures.

4.1.2 FDE Protection Types

For the description of our model, we define three basic levels of protection, as summarized in Table 4.2.

Here, FDE protection (of any type) means that the data confidentiality is enforced. A simple case of a device theft means that only hardware is lost. Data remain encrypted and not accessible to the attacker. This scenario is covered by the *Pure FDE* protection. The importance of *authenticated FDE* comes into play when the stolen or seized device returns to the user (and this often occurs in reality; and example can be mandatory border checks).

Type	**Description**
Pure FDE	Length-preserving encryption that provides confidentiality only.
Authenticated FDE	Encryption that provides both confidentiality and integrity protection, but limited by COTS devices (no hardware for authentication).
HW-trusted	The ideal solution with confidentiality and integrity protection. It stores some additional information to external trusted storage in such a way that the system can detect data replay.

Table 4.2: Discussed types of FDE protection.

This situation is generally enforced by a security policy and compliance of users. The reality is different – experiments show that people plug-in foreign devices, even if such devices are obtained under suspicious circumstances [73].

Authenticated encryption enforces that a user cannot read tampered data but will see an *authentication error*. It not only stops any attempts to use tampered data on higher layers, but also helps a user to realize that the device is no longer trustworthy. An overview of the features among FDE types is summarized in Table 4.3.

FDE type:	**None**	**Pure FDE**	**Auth. FDE**	**HW trusted**
Confidentiality	×	✓	✓	✓
Integrity	×	×	✓	✓
COTS hardware	✓	✓	✓	×
Detect silent corruption	×	×	✓	✓
Detect data tampering	×	×	✓	✓
Detect data replay	×	×	×	✓
Whole sector change	×	(✓)	(✓)	(✓)

Table 4.3: Overview of FDE features.

4.1.3 Data Corruption and Forgery

So-called silent data corruption [24] is a common problem in persistent storage. This problem occurs when data are unintentionally and randomly corrupted while traversing through the storage stack. It is generally caused by flaky hardware (unstable memory bits and loose cables), triggered by an external influence (ionizing radiation) or by misplacement of data (correct data are written to an incorrect place). Data are then stored and used in a corrupted form.

A solution is to detect data corruption by checksums such as CRC32 [74]. This solution is not adequate if we want to detect an intentional unauthorized change (an attacker will simply fix a checksum).

An active attacker can not only cause undetected data corruption by simulating silent data corruption but can also attempt to forge the data by corrupting a specific disk area. A more sophisticated attacker can store additional data using steganographic techniques (conceal other data) to unused areas of a disk. We have to use a cryptographic integrity protection to detect such a situation.

4.1.4 Replay Attacks

In the strong view of cryptographic integrity protection, we should also detect data replacement using old content (revert to snapshots, also called *replay attack* [75]). Such a requirement cannot be fulfilled without an additional trusted metadata store independent of the storage itself. If we have such a storage, it can be used to store a Tamper Evident Counter (TEC).

The content of the entire storage can always be completely replaced with an older snapshot. Without additional and trusted information, users cannot recognize such a situation. An attacker can also revert only a partial part of the storage (in our case, selected sectors). From the cryptographic perspective, this situation cannot be completely prevented or detected (it would require breaking the independence of sectors).

In this text, we present algorithms that *do not protect* from the replay attack. This decision is based on the fact that our work is focused on utilizing standard disk drives without any additional hardware requirements.

4.2 Encryption and Data Integrity

Data encryption and integrity protection can be performed on different layers. An efficiency advantage comes with implementation on higher layers (only used areas are processed). However, integrity protection on the FDE layer provides at least some integrity protection of storage space, even in situations when a higher layer does not provide such guarantees (unfortunately, this is still quite common).

4.2.1 Length-preserving Encryption

Length-preserving encryption algorithms are used in current FDE solutions. These algorithms transform original data (plaintext) to its encrypted form (ciphertext), and confidentiality is based on the use of secret keys. Storage data are (for performance reasons) always encrypted with symmetric block ciphers.

The block sizes of these symmetric ciphers are typically 16 bytes. The device sector size is considerably larger (at least 512 bytes); thus, to apply encryption per sector, we have to utilize a *block encryption mode* inside a sector.

Current FDE systems use CBC [31] or XTS [39] modes. The CBC mode has many known problems [14] and specific requirements.

The XTS mode, due to the internal parallel processing of blocks, can leak more information about a change in the plaintext than other modes. If only a single byte changes in a sector, then we can localize the change with the internal cipher block granularity. An ideal FDE system should produce a pseudorandom change of the entire sector data. Also, these modes produces the same plaintext data encrypted to the same sector always produce the same ciphertext. In cryptography, this means that such a system does not provide indistinguishability under chosen plaintext attack (IND-CPA) [76, 77].

4.2.2 Authenticated Encryption

We have two options for integrity protection combined with device encryption: either to use *Authenticated Encryption with Additional Data* (AEAD) [78, 79] or to combine length-preserving encryption with an additional cryptographic integrity operation. The major difference is that for the combined mode, we can ignore integrity tags and decrypt the data without such tags. In the AEAD mode, the authentication is an integral part of decryption. Additionally, for the combined mode, we need to provide two separate keys (encryption and authentication), whereas the AEAD mode generally derives the authentication key internally. Both mentioned integrity methods calculate an *authentication tag* from the final ciphertext (encrypt-then-MAC).

The encryption operation output consists of the encrypted data and the authentication tag. Authentication mode with additional data (AEAD) calculates the authentication tag not only from the input data but also from additional metadata, called additional authentication data (AAD). Table 4.4 summarizes examples of the encryption modes mentioned in this text.

4.2.3 Initialization Vectors

The Initialization Vector (IV) is a value for an encryption mode that impacts encryption. In FDE, the IV must always be derived from a sector number (offset from the device start) to prevent malicious sector relocation. The sector number guarantees that the same data written to different sectors produce different ciphertexts. The proper use of IVs and nonces depends on the exact encryption mode and is *critical* for the security of the entire solution. Table 4.5 briefly describes the IV types used in our work with a formal definition in Section 2.5.3. For the CBC mode, we must use an IV that an adversary cannot predict; otherwise, the IV value can be used (in combination with a specially formatted plaintext) to create special patterns in the ciphertext (watermarks) [14, 2]. Some encryption modes (such as XTS) solve this problem by encrypting the IV such that they can use a predictable sector number directly.

Mode	Description
AES-CBC	AES [36] non-authenticated mode used in legacy FDE systems. [14]
AES-XTS	AES [36] non-authenticated mode used in recent FDE systems. [39]
AES-GCM	AES [36] in the Galois/Counter authenticated mode. [80] Due to only 96-bit nonce, it can be problematic in the FDE context. [81]
ChaCha20-Poly1305	Authenticated mode based on the ChaCha20 [82, 83] cipher and the Poly1305 authenticator. Only 96-bit nonce is used.
AEGIS128 / 256	AEAD ciphers based on AES round function (CAESAR [84] finalist) [85].
MORUS640 / 1280	AEAD ciphers designed for modern hardware optimizations (CAESAR [84] finalist) [86].

Table 4.4: Examples of encryption algorithms.

IV	Description
plain64	Sector as a 64-bit number (device offset). Used for the XTS mode [62].
ESSIV	Encrypted Salt-Sector IV. The sector number is encrypted using a salt as the key. The salt is derived from the device key with a hash function. Used for the CBC mode [62].
random	IV generated on every sector write from a Random Number Generator (RNG). Used for AEAD modes.

Table 4.5: Initialization vectors (IV generators).

The IV must always be unique per sector. In some cases, the IV must be a *nonce* (a public value that is never reused). Repeating an IV for different sectors not only opens a possibility to malicious sector relocation but can also violate a security restriction (in the GCM mode repeating a nonce value is fatal [87]).

4.2.4 Error Propagation in Encrypted Sector

With the symmetric encryption in the processing stack, error propagation is amplified. One bit flip causes a random corruption in at least one cipher block (typically 16 bytes) up to the entire sector. This is illustrated in Figure 4.1.

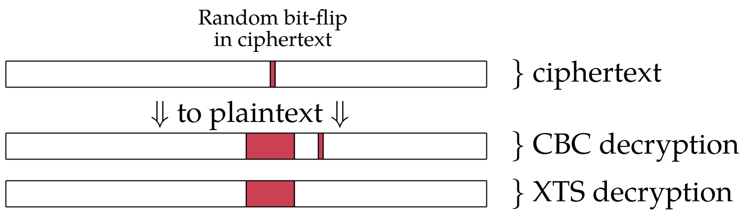

Figure 4.1: Error propagation in encrypted sector.

Such a "random corruption" means that decrypted data are a product of the decryption of a modified ciphertext. By definition, modern ciphers [44] must produce a random-looking output. In other words, a user will see a block full of data garbage after decrypting corrupted data.

For encryption, the change propagation is, in fact, a desirable effect. The ideal situation is that any change in the sector data is propagated to a pseudo-random change to the whole encrypted sector as illustrated in Figure 4.2.

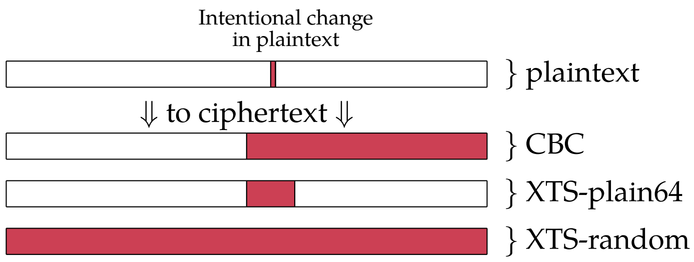

Figure 4.2: Change propagation to encrypted sector.

4.3 Metadata Storage Placement

The absence of per-sector metadata (to store integrity protection data) is a well-known problem. Integrity protection requires a length expansion of the processed data and thus needs additional storage [88]. Common sector sizes are 512 and 4096 bytes, and we need an independent metadata per-sector.

4.3.1 Metadata in Hardware Sector

A reliable way to handle integrity metadata is to do so directly in the device hardware sector. One approach proposed in [89] is to convince manufacturers of storage drives to increase the hardware-sector size to accompany integrity data directly there. This solution obviously cannot be used for existing drives.

An in-sector integrity data approach appeared in 2003 as the T10 Data Integrity Field (DIF) extension for SCSI devices [90]. This idea was implemented by vendors as T10 Protection Information (T10-PI) where a sector is expanded by 8 bytes (the sector size is 520 bytes) [23].

For COTS devices, the DIF extension is quite rare, expensive and requires a special controller. The fixed provided metadata space is not large enough for cryptographic data integrity protection that requires storing an Initialization Vector to metadata.

4.3.2 Metadata Stored Separately

Per-sector metadata can be stored in a separate storage space on the same disk [91] or an external fast storage [88]. We can also join several smaller hardware sectors and metadata into one large virtual sector presented to the upper layer.

This approach is used in authenticated modes in the FreeBSD GEOM encryption system [92, 93]. The initial GEOM Based Disk Encryption (GBDE) tried to use additional metadata for generated per-sector keys. This design does not provide safe atomic sector updates and also significantly decreases performance (throughput is only 20-25% of an underlying device) [94].

Some of these problems were fixed by the GELI disk encryption [93]. Here, the upper layer uses a 4096-byte sector, while internally, it splits data into 512-byte native sectors (each contains its own data and metadata).

In this case, for every presented sector, 8+1 native sectors are used. This concept ensures that integrity data are handled on the atomic hardware sectors. Although the above filesystem sees the 4096-byte sector as an atomic unit, the device operates with 512-byte units and can possibly interleave multiple writes.

Another problem with this design is that the virtual sector is always larger than the native device sector. If the native sector is 4096 bytes, then the presented virtual sector can be larger than the optimal block size for the filesystem above (for some filesystems, the optimal size can be a page size, and thus, it is 4096 bytes in most situations).

Enterprise storage vendors also implement a multiple-sector schema, generally with 8 data sectors + 1 common metadata sector [24].

4.3.3 Interleaved Metadata Sectors

Metadata [bytes]	**Space** [%] 512B sector	**Space** [%] 4096B sector
4 (32 bits)	0.78	0.10
16 (128 bits)	3.03	0.39
28 (224 bits)	5.26	0.68
32 (256 bits)	5.88	0.78
48 (284 bits)	9.09	1.16
64 (512 bits)	11.11	1.54
80 (640 bits)	14.29	1.92

Table 4.6: Space for per-sector metadata.

Our solution combines the use of a device integrity profile with a metadata per-sector stored independently in dedicated sectors. The presented sector size is configurable. We can use the same size as a native device, but we can also increase the presented sector size (atomicity is then ensured by journaling, as described in Section 4.3.4). The combined data and metadata are then presented to the block layer as a new virtual device with a specific integrity profile (software-emulated DIF-enabled device).

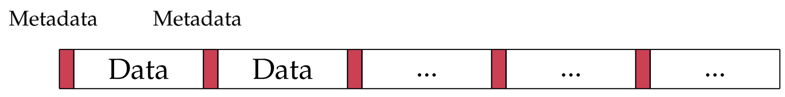

Figure 4.3: Interleaved metadata sectors.

The metadata are stored in special sectors that are interleaved with data sectors. According to the required metadata size, one sector contains a meta-

data store for several consecutive sectors. An illustration of this layout is provided in Figure 4.3. Required space examples are illustrated in Table 4.6. The interleaving format easily allows to resize block device later (in specific steps according to interleaving parameters).
The required size of additional sectors is calculated as follows:

$$TagsPerSector = \lfloor \frac{SectorSize}{TagSize} \rfloor$$
$$Tags_{sectors} = \lceil \frac{Data_{sectors}}{TagsPerSector} \rceil$$

The use of storage is optimal (no wasted space) when the sector size is a multiple of the tag size. Additional metadata space (part of a device used for metadata) is calculated as

$$100 \frac{Tags_{sectors}}{Data_{sectors} + Tags_{sectors}} \%.$$

4.3.4 Recovery on Write Failure

A device must provide atomic updating of both data and metadata. A situation in which one part is written to media while another part failed must not occur. Furthermore, metadata sectors are packed with tags for multiple sectors; thus, a write failure must not cause an integrity validation failure for other sectors.

A metadata-enabled device must implement a data journal that provides a reliable recovery in the case of a power failure. The journal can be switched off if an upper layer provides its own protection of data.

In some specific situations, a journal could provide unintended additional information about the encrypted device (such as last written data offsets or old data content of unfinished sector writes). A journal requires additional storage and decreases performance on write (data are written twice). The journal size is generally just a fraction of the device capacity.

4.4 Linux Kernel Implementation

We split the solution into *storage* and *cryptographic* parts. The former provides an area for per-sector metadata for commercial off-the-shelf storage devices. The latter extends disk encryption by cryptographically sound integrity protection. For key management, we extended the LUKS encryption system [11] to provide support for our integrity protection.

Our approach is based on the existing block layer integrity profile infrastructure [23] and is fully contained in the device-mapper kernel subsystem.

The only condition is that the Linux kernel must be compiled with support for the block integrity profile infrastructure (`CONFIG_BLK_DEV_INTEGRITY` option).

The implementation is fully contained in the device-mapper kernel subsystem and consists of a new *dm-integrity* target that provides a virtual device with a configurable integrity profile and an extension of *dm-crypt* for using authenticated encryption. Our implementation has been available in the mainline Linux kernel since version 4.12 [95].

The principle of the device-mapper architecture is that separate functions are combined by stacking several devices, each with a specific *target* that implements the needed function.
For active integrity-enabled encryption, there are three stacked devices:

- *dm-crypt device* (encrypts and authenticates data),
- *dm-integrity device* (provides per-sector metadata) and
- *an underlying block device* (disk or partition).

The top-level device is available to a user and can be used directly by an application or an arbitrary filesystem. The *dm-integrity* device can also operate in a standalone mode. In this case, the *dm-crypt* device is omitted.

4.4.1 The dm-integrity Module

The implemented *dm-integrity* device-mapper target creates a virtual device with arbitrary-sized per-sector metadata over the standard block device and presents it to the system as a device with a specific integrity profile. The driver implements an optional data journal.

The *dm-integrity* target can operate in two modes:

- as a provider of per-sector metadata for the upper device (as a *dm-crypt* target) or
- in standalone mode, where it calculates and maintains basic data integrity checksums itself.

At the device initialization step, the integrity target takes the underlying device and tag size and then writes the superblock to the device. The following activations then just read data from the superblock. An activated device creates a new integrity profile named `DM-DIF-EXT-TAG`.

The additional metadata are produced by the driver on top of the integrity module (integrity module only processes metadata, it does not allocate them). For reads, it combines the data and metadata sectors and submits them to the upper layer. For writes, it splits data and metadata and submits new I/O requests to the underlying device.

4.4.2 The dm-crypt Extension

The *dm-crypt* module is a Linux in-kernel implementation of FDE. To support authenticated encryption, we implemented the following parts:

- new parameter and configuration table format for integrity protection AEAD specification,
- interface to the underlying I/O integrity profile (provided by *dm-integrity*),
- processing of requests specified in Section 4.4.3,
- new random-IV generator and
- configurable encryption sector size.

All cryptographic primitives are provided by the kernel cryptographic API. This provides us with all the hardware support and acceleration available on a particular hardware platform.

The composed AEAD mode requires an independent key for integrity; thus, the activation requires that the userspace provides a key that is internally split into encryption and integrity parts.

When an integrity check fails, the entire I/O request is marked as failed with the specific error code (EILSEQ) and returned to the block layer. To achieve availability of data, a user can then stack redundant storage mapping (usually RAID) above such an integrity protected device; this layer then should react to a detected error by redirecting the operation to another redundant device.

The reality in current Linux RAID implementation is that it quietly amplifies silent data corruption, as experiment with stacking dm-integrity device below RAID shows [96]. Here the additional authenticated layer actually stopped RAID mechanism from working with corrupted data.

4.4.3 Sector Authenticated Encryption

To perform an authenticated encryption operation over a sector, we define the format of a sector authentication request (Table 4.7). The additional data (AAD) for our request contain the sector number and the IV. Such a request detects a sector misplacement and a corrupted IV. The request definition is compatible with the IEEE 1619 storage standard [80, 97].

AAD		**DATA**	**AUTH**
authenticated		authenticated + encrypted	**TAG**
sector	IV	data in/out	tag

Table 4.7: AEAD sector authentication request.

4.4.4 Random IV

We store persistent IV data with the additional per-sector metadata. With the available extra metadata space, we define a new *random-IV*. The random-IV regenerates its value on every write operation by reading it from a system Random Number Generator (RNG). The random-IV length should be at least 128 bits to avoid collision (expecting that an attacker has the capability to record all previous Initialization Vectors and sectors).

Since stored values are visible to an adversary, the random-IV cannot be used for modes that require unpredictable IVs.

Some systems could be slow if the RNG is not able to produce this amount of random data. However, our tests in Section 4.5 with a recent kernel show that the performance of the system RNG does not cause any noticeable problems. The only limitation of the random-IV is that during the early boot (before the RNG is initialized), the system must avoid writes to the device. Systems typically boot from a read-only device, and the RNG is properly seeded during this phase and becomes fully operational; thus, it is not an issue.

For the XTS mode and additional authentication tag with a random IV, we can improve change propagation as the entire sector is always pseudorandomly changed on each write, independently of the plaintext data content (as illustrated in Figure 4.2). The random IV must not be used with length-preserving XTS mode (without authentication tag) because in this mode decryption no longer depends on sector position (sector relocation attacks are possible).

4.4.5 Formatting the Device

The integrity tag is calculated for all sectors, including sectors that are not in use. On the initial activation, the device has to recalculate all integrity tags; otherwise, all reads fail with an integrity error. Such an integrity error can occur even on the initial device scan (newly appeared device is automatically scanned for known signatures).

The initial device formatting is quite time consuming, but fortunately, it must be done only once during the disk initialization phase. Disk encryption tools should perform this step automatically. We implemented such a function in the LUKS [11] cryptsetup tool.

4.5 Performance

We present three simple performance benchmarks to investigate the usability of our solution. The first benchmark is a simple benchmark for linear access to the device (read or write), the second is a synthetic I/O benchmark with interleaved reads and writes, and the third is a C-code compilation time benchmark with the underlying filesystem backed by our integrity protected device.

We used the mainline kernel 4.17 [95] with additional AEAD modules for AEGIS and MORUS [98] and cryptsetup tool [11]. The tests ran on a basic installation of the Fedora Linux, were repeated five times, and the arithmetic mean with the standard deviation (in the format *value* $\pm SD$) is then presented. All cryptographic hardware acceleration modules were loaded.

Encryption and Integrity Mode	**Metadata** IV [bytes]	TAG [bytes]
NULL cipher: no encryption, no integrity	-	-
no encryption, CRC32 integrity	-	4
AES256-XTS-plain64, no integrity	-	-
AES256-XTS-random, no integrity	16	-
AES256-GCM-random, AEAD integrity	12	16
AES256-XTS-random, integrity HMAC-SHA256	16	32
ChaCha20-random, integrity Poly1305	12	16
AEGIS128-random, AEAD integrity	16	16
AEGIS256-random, AEAD integrity	32	16
MORUS640-random, AEAD integrity	16	16
MORUS1280-random, AEAD integrity	16	16

Table 4.8: Tested encryption modes and metadata size.

The hardware configuration represents a typical desktop configuration that we expect our solution would be used and is described in Table 4.9.

Tests on more older hardware (Lenovo laptop, see Table 4.10) was part of the abridged publication [4]. These tests used kernel 4.12 and do not include new AEAD modes implementations (not available at that time).

Hardware Configuration #1
Intel Core i7-4790 3.60 GHz CPU (4 cores)
CPU has AES-NI, SSE3 and AVX2 instructions
32 GB RAM DDR3 1600 MHz
Samsung SSD 850 Pro 512 GB, 2.5″ Drive

Table 4.9: Hardware configuration for desktop test.

Hardware Configuration #2
Lenovo x240 laptop
Intel Core i7-4600U 2.10 GHz CPU (4 cores)
CPU has AES-NI and SSE3 instructions
8 GB RAM Micron DDR3 1600 MHz
Toshiba THNSFJ256GCSU 256 GB SSD, 2.5″ Drive

Table 4.10: Hardware configuration for older laptop test.

4.5.1 Linear Access

We ran tests that measured the speed of reads and writes to the entire device. The I/O block size was fixed to 4 kB (presenting a page cache I/O that always produces a page-sized I/O as the minimum).

We ran the tests both with the *dm-integrity* data journal and without the journal (this simulates a situation when a journaling is already present on an upper layer). The measured data are presented in Figure 4.4 and Figure 4.5.

The measured data show that the overhead of the length-preserving encryption in this scenario is almost negligible. The only visible overhead is for the write operation.

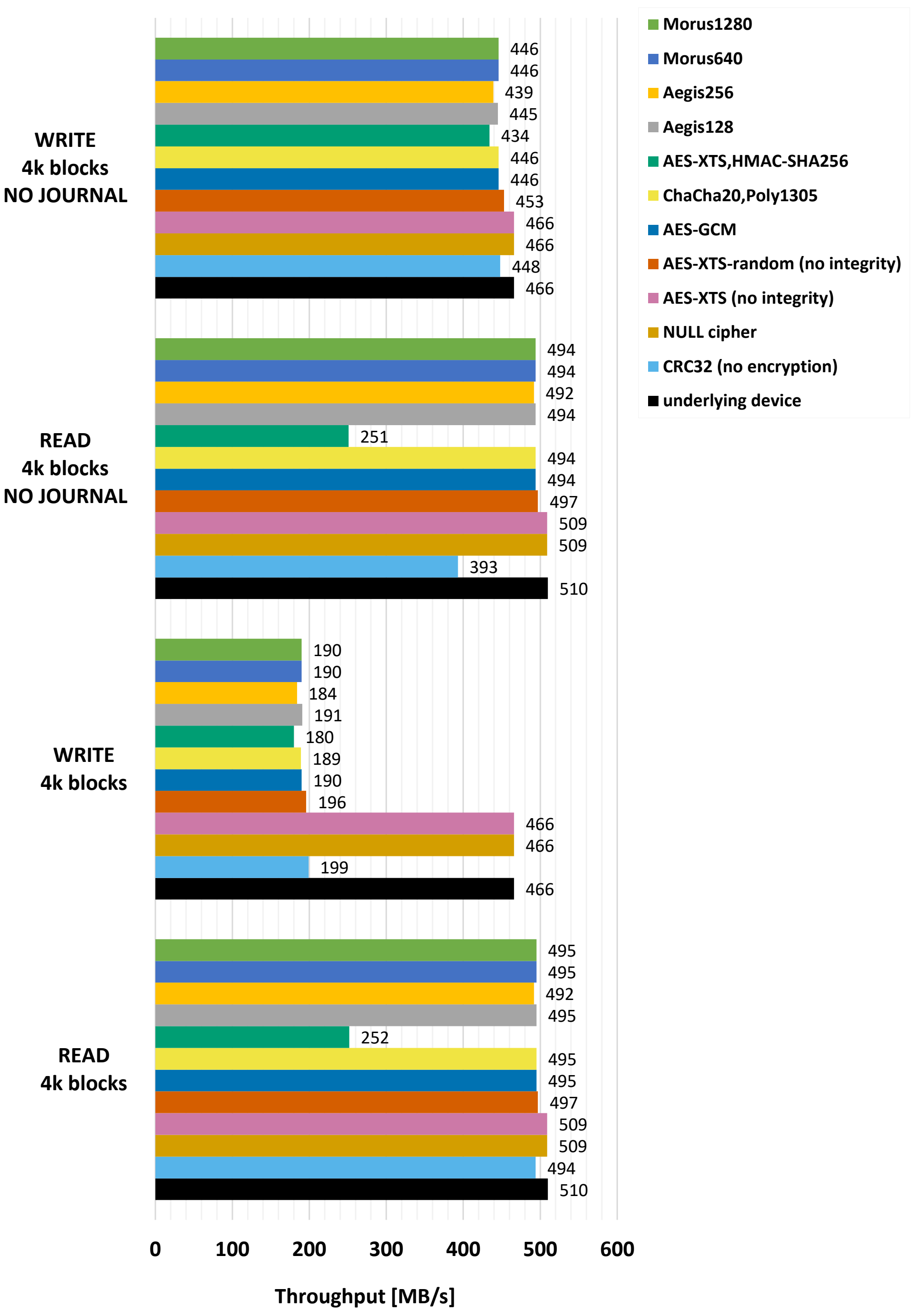

Figure 4.4: Device throughput on a solid-state disk (desktop).

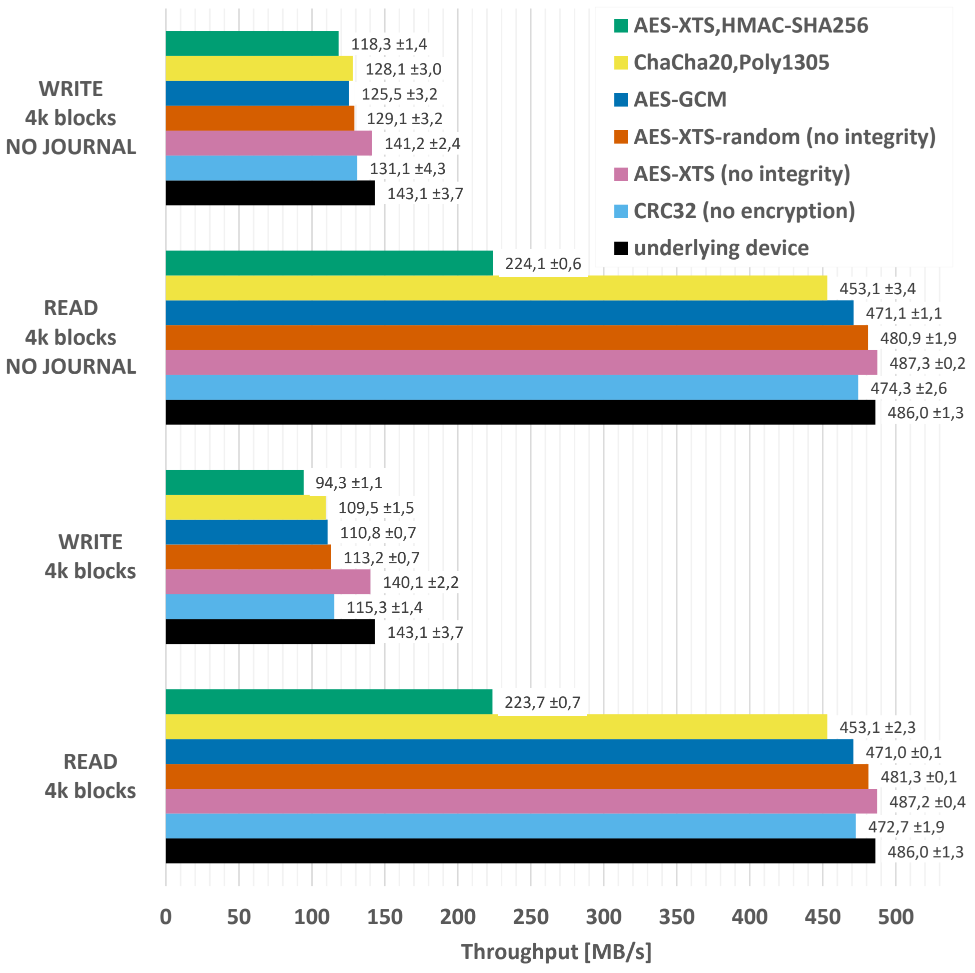

Figure 4.5: Device throughput on a laptop solid-state disk.

Disabling the data journal has an effect only for write operations. The most visible output is the overhead of additional metadata processing. The overhead of cryptographic data integrity processing is visible for all authenticated modes.

4.5.2 Random I/O Throughput

We simulated workload with the *fio* [99] (Flexible I/O Tester) utility. We used a *fio* profile simulating mixed reads (70%) and writes (30%) that generates random data access offsets with I/O operations of the 8k block in size. The test uses 16 parallel jobs, asynchronous I/Os and runs for at least 100 seconds.

The exact parameters of the *fio* command were as follows:

```
fio --filename=<DEVICE>
 --direct=0 --bs=8k --rw randrw:16
 --refill_buffers --norandommap
 --randrepeat=0 --ioengine=libaio
 --rwmixread=70 --iodepth=16
 --numjobs=16 --runtime=100
```

Encryption mode and Authentication mode	**SSD [seconds]**	**SSD (no journal) [seconds]**
underlying device	1356,2 ±40,9	
CRC32 checksum	1343,4 ±13,9	1346,6 ±31,7
NULL cipher	1365,5 ±39,9	
AES-XTS-plain64	1341,6 ±11,4	
AES-XTS-random	1352,9 ±26,8	1335,1 ±24,1
AES-GCM-random integrity AEAD	1347,8 ±26,4	1358,2 ±55,8
AES-XTS-random integrity HMAC-SHA256	1343,8 ±7,3	1330,0 ±19,2
ChaCha20-random integrity Poly1305	1364,5 ±38,0	1370,5 ±39,2
AEGIS128-random integrity AEAD	1353,5 ±59,6	1339,7 ±47,2
AEGIS256-random integrity AEAD	1342,7 ±35,4	1363,7 ±36,3
MORUS640-random integrity AEAD	1355,4 ±16,9	1361,1 ±15,8
MORUS1280-random integrity AEAD	1344,5 ±18,7	1340,6 ±21,5

Table 4.11: Compilation time test for desktop test.

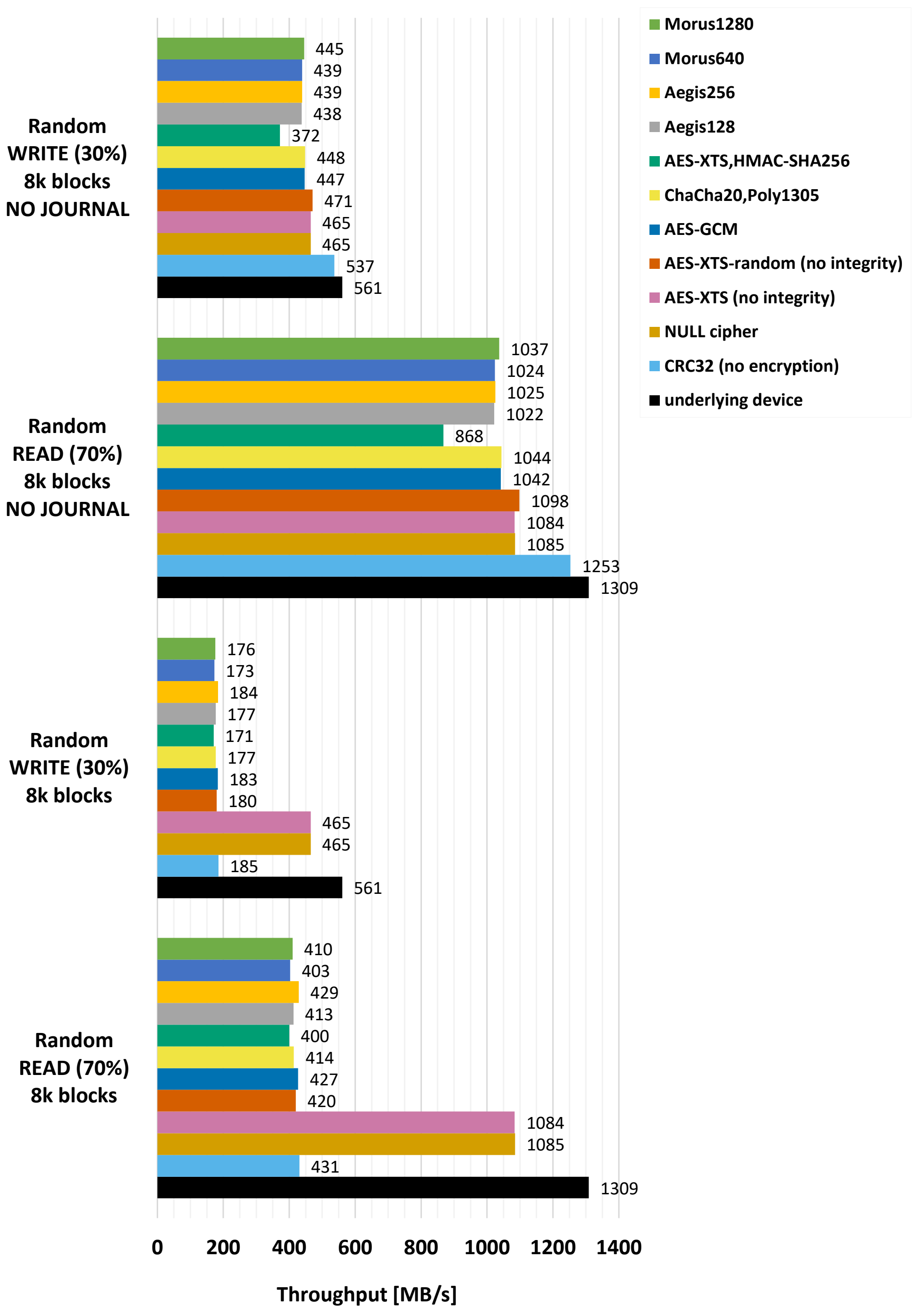

Figure 4.6: *fio* simulated load on a solid-state disk (desktop).

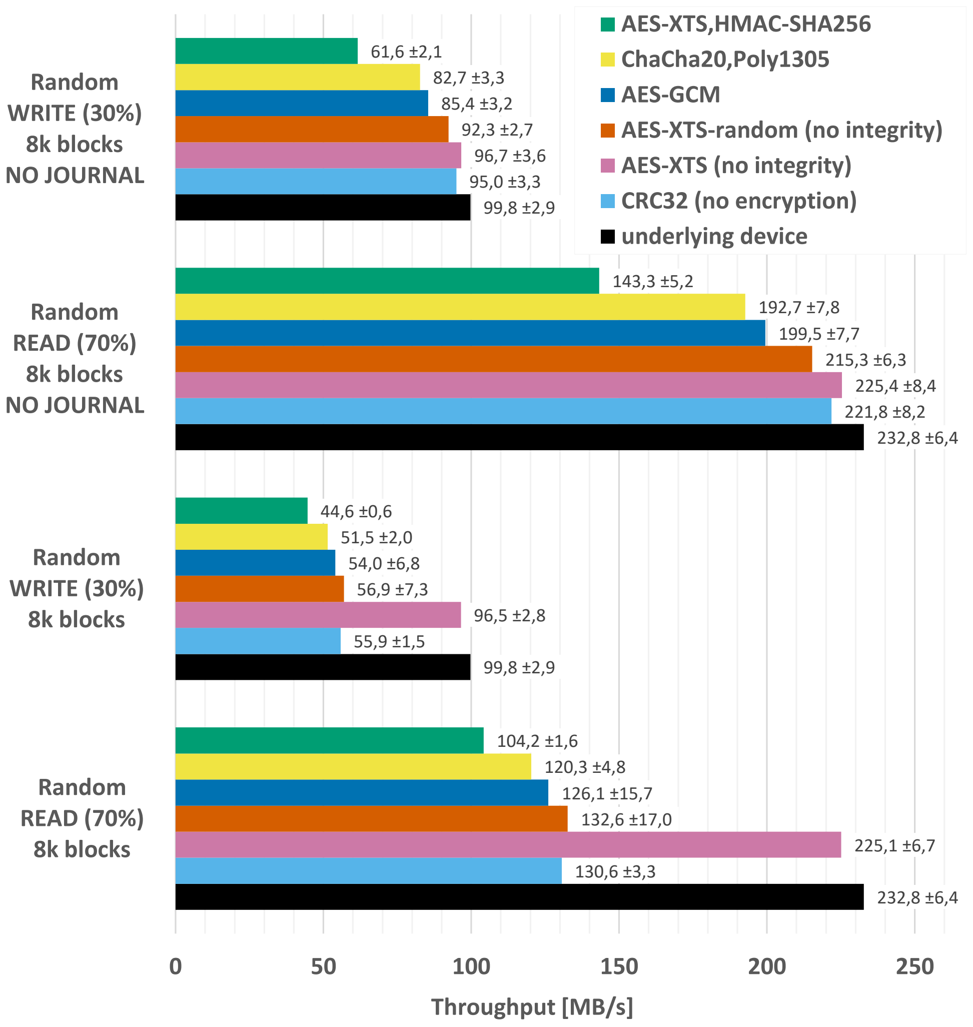

Figure 4.7: *fio* simulated load on a laptop solid-state disk.

This test should represent the worst case scenario in our comparison. The measured results are presented in Figures 4.6 and 4.7. We can see that a journal has a small effect in this scenario; the major visible slowdown is in the additional metadata processing.

4.5.3 Compilation Time Test

This test simulates a very simple application workload and measures the overhead of an integrity-protected storage in a loaded system (common laptop use).

The test unpacks mainline Linux kernel sources, flushes memory caches and measures the compilation time of a specific kernel configuration (compilation uses all available CPU cores in parallel). We used the *xfs* filesystem with default parameters over our device in this scenario.

The measured times are presented in Table 4.11. We also measured the same encryption mode with the *dm-integrity* journal switched off.

From the measured numbers, we can see that the slowdown varies in all cases under 10%. Neither data journal nor authenticated encryption presents a significant slowdown with this type of workload.

4.6 Conclusions

Our goal was not only to show that the combination of confidentiality and cryptographic data integrity protection is possible at the FDE layer, but also to highlight the need for proper cryptographic data integrity protection in general. We focused on existing COTS devices without any specific hardware requirements. Almost all laptops are currently delivered with an SSD. In this scenario, the performance evaluation shows that our data integrity protection is usable for these systems.

Our solution is based on existing concepts only, even though the proposed authenticated encryption request format is designed to be compliant with the IEEE 1619.1 standard [97]. This helps the deployment of our integrity protection to systems that require security certifications.

The *dm-integrity* module provides a generic solution for additional metadata per sector by emulating HW-based DIF profiles with a software-defined function. The price to pay is decreased storage performance and storage capacity, but we believe this is balanced by a zero-cost investment in required hardware (our solution just uses existing COTS devices).

Our practical implementation has shown that the performance is in many situations already acceptable for recent systems with SSDs and could be in a limited scope also for devices with rotational drives.

The optimization of the *dm-crypt* FDE implementation in Linux took several years until it achieved today's performance; thus, we expect that the combination with the *dm-integrity* will follow a similar path.

The extension to *dm-crypt* FDE is also, as it was a major design goal, algorithm-agnostic. The configuration of another encryption mode is just a configuration option.

As a joined effort [98] with our work, an implementation of AEGIS [85] and MORUS [86] authenticated modes for Linux kernel was finished, and we can say that the authentication encryption algorithm of choice today is one of these CAESAR [84] finalists. Even if there is a better AEAD mode introduced later (like a new GCM-SIV [87, 100]), it can be easily used with our solution once it becomes available.

Both configurable metadata per-sector and encryption upgrades of algorithms are the major advantages to hardware-based encryption solutions, where any reconfiguration during their lifetime is almost impossible.

All code presented in this work is released under the open-source GPL2 license and has been included in the Linux mainline kernel since the version 4.12 (new AEAD modes since the version 4.18) and userspace cryptsetup tool [11] since the version 2.0.0.

4.6.1 Future Work

Integrity protection is often used on a higher layer than device sectors [101]. The *dm-integrity* feature could be used later even for another task such as an application-level integrity protection (application could send its own data to be stored on per-sector level) of storing additional Forward Error Correction codes (the storage could then not only detect integrity problems but also fix basic random data corruptions).

Additionally, the same principle can be applied to new storage device types, such as persistent memory (or any byte-addressable persistent storage), where we can easily increase the size of the additional authentication tag (in principle, we can use virtual sector of any size). In this case, the *dm-integrity* layer can be omitted (the atomicity of data and metadata writes is provided by the storage itself), while the *dm-crypt* cryptographic part of the solution remains the same.

5 Passwords-Based Key Derivation for Disk Encryption

Passwords usually contain little entropy, exhibit poor randomness characteristics and cannot be directly used as encryption keys. Password-based key derivation functions (PBKDF) are a common solution to this problem.

One specific use of PBKDF is in full disk encryption (FDE) applications, where it is used to derive a decryption key from a user-provided passphrase.

As the only standardized form of PBKDF, *PBKDF2* [55] is designed to be a strictly sequential function that allows to set a number of iterations. The memory footprint of PBKDF2 is constant and small, which allows very efficient brute-force attacks to be performed on GPU or ASIC systems [56]. The only sufficiently scrutinized alternative was *scrypt* [58].

The problem with lack of algorithms that can effectively defeat massively parallel attacks led to the *Password Hashing Competition* (PHC) initiative [7]. The PHC was run by an independent panel of experts and the main goal was to identify and analyze new password hashing schemes. While PBKDF is not the main focus of the competition, some of the selected candidate functions are usable also as a PBKDF and could replace the frequently-used PBKDF2 in the future.

In this chapter, we evaluate suitability of PHC second round finalists for use in a KDF application with focus on disk encryption key management. The main contributions are:

- a definition of requirements of PBKDF for the FDE environment,
- description of common building blocks in submitted algorithms,
- measurements based on a real FDE use case,
- fixes of several implementation issues discovered by our tests [9].

5.1 KDF and Disk Encryption Environment

FDE environment is not always friendly to all security features of a password hashing function.

Firstly, a disk encrypted on one machine must be often read (decrypted) on another machine where the processing power can be significantly different (not only in the sense of speed and number of CPU cores, also it can be a completely different computer platform, where some operations are slower or where an important acceleration feature is missing). For users, it is probably not a big problem if unlocking of a disk takes long time (within reasonable limits), but it can be a real usability problem if unlocking is not possible at all.

Secondly, a decryption environment can be very limited (an example is unlocking of a system disk in a bootloader, where we cannot use process or thread management or where strict memory limits are imposed).

While we are focusing on desktop or server platforms, requirements and even evaluation of algorithms can be directly applied to smart phones or tablet computers. These FDE-capable devices have several CPU cores and a decent amount of memory. Requirements are also applicable to key management in the area of non-volatile RAM (persistent memory) encryption.

5.2 PBKDF Requirements for a Disk Encryption Application

If a password-based key derivation function is used in a FDE environment, we propose the following requirements:

1. A key derivation function is a deterministic algorithm defined by the choice of a pseudo-random function [53]. It also must be a one-way, collision-resistant function.

2. It must be able to use resources in such a way that it allows users to unlock a device in an acceptable time and using acceptable resources. Resources can be the time of calculation, the level of parallelism or the amount of memory required.

   Overly long unlocking time can lead to a risk that users switch the encryption off to speed up their system start. The effort to harden security then would be counterproductive.

3. Possible configurations should cover high variety of systems from embedded systems to high-end storage servers.

4. The level of parallelism must not block the use of this function in systems where parallel processing is not available (there will be a time penalty, but the function should be still usable).

   An example for requirements 3 and 4 is encryption of portable disk drives. Such a disk should be readable on the majority of systems. Some systems could be embedded or old devices without parallel processing capability. Unlocking time will be much longer on such systems, but users will still be able to access the encrypted data.

5. It should have predictable run-time calculations (to benchmark an algorithm on a particular machine and select attributes that fit best).

This requirement copies current work-flow of some existing disk encryption systems that benchmark the system to achieve best approximation of the unlocking time. This is mainly useful for encryption of a boot or fixed disk where the disk is always unlocked on the same system.

6. Configured resources should not significantly influence each other. For example, increasing computational time should not significantly affect memory requirements.

7. Algorithms must be able to take input (password) of any length.

8. Maximal output length (keys) should not be significantly limited, so a function can derive multiple keys from a single input.

   While there are options to seed another algorithm (like a stream cipher) with the fixed key, it is preferred to have one standardized key-derivation algorithm instead of chaining several different key generators. However, this requirement can add another final step (like iterated hashing) to the key-derivation algorithm.

9. Any change in running time caused by the change of input password length should be marginal. The running time must not provide information about password attributes (such as length).

10. Used cryptographic primitives should be upgradable and should be available in common cryptographic libraries.

11. Cryptographic algorithms should not be hard-coded in the function's design.

    The lifetime of an encrypted disk is usually years to decades. It should be possible to replace an underlying primitive even during its lifetime if a security problem is uncovered. Upgradable components also help to fine-tune for specific needs, typically for certified systems where only a limited set of algorithms is allowed. The downside of allowing interchangeable components is that users will sometimes select less secure combinations.

12. Algorithms must not be patent encumbered.

5.3 KDF Building Blocks

A key derivation function can be constructed either as a completely new cryptographic primitive or more often it can be built with existing primitives as building blocks. Usually, it is a combination of both, where the newly designed part adds a memory requirement property while the computationally intensive part is based on utilizing existing cryptographic primitives.

The major problem with newly designed cryptographic primitives is that there is no adequate security analysis. If we are using a thoroughly analyzed hash function, security of a KDF can be derived from the security of the hash function. Since FDE is not a short-lifetime problem (some encrypted devices are already used for a decade without reformatting) the preference is to use proven cryptographic primitives in the design.

5.3.1 Cryptographic Primitives

- *Hash* functions or *block ciphers* are commonly used. For hash functions, it is usually one of the SHA family or a sponge function [102] like BLAKE (often the 64-bit optimized variant BLAKE2b [103]). Only a single property of a function can be used (for example compression).

- *Reduced cryptographic primitives* are hash functions or block ciphers with a reduced number of rounds. These are often used where performance is an issue and repeated call of a full-round function is slowing down the whole algorithm above acceptable limits. In general, this solution requires new security analysis similar to the situation when a new cryptographic primitive is used.

5.3.2 Concepts to Utilize Resources

The main intent is to complicate password search attacks.

- *Strictly sequential processing* prevents an attacker from using a massively parallel system to speed up function run.

- *Memory-hard function* [58] is the concept of using as much memory space as possible for a given number of operations.

  A *sequential memory-hard function* adds a property that it is not possible to achieve a significantly lower cost if a parallel function is used instead.

- *Paralellization* denotes the use of available parallel capabilities in a function (for example utilize common multi-core CPUs).
- *Server relief* is the possibility of delegating part of computation to another system (for subproblems where the system in question does not need to be trusted). This property is not important for FDE (disk unlocking often runs in an environment where it cannot easily communicate with other systems).
- *Client-independent upgrade* is the possibility of increasing resource usage (cost) without providing a password (upgrade password hashes without requiring to enter all related passwords). This property is not usable for a FDE because a hash (a secret key) is not available for an upgrade (without knowledge of the user input).

5.3.3 Ingredients

Attributes influencing the use of resources are often called ingredients.

- *Salt* is a unique sequence of random and public bits added to a user password before processing. The goal is to increase the cost of brute-force attacks (an attacker cannot build a common dictionary and has to build a table for every case separately).
- *Garlic* configures required memory cost. In reality, increasing the memory requirements also increases the running time (memory accesses take time).
- *Pepper* increases the time of processing by making some of the bits of salt secret, so the password-checking algorithm must verify more variants.

5.3.4 Processing Unlimited Input and Output

If a function is designed using an underlying cryptographic primitive with limited block size (a hash function usually dictates the block size), the input has to be compressed to the requested block size, while preserving entropy.

One known problem in PBKDF2 is that for the input exceeding the internal block size, the hash of the input is used instead. That leads to simple collisions (an input behaves the same as a hash of itself). When hashing is used, it should avoid such a collision problem by design (usually by adding some attribute value in initial hashing).

For unlimited output (used to derive multiple keys from a single password), hashing can be used as well. It is usually an iterative form of hashing (sequentially applying a hash function, possibly with the help of a counter).

5.4 PHC Candidates as KDF Algorithms

The following list is based on the PHC second-round candidates and proposed algorithms (which were added in later rounds). The main focus here is to compare them with our requirements described in Section 5.2.

A more detailed survey of candidates is available in [104], but it does not contain some functions introduced later (Argon2 or Catena instances).

5.4.1 Argon

There are two different algorithms *Argon* [105] (based on the original submission) and *Argon2* in two variants *Argon2d* and *Argon2i*.[1] Argon is designed mainly to maximize the cost when used on non-Intel 64-bit architectures. Authors suggest to use Argon2i for KDF applications.

Argon after the initial hashing phase (BLAKE2b) and initial permutation round (based on 5-round reduced AES-128 with a fixed key) creates a memory layout (a matrix of blocks, where size depends on memory cost). The round part then combines shuffling the rows and columns of the matrix, repeated according to the time cost parameter. The finalization part uses XOR of the matrix blocks for an iterated hashing step that produces an output of requested length.

Argon2 initially hashes the input and then fills the memory (matrices of blocks according to a parallel property). Every matrix is computed using a compression function (based on BLAKE2b round function). Input block indices are either selected randomly (in *Argon2d*, data-dependent version) or pseudo-randomly (in *Argon2i*, data-independent version). The whole procedure is repeated based on the time cost parameter. The final step is to use XOR applied to the matrix columns to calculate the final block. The output is generated from the final block using iterative hashing.

5.4.2 Battcrypt

Battcrypt (Blowfish All The Things) [107] constructs a memory-hard function using the Blowfish block cipher. Initially it hashes the password and salt using SHA-512 and uses the output as a Blowfish cipher key (with zeroed

1. Later hybrid version *Argon2id* was introduced as a response to security analysis [106].

IV) in CBC mode. The memory is organized in blocks (the number depends on the memory cost parameter) and is initialized using Blowfish encryption. The work phase (repeated, based on the required time cost) then traverses the memory and modifies it using XOR and Blowfish CBC encryption (the function is data-dependent). The output is then produced using one SHA-512 hash operation. There is also a separate KDF mode, where a variable output is provided by additional iterated hashing.

5.4.3 Catena

Catena [108] is a generic password hashing framework. The submitted version presents two instances of this framework, *Catena-Dragonfly* based on a bit-reversal graph and *Catena-Butterfly* based on a double-butterfly graph. Catena uses a reduced-round BLAKE2b function.

Catena consists of an initialization, where a password is hashed and one call of the *flap* function is performed. The work phase then iteratively calls the flap function and the hash function according to the garlic parameter. The flap function consists of sequential initialization of memory and applying a function that provides random memory accesses, followed by a call of a memory-hard function. For the given instances, *SaltMix* function (accessing memory according to provided salt) is used for the first part and bit-reversal hashing or double-butterfly hashing for the second memory-hard part. For a KDF, the algorithm uses *Catena-KG* mode that adds an iterative hashing step to producing a variable-length output.

5.4.4 Lyra2

Lyra2 [109] is a password hashing function based on the concept of cryptographic sponge [102] with duplex construction (operating in stateful mode). Lyra2 is constructed as a strictly sequential algorithm. The internal state of the sponge is never reset between algorithm phases. Except for the final wrap-up phase, a reduced-round sponge is used.

The initial password is absorbed by the underlying sponge (BLAKE2b). Then a matrix of blocks is constructed in memory (setup phase). The number of columns in the matrix is hard-coded in the algorithm while the number of rows is defined by the memory cost parameter. The memory is filled up using the sponge duplex function, XOR, and bit rotation. Next phase (wandering) is the most time consuming phase and is similar to the setup phase with an additional data-dependent operation and repetition based on the time cost parameter. Finally, the wrap-up phase contains one full round of sponge absorb over one fixed final memory cell. The output is generated using the sponge squeeze function.

5.4.5 Yescrypt

Yescrypt [110] is an extension of the *scrypt* [58] algorithm. The original *scrypt* design already supports both time and memory cost.

Initial PBKDF2 round (one iteration with SHA-256) processes the input password and generates input memory blocks (depends on parallel cost). Blocks are processed by a sequential memory-hard function called *ROMix*, based on the internal *BlockMix* function. The processed blocks are used as a salt for the final PBKDF2 one-round iteration step that produces output of requested size.

Candidate function	**Cryptographic primitive**	**Modification for KDF**	**Unlimited input**	**Unlimited output**
Argon	AES-128 (reduced) BLAKE2b	no	hashing	iterative hashing XOR on init
Argon2i	BLAKE2b (compression)	no	hashing	iterative hashing
battcrypt	Blowfish SHA-512	yes	hashing	iterative hashing
Catena	BLAKE2b (reduced)	yes	hashing	iterative hashing
Lyra2	BLAKE2b	no	sponge absorbing	sponge squeeze
yescrypt	Salsa20/8 SHA-256	no	hashing PBKDF2	PBKDF2

Table 5.1: Cryptographic primitives used in candidate functions.

Yescrypt extends scrypt using optional flags that can modify ROMix memory accesses from *write once, read-many* to *mostly read-write* operation. It also fixes usage of the initial PBKDF2 round (to avoid simple collisions) and allows to increase the processing time with the given memory cost (scrypt does not enable that independently). The parallelism present in scrypt on a high level is possible in yescrypt also inside the ROMix function. The

Salsa20/8 cipher (used in scrypt BlockMix) is partially replaced by a *parallel wide transformation function* that is intended to utilize 64-bit single instruction, multiple data (SIMD) computations.

5.4.6 Algorithms not Selected for Further Testing

MAKWA [111] is not designed as a memory-hard algorithm so it does not fit our criteria. The most interesting attribute is the introduction of delegation that means that a part of the processing cost can be delegated (offloaded) to external, untrusted systems.

Parallel [112] is a very straightforward function based on the underlying hash function (SHA-512). The design is comparable to PBKDF2 and does not allow to specify memory cost, and so it does not fit our needs.

POMELO [113] is designed as a new cryptographic primitive (not utilizing any existing cryptographic hash or cipher). It is built on the concept of a non-linear feedback function.

Candidate	**Notes**
Argon	- design replaced with *Argon2*
	- uses reduced round *AES*
Argon2	- based on *BLAKE2*
	- need to use data-independent version *Argon2i*
battcrypt	- based on *Blowfish* only
	- KDF mode is separate
Catena	- possible use of reduced *BLAKE2*
Lyra2	- partial use of reduced *BLAKE2*
yescrypt	- compatibility with *scrypt*, adds complexity
	- requires *PBKDF2*
MAKWA	- not a memory-hard algorithm
Parallel	- not a memory-hard algorithm
POMELO	- new cryptographic primitive
Pufferfish	- uses own *Blowfish* implementation
	- *bcrypt* replacement only

Table 5.2: Notes on cryptographic primitive use.

Pufferfish [114] is designed to *extend life of bcrypt*. It rewrites the underlying Blowfish implementation to use 64-bit words to better utilize 64-bit platforms and provides for a variable length output by using iterative hashing (SHA-512). Considering the special Blowfish implementation and the narrow design goal, this algorithm was not selected for testing.

Table 5.2 summarizes the cryptographic primitives used in candidate functions based on description above and the requirements defined in Section 5.2.

Table 5.1 summarizes which cryptographic primitives are used in the described functions, whether code changes are needed for the KDF mode of operation and how unlimited input and output are processed.

For comparison, in the following tests we also include *POMELO*, even though the algorithm does not use any existing cryptographic primitives (our criteria prefer algorithms based on existing cryptographic primitives).

5.5 PHC Candidates Usability Tests

Tests were run on a 64bit Linux operating system (both kernel and userspace is fully 64bit), on two platforms

- X86-64: Lenovo X240 notebook with i7-4600U, 2.10GHz CPU and 8GB RAM with AES-NI and SSE instructions available (typical user notebook configuration),
- PPC64: PowerPC7 server in big-endian mode, 128GB RAM (for endianess and portability tests).

All algorithms are compiled without a parallel attribute to eliminate an advantage of algorithms that use internal parallel processing (all algorithms use only one thread). The following figures are based on X86-64 platform measurements.

5.5.1 Run-time Test

We ran a test that measures real memory use and run-time for the increasing memory or time cost parameter while the other parameters are fixed. The tests are described in more detail with complete source code in our test report repository [9].

Run time measurement used *clock_gettime*(*CLOCK_MONOTONIC*) function call on the Linux platform. The test run as a special forked process started for each test separately, tests are repeated for 5 times and arithmetic mean (for the time) or maximum (for the memory) of measurements is used. Algorithms that provides optimized variant are tested separately (only AES-NI and SSE variants). Because AES-NI and SSE instructions are not usable on PowerPC system, only reference implementations run (and compile) there.

The provided *PHC* interface (see Figure 5.1) does not allow to set parallel cost attribute at run-time.

```
int PHS(void *out, size_t outlen,          /* output */
        const void *in, size_t inlen,      /* input */
        const void *salt, size_t saltlen, /* salt */
        unsigned int t_cost,               /* time cost */
        unsigned int m_cost);              /* memory cost */
```

Figure 5.1: PHC candidates required C interface.

In the follow-up discussion to our tests [115], an alternative interface for parallel attribute testing was introduced. This interface slightly modifies the source code of the candidates (to expose the parallel attribute).

If the candidate provides an optimized variant, it is tested separately. Reference implementations are usually not optimized for speed but can be used to show how the algorithm will behave on a system that does not have required acceleration functions (AVX, SSE or AES-NI instructions in our case).

The overview plot of measurement of dependence of memory cost and run time is shown in Figure 5.2.

Name	**t_cost for 2x**	**t_cost for 3x**	**t_cost for 4x**	**t_cost for 5x**	**m_cost calculation**
Argon	2	3	4	5	x
battcrypt	0	1	2	-	$log_2(x) - 3$
Catena	2	3	4	5	$log_2(x) + 4$
Lyra2	2	3	4	5	$x/24$
POMELO	1	-	2	-	$log_2(x) - 3$
Pufferfish	-	0	-	1	$log_2(x)$
yescrypt	3	4	5	6	$log_2(x)$

Table 5.3: Parameters for number of rounds [116].

Figure 5.3 shows comparison based on the calculation of required memory and partially normalized by the number of rounds of underlying function calls is described. The memory and time cost parameters are calculated according to Table 5.3.

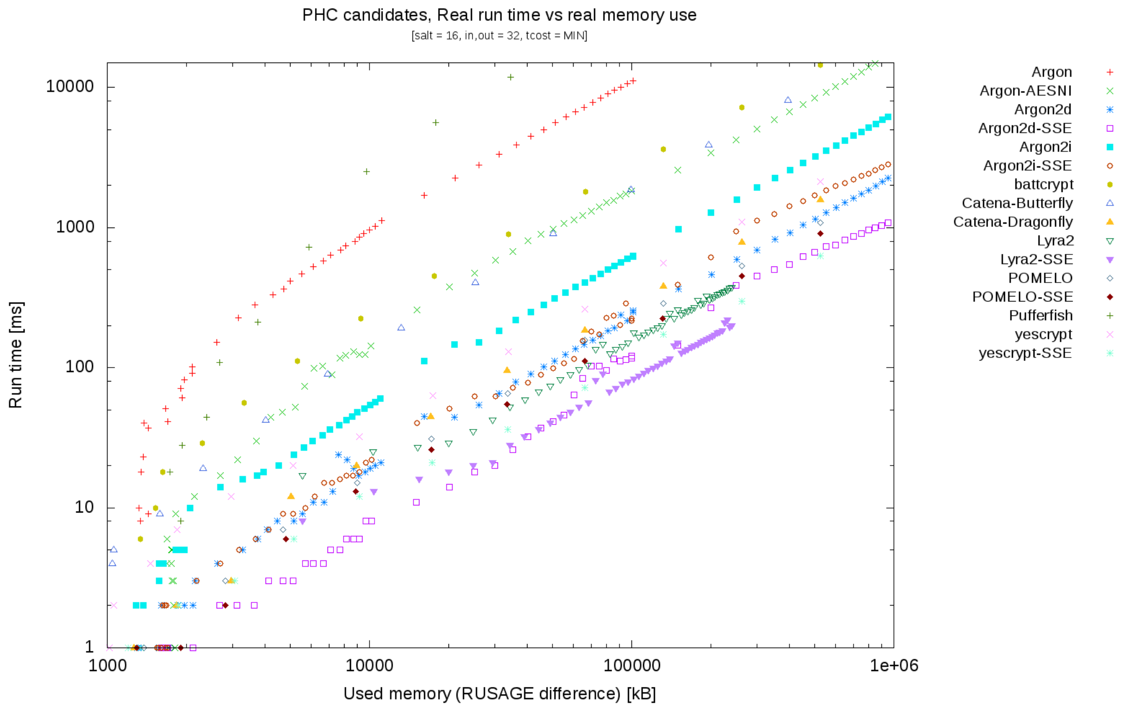

Figure 5.2: Real used memory and run-time while increasing memory cost parameter.

5.5.2 Specific Use Case Measurement

Some FDE systems already include machine benchmarks with the intention of fine-tuning KDF parameters to perform best on a particular machine. An example is to benchmark the iteration count for PBKDF2 in such a way that unlocking should take a specific time (on the same machine).

The following test tries to answer the question whether a function is able to provide usable parameters on the test machine for the given set of memory and time use conditions.

The test limits memory to 1 MB, 100 MB or 1 GB and also limits the runtime. The first run is the time minimum for the required memory limit, the second is an approximate 1 second run-time and the last one is run-time of approximate 20 seconds.

The selection of one second simulates *the default iteration time used in the Linux Unified Key Setup (LUKS) FDE system* [14], and 20 seconds represent *the limit that is usually acceptable for users when waiting for unlocking a device*.

Candidate	**Memory**	**Reference**			**Optimized**		
		min	∼1s	∼20s	min	∼1s	∼20s
Argon	1MB	✓	✓	✓	✓	✓	✓
	100MB	[11.6s] ✓	×	✓	[1.8s] ✓	✓	✓
	1GB	[119.1s] ✓	×	×	[18.0s] ✓	×	✓
Argon2i	1MB	✓	✓	✓	✓	✓	✓
	100MB	✓	✓	✓	✓	✓	✓
	1GB	[6.6s] ✓	×	✓	[2.9s] ✓	×	✓
battcrypt	1MB	✓	✓	✓			
	100MB	[3.6s] ✓	×	✓		n/a	
	1GB	[28.8s] ✓	×	×			
Catena Butterfly	1MB	✓	✓	×			
	100MB	[1.8s] ✓	×	✓		n/a	
	1GB	[35.6s] ✓	×	×			
Catena Dragonfly	1MB	✓	×	×			
	100MB	✓	✓	✓		n/a	
	1GB	[2.9s] ✓	×	✓			
Lyra2	1MB	✓	✓	✓	✓	✓	✓
	100MB	✓	✓	✓	✓	✓	✓
	1GB	[1.6s]✓	×	✓	✓	✓	✓
POMELO	1MB	✓	✓	✓	✓	✓	✓
	100MB	✓	✓	✓	✓	✓	✓
	1GB	[2.1s] ✓	×	✓	[1.8s] ✓	×	✓
yescrypt	1MB	✓	✓	✓	✓	✓	✓
	100MB	✓	✓	✓	✓	✓	✓
	1GB	[4.2s] ✓	×	✓	[1.2s] ✓	×	✓

Table 5.4: Ability to cover preset limits.

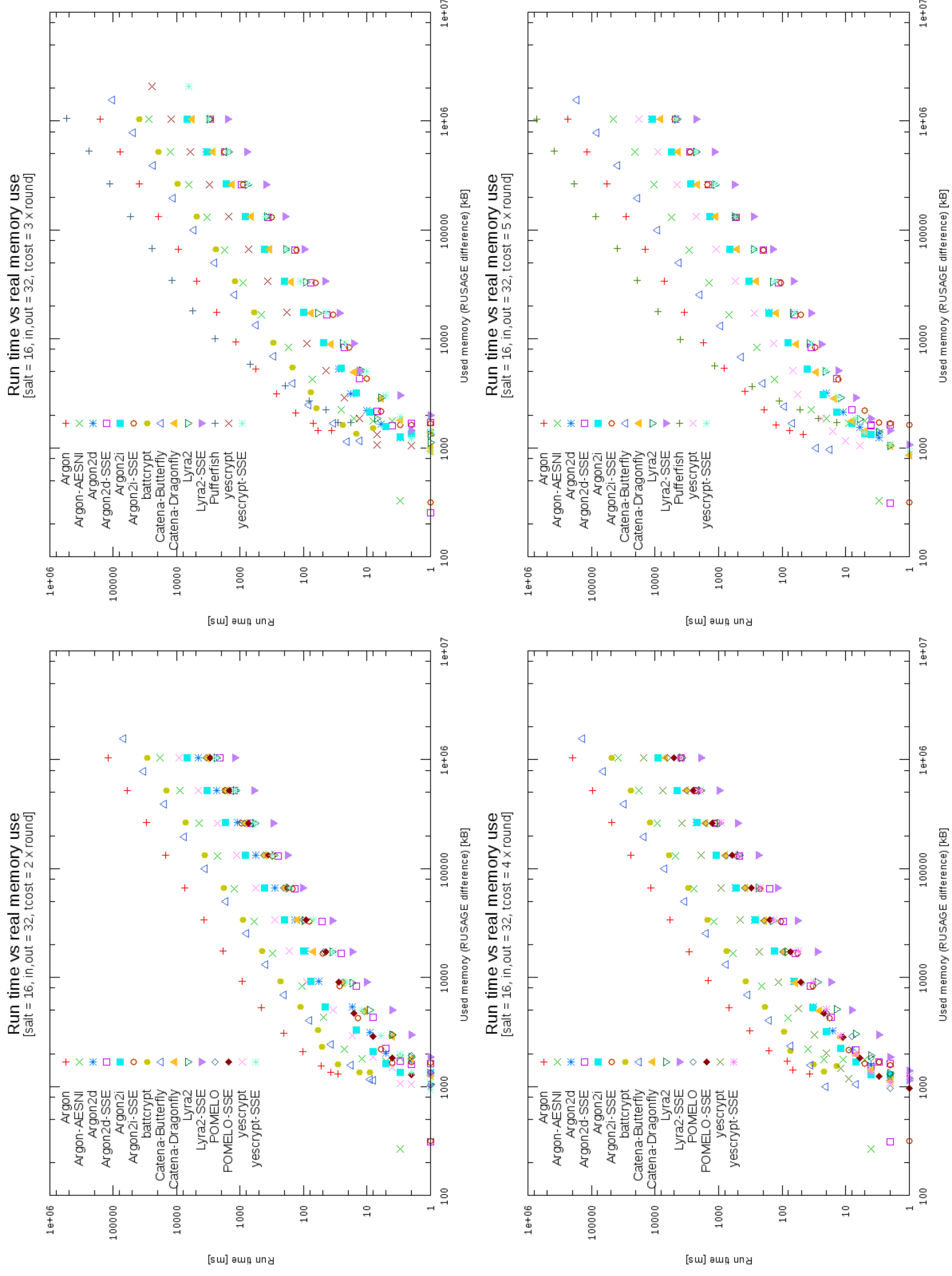

Figure 5.3: Real used memory and run time (normalized to performed rounds).

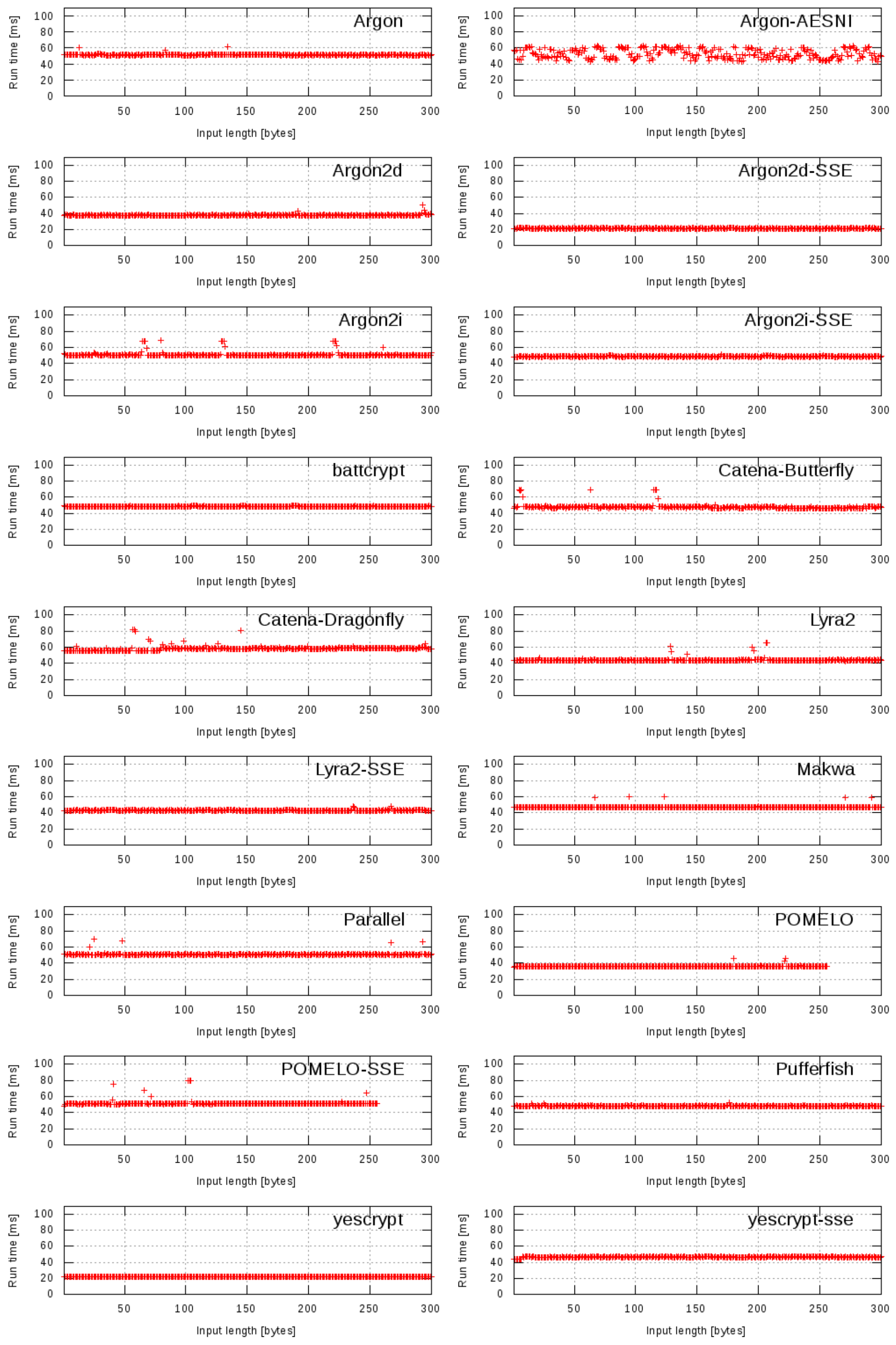

Figure 5.4: Input length test.

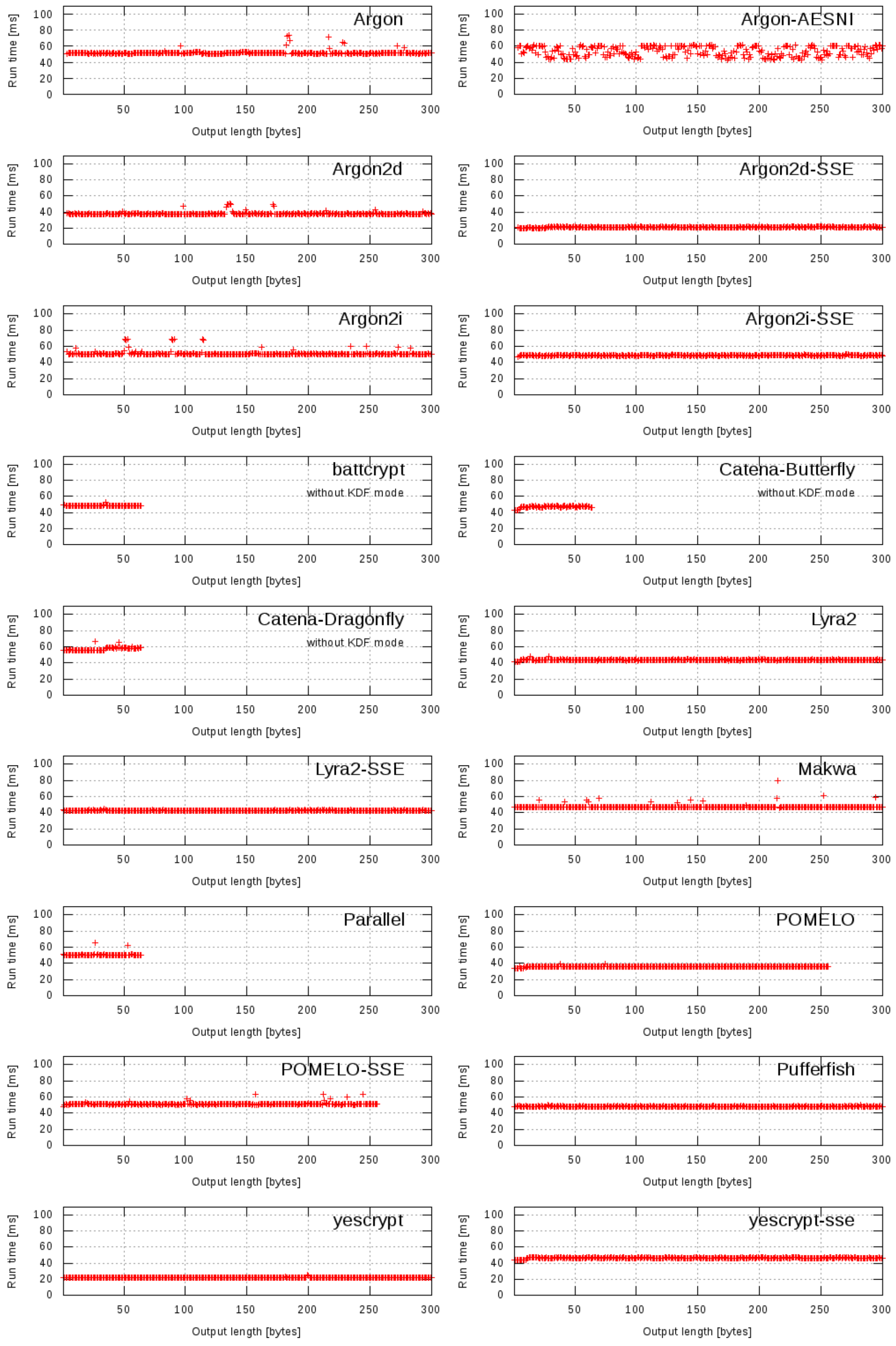

Figure 5.5: Output length test.

Measured time includes a complete run of the key-derivation, including all algorithm setup costs. It does not include the additional time needed for the operating system to boot or to initialize the required environment.

Since the attributes can be set in discrete steps only, the measured time and memory for different candidates are slightly different and cannot be directly compared to each other. The table shows only values when a particular combination is possible for a particular test case. A detailed description of the parameters used is included in our test report [9].

When a failure is listed in Table 5.4, the reason can be either that the function run-time is longer even for the minimum memory cost parameter, or because a function parameter limit was exceeded (for example, Catena seems to support only maximum cost value 255 that is not high enough for the required limits).

Candidate	Memory cost	Time cost
Argon	0	2000
Argon-AESNI	0	1500
Argon2d	100	255
Argon2d-SSE	100	255
Argon2i	100	300
Argon2i-SSE	100	500
battcrypt	0	17
Catena-Butterfly	10	60
Catena-Dragonfly	14	100
Lyra2	6	500
Lyra2-SSE	6	1000
MAKWA	0	30000
POMELO	0	11
POMELO-SSE	0	13
Parallel	0	14
Pufferfish	0	12
yescrypt	10	1
yescrypt-SSE	10	10

Table 5.5: Used parameters for input and output length tests.

5.5.3 Impact of Input Length to Run Time

This test illustrates that run time should not be dependent on input password length. The input is random password increased from 1 to 300 bytes.

Other parameters are fixed to parameters in Table 5.5 (memory cost and time cost chosen to be minimal, just to provide run time around 50ms on the particular machine). Figure 5.4 illustrates real measured data.

5.5.4 Impact of Output Key Lengths to Run Time

This test illustrates that run time should not be dependent on output key length. The input is random 32-bytes password and the output key length is increased from 1 to 300 bytes.

Other parameters are fixed to parameters in Table 5.5 (memory cost and time cost chosen to be minimal, just to provide run time around 50ms on the particular machine). Figure 5.5 illustrates real measured data.

5.5.5 Fixed Implementation Issues

Our test framework discovered several issues with the proposed algorithms. Most of them are just implementation problems, but for cryptographic algorithms, it is very important that border cases are processed properly (even if it is just a reference implementation).

As examples, out testsuite found following notable issues:

- Several candidates (*Argon* family, *Lyra2*, *POMELO*, *Pufferfish*) were not ready for big-endian platforms (producing different output on platforms with different endianness).
- *Argon2* mixes C and C++ interfaces leading to a crash if used in plain C code.
- The optimized *Argon2d* implementation output differs from the reference implementation for certain parameter values (code bug).
- There were several missing parameter boundary tests (*Argon* produced a non-random hash for input longer than 255 characters).
- *Catena* allowed use of unsupported time cost parameters (leading to a less secure function).
- *Lyra2* crashes on certain parameter values (missing a lower bound check).

Most of these were fixed either as follow-up patches or just temporarily in our testsuite. *Argon2* authors fixed all of reported problems (and Argon2 was accepted as a replacement for the Argon algorithm eventually).

5.6 Conclusions and Open Issues

Based on the criteria defined in Section 5.2, we executed several run tests of selected password hashing candidates. We took advantage of the ongoing Password Hashing Competition and based our selection on the second round finalists. Some tests mentioned in [9] are just illustrative overviews of possible parameter configuration, but were also useful to discover code portability problems and some other issues in implementations. Some of the plots were discussed on the PHC list, providing new ideas for proper comparison of algorithms and also helped to fix some mistakes in tests.

Table 5.4 shows that some functions cannot provide valid attributes for our test cases (if the minimal run-time is higher than the requested one for the intended memory use, it means that the user will be forced to wait longer than the default interval). If the minimal time exceeds 20 seconds, the algorithm is practically not usable for the given memory limits (no user will probably accept such a long delay without any possibility of reducing it).

In general, for disk encryption environment, selection and tests were able to identify candidates that clearly do not fit the intended use. These include candidates that do not have the configurable memory property (MAKWA, Parallel) or are not aligned with our requirements (POMELO, Pufferfish).

We have shown that some candidates are very slow in the presented form (and it is a question whether an optimization could bring the requisite speedup). Others would hardly be usable without changes in our scenario (Argon, battcrypt, Catena-Butterfly).

Some of these issues would be probably solvable through a modification of the algorithms (for example, by replacing the underlying cryptographic primitive), but this was out of scope for this work.

Unfortunately, there were also implementation issues. Some candidates are still not prepared to run in the big-endian environment. However, these are technical problems that can be easily fixed later. We demonstrate that implementation of cryptographic functions brings new problems not easily detected on paper alone.

The most promising candidates identified in our tests appear to be *Argon2i*, *Lyra2* and *yescrypt*.

On June 20, 2015 – Password Hashing Competition panel announced *Argon2* as a basis for the final winner and gave special recognition to *Catena*, *Lyra2*, *MAKWA* and *yescrypt* [7].

From the user perspective, it is important that a derivation function is not only secure but also usable in real situations. If the minimal run time exceeds tens of seconds, only a limited user base is likely to accept it.

6 Linux Unified Key Management Extension

6.1 Disk Encryption Key Management

LUKS (Linux Unified Key Management) [10, 14] was designed by Clemens Fruhwirth as a key management format for Linux disk encryption. At that time (2004), the Linux disk encryption configuration was complicated for a user. All configuration parameters need to be specified as command line options, and encryption keys were usually derived directly from the user passphrase. Introduction of LUKS substantially improved configuration of disk encryption and provided a better key management. Since 2009 the maintenance and further development of LUKS was handed over to the author of this thesis.

Introduction of LUKS (version 1) moved all the configuration to the binary partition header, enabling a user to configure disk encryption simpler. LUKS also introduced a concept of keyslots [50], so there can be more unlocking passphrases. A user can also change his password without reencryption of the full device.

Our motivation here was to extend this format of new features (and help easy adoption by existing LUKS users), namely we focus on authenticated encryption from the Chapter 4 and a new PBKDF discussed in Chapter 5.

Unfortunately, format changes would be complicated and incompatible. The decision was to keep the LUKS1 as the stable, backward-compatible format and develop a new format, that can also be used for the future extensions.

We named this format LUKS2 (or LUKS version 2). With the release of cryptsetup 2.0.0 it is already part of many Linux distributions. All users can use described functions without the need of installing or compiling any local source code.

LUKS was always primarily intended for COTS devices (plain disks) and must work without the need of any special hardware (in comparison to self-encrypting drives).

LUKS does not provide any steganography of plausible deniability functions (despite it can be used as a building block for them). From the public header, it is visible that the device has an encrypted part.

The metadata on disk and encrypted keyslots should be treated as a public information. It does not mean that user should not protect them, but if an attacker has a copy of the LUKS header, he still needed to run expensive PBKDF operations to find an unlocking passphrase.

6.2 LUKS1 limitations

While the introduction of LUKS1 was a success, it has some weak points and limitations. Some problems are summarized below. Most of them appeared as real use cases. Some issues were uncovered only when new storage architectures or functions like authenticated encryption were introduced.

- Some attributes stored in LUKS header are configurable (like a cipher mode), but some are fixed. The most problematic is the fixed PBKDF2 algorithm (only internal hash function is configurable).
- An authenticated encryption that uses dm-integrity subsystem needs configuration attributes to be stored in the header. (For example, information if a device has an active journal.)
- The key digest is fixed to 20 bytes of data, as a reminder of old hardcoded PBKDF2-SHA1. The recent version could use different hash configuration for internal hash function, but only first 20 bytes are stored. A possible collision causes that a wrong candidate key is validated as the correct key and the storage is unlocked with the wrong key set. This situation does not affect the confidentiality of the data but could cause data corruption if such a device with the wrong key is used.
- Anti-Forensic filter [10] is hardcoded into keyslot encryption and cannot be changed. This filter is ineffective on modern flash-based storage and should be either removed or modified in future. Also, it contains an implementation mistake (mentioned in LUKS1 specification [10] and kept in place for compatibility reasons).
- There is no header metadata redundancy, and the unfortunate placement of binary metadata in the first sector causes quite common problems with header corruption. Some operating systems that do not recognize LUKS format sometimes mark an unrecognized device with own signature that overlaps with LUKS header.
- While keyslots are independent, some parameters are shared among all of them. We cannot have keyslots with different encryption parameters, key sizes or different PBKDF.
- The header does not allow to store persistent activation flags or reencryption intermediate state (when two sets of attributes are temporarily needed).

6.3 Cryptsetup and LUKS2 Design

We developed a new LUKS2 format that can replace LUKS1. A user is not forced to use it unless he wants to use new extensions.

Detailed design goals from the developer's perspective are described in Appendix A. This description focuses on the important factors for the implementation of extensions described in previous chapters of this thesis.

To build an algorithm-agnostic system, we must be prepared for a change of any attributes in future. For this reason, the metadata stored in LUKS2 is in a generic format. We used JSON [117] as a configurable and straightforward format that is also human readable. A simple example is in Figure 6.1 as a token definition (described in Appendix A, section A.3) that is in human readable form almost self-explanatory.

```
{
  "type":"luks2-keyring",
  "keyslots":[ "1" ],
  "key_description":"MyKeyringKeyID"
}
```

Figure 6.1: Token configuration example.

There are of course other similar formats, but JSON is simple enough and an implementation of parser requires only very limited additional code. Some other formats (notably XML) are not easily human-readable and also allows complicated constructions that we do not need. Based on experiments, we changed structure several times and the change in JSON was always very straightforward and simple (in comparison to change in binary structures).

The LUKS1 metadata can be easily converted to LUKS2 format. With LUKS2, metadata is also replicated so that we can recover from simple data corruptions. The only exception is binary encrypted keyslots, where duplication of metadata would directly remove anti-forensic function purpose.

The format also allows storing objects that internal implementation does not directly support. Such metadata can be used for external tokens that communicate with hardware. Also, it can store additional information (an example is to a store for additional random-seed to properly initialize RNG before the system is even decrypted).

6.3.1 Extension for Argon2

Adding additional cost attributes for a new PBKDF (memory and number of threads) is with the new format trivial and looks like an example in Figure 6.2.

```
 {
  "type":"argon2i",
  "time":4,
  "memory":222434,
  "cpus":2,
  "salt":"l8NTVxNKf3+Sjd3XD0WcYVSC+CpHSS+o5zbEH+Q/KWc="
}
```

Figure 6.2: PBKDF configuration example.

We support *Argon2i* and *Argon2id* algorithms flavours (*Argon2d* is not suggested for PBKDF use case) [105]. Using Argon2 in real software is unfortunately not so straightforward as expected.

The first issue is that Argon2 is not implemented in any common cryptographic library yet (we hope that it appears at least in OpenSSL soon). So we have to use either bundled code or a separate library (if available) as provided by authors. Using bundled code is not a good security practice and need to be removed in future.

Another issue is setting costs of PBKDF. As described in section 5.5.1, memory and time cost influences each other. Argon2 authors, in a draft of RFC document [118], provide some guidelines for cost parameters setup. For hard-drive encryption (and 2 GHz CPU and 3 seconds run time) they suggest using 6GiB of memory. As our benchmark shows, such a high cost is almost impossible to achieve even with high-end systems (and provided implementation). More realistic value seems to be 1 GiB (with 2 seconds run time).

We have implemented benchmark (similar to PBKDF2 benchmark that is already present in cryptsetup) that tries to find parameters according to a user needs. The benchmark has to prioritize runtime to memory requirements because extending derivation time over a requested time would make the system very user unfriendly. The benchmark increases required memory until it finds an acceptable compromise. If the system is powerful enough, it also increases the time (iteration) cost until the requested time cost is met. This way we can provide a reliable LUKS2 implementation with Argon2 PBKDF (with the maximal available cost parameters) even for slower systems, like popular Raspberry Pi systems.

Another problem is specific to the Linux platform. Memory management in most Linux systems use over-committing strategy [119] for memory allocations (it allows to allocate more memory than system actually can physically provide). This strategy works quite well for common use cases but not for the memory-hard algorithms that *really* use the allocated memory. Moreover, even additional virtual memory (swapping) is highly ineffective here.

If a Linux kernel hits the out-of-memory situation and all previous attempt to flush unused memory failed, it uses so-called out-of-memory killer, that select a process that seems to cause memory shortage and terminates it. The process that uses a lot of memory for key derivation is, of course, the first one to terminate. The problem is that process termination is unconditional, and the application itself could potentially keep some sensitive data in memory. This is exactly the situation when a user is unlocking a LUKS2 disk formatted on a system with more memory requirements, and the terminated process has an unlocking password in memory. A usable and portable solution for this problem is part of the future work.

6.3.2 AEGIS and MORUS AEAD ciphers

Section 4.2.2 describes required attributes for an authenticated encryption mode suitable for disk encryption. Because cryptography implementations in the Linux kernel did not provide any suitable configurations, we have decided to improve the situation by implementing some new CAESAR competition [84] candidates. The implementation is part of diploma thesis of Ondrej Mosnáček [98] and was supervised by the author of this thesis.

We have selected few candidates from the last round of CAESAR cryptographic competition [84] that are suitable for our disk encryption use case and implemented them for the Linux kernel cryptography interface. Two selected candidates, AEGIS [85] and MORUS [86], are currently CAESAR finalists. The provided implementation was subsequently submitted and accepted into the Linux mainline kernel [95] for version 4.18.

The implementation contains generic modules (usable on all hardware platforms that Linux kernel supports) and an optimized version for Intel processors that supports AES-NI instructions (for use in AEGIS) and SSE2 and AVX2 instructions (for MORUS). Implementation of these ciphers is available through standard kernel cryptographic interface, thus available for all users of the kernel (not only for disk encryption). Basic performance tests for our use case are presented as the part of measurements in the Section 4.5.

6.3.3 LUKS2 Extension for AEAD

Configuration for AEAD in LUKS2 is just an extension of a structure used for a non-authenticated mode. Examples of metadata are in Figure 6.3 for a native AEAD mode, and in Figure 6.4 for composed mode (AES-XTS with HMAC(SHA256) integrity tags). Note that we do not use additional journal encryption there (journal metadata is visible to an attacker, but data integrity is still guarded by the authentication tag).

```
{
  "type":"crypt",
  "offset":"4194304",
  "iv_tweak":"0",
  "size":"dynamic",
  "encryption":"aegis256-random",
  "sector_size":512,
   "integrity":{
    "type":"aead",
    "journal_encryption":"none",
    "journal_integrity":"none"
   }
}
```

Figure 6.3: AEAD configuration example.

```
{
  "type":"crypt",
   "offset":"4194304",
   "iv_tweak":"0",
   "size":"dynamic",
   "encryption":"aes-xts-plain64",
   "sector_size":512,
   "integrity":{
     "type":"hmac(sha256)",
     "journal_encryption":"none",
     "journal_integrity":"none"
    }
}
```

Figure 6.4: Composed AEAD configuration example.

6.4 Non-cryptographic Data Integrity

While the goal of our work was to provide authenticated encryption, the dm-integrity [120] module can be used separately to provide non-cryptographic parity checking on COTS devices.

For this use case, we provided a separate application called *integritysetup* as part of the cryptsetup project [11]. This side-effect functionality will be probably also integrated into volume management systems (a user can add a volume that checks data parity). Nice demonstration what such a volume can uncover is an experiment that shows how software RAID on Linux can distribute silent error to multiple disks [96].

7 Conclusions and Future Work

The goal of this thesis is to analyze selected current problems with Full Disk Encryption systems and propose new improvements. Our primary focus is on COTS devices that do not provide any hardware support for authentication or integrity protection.

As part of our effort, and based on our independent implementation, we analyzed TrueCrypt [60] disk encryption format. Development of the TrueCrypt was abruptly terminated; so we extended our analysis and implementation of new functions in VeraCrypt [61] that seems to be the most promising successor to TrueCrypt.

7.1 FDE Authenticated Encryption

The length-preserving nature of FDE is perceived as a significant hurdle to provide cryptographic integrity protection. Despite that, many systems use FDE as an underlying layer to provide encryption. The situation in open-source Linux filesystems is not any better, none of the widely deployed filesystems in the mainline Linux kernel implement authenticated encryption.

We provide a proposal and a practical implementation (included as a part of the mainline Linux kernel since version 4.12 and cryptsetup since version 2.0) that combine both confidentiality and cryptographic data integrity protection. We believe that the need for proper cryptographic data integrity protection, in general, should be one of the primary goals for secure storage development in the future.

The idea to use authenticated encryption uncovered several issues that we had to solve. The first one is the lack of suitable AEAD algorithms. Our solution is prepared for the use of new algorithms in the future. Nevertheless, as an independent part of our research, there is an implementation of two new AEAD ciphers based on CAESAR cryptographic competition [98]. This implementation was accepted in the mailnline Linux kernel version 4.18.

The problem with the length-preserving nature of disk encryption led to the idea to implement software implementation of per-sector metadata store that securely provides needed metadata space for authentication tags. We demonstrated that the major concerns related to the decreased performance of such a design are, with a new flash memory-based storage, significantly reduced. Our practical tests show that for some use patterns, there is no visible performance decrease. Our per-sector metadata emulating store can be even used for parity (non-cryptographic) integrity protection.

Future research will investigate the possibility to implement such a per-sector store directly over persistent byte-addressable storage. This solution would provide native performance while the authenticated encryption remains the same.

7.2 Passphrase Protection for FDE

The next significant part of our research investigates issues relates to key management for disk encryption. We propose a replacement of the commonly used password-based key derivation function PBKDF2 with a memory-hard function. This effort increases the cost of a dictionary-based attack to key storage.

We analyzed proposals for the Password Hashing Competition [7] not only for our use cases, but also from other practical aspects. We used the Argon2 [105] algorithm (the winner of the competition) as a replacement for the key derivation function for the LUKS disk encryption.

The practical output of all our research activities covered in this thesis is a proposal of the extended LUKS2 on-disk format for disk encryption management. This format allows to configure authenticated disk encryption easily, but also enables other extensions in the future. It also uses the Argon2 function as the default password-based key derivation function. The implementation uncovers some interesting practical issues with the use of a memory-hard function on real systems. For example, the need to tune the Linux memory over-committing strategy or an effective setting of cost parameters in limited embedded devices. Analysis of dictionary attack costs with various cost parameters and comparison with previous analyses of PBKDF2 is another future research goal.

A LUKS2 On-Disk Format Specification

Document History

Version	Date	Author
1.0.0	2018-08-02	Milan Broz `<gmazyland@gmail.com>`
Snapshot of `https://gitlab.com/cryptsetup/LUKS2-docs`		

A.1 Introduction

LUKS2 is the second version of the *Linux Unified Key Setup* for disk encryption management. It is the follow-up of the LUKS1 [10, 14] format that extends capabilities of the on-disk format and removes some known problems and limitations. Most of the basic concepts of LUKS1 remain in place as designed in *New Methods in Hard Disk Encryption* [14] by Clemens Fruhwirth.

LUKS provides a generic key store on the dedicated area on a disk, with the ability to use multiple passphrases[1] to unlock a stored key. LUKS2 extends this concept for more flexible ways of storing metadata, redundant information to provide recovery in the case of corruption in a metadata area, and an interface to store externally managed metadata for integration with other tools.

While the implementation of LUKS2 is intended to be used with Linux-based dm-crypt [62] disk encryption, it is a generic format.

A.1.1 Design Goals

The LUKS header provides metadata for a disk encryption setup. LUKS1 version [10] contains a binary header for storing necessary metadata (like encryption algorithms parameters) and eight keyslots for independent passphrases to unlock one volume key.
The LUKS2 format is designed to provide these features:

- Cover all possibilities of LUKS1.
- Support configurable memory-hard key-derivation algorithms.
- The new header can store additional metadata for external tools and expose an interface for regular updates.
- LUKS2 uses only a small binary header that can be easily used by automatic detection tools like *blkid* in Linux. The binary header is par-

1. LUKS can use a passphrase or a key file, both are processed identically.

tially compatible with LUKS1, so legacy tools still recognize a partition as the LUKS type and can see device UUID. All other metadata are stored in the non-binary header.

- The header includes a checksum mechanism that detects data corruption and unintentional header mangling.
- LUKS header can be detached from a LUKS device and can be stored on a separate device or in a file. With the detached header, the encrypted device contains no visible and detectable metadata.
- Metadata area is stored in two copies to allow for a possible recovery.[2] The recovery is transparent for most of the operations (device should recover automatically if at least one header is correct).
- Keyslot binary area is not duplicated (for security reasons), but the area is now allocated in higher device offset where a random data corruption should happen more rarely.
- A header can be upgraded in-place for most of existing LUKS1 devices.[3]
- Header store persistent flags that are used during activation.[4]
- The number of keyslots is limited only by the provided header area size.[5]
- Keyslots have priorities. Some keyslots can be marked for use only if explicitly specified (for example as a recovery keyslot).
- Metadata are stored in the JSON format that allows for future extensions without modifying binary structures. Such an extension is for example support for authenticated encryption or support for online data reencryption.
- All metadata are algorithm-agnostic and can be upgraded to new algorithms later without header structure changes.
- Volume key digest is no longer limited by 20 bytes (based on legacy SHA-1) as in the LUKS1 header.

2. A common issue with LUKS1 is metadata corruption caused by a partitioning tool that does not recognize LUKS format.
3. A configuration that does not use default on-disk alignment of user data offset could lack needed space for new metadata area.
4. Example of a persistent flag is support for the TRIM operation. These flags should replace the need for flags in the external `/etc/crypttab` file.
5. Reference implementation limits the number of keyslots to 32.

- Keyslot can contain an unbound key (key not assigned to any encrypted data segment) that can be used for external applications. Size of an unbound key can be different from volume key size in other keyslots.
- The header contains a concept of *tokens* that are objects, assigned to keyslots, which contain metadata describing *where to get unlocking passphrase*. Tokens can be used for support of external key store mechanisms.

A.1.2 Reference Implementation

The LUKS2 format is currently implemented and fully supported in libcryptsetup [11] on Linux systems, together with the LUKS1 format. The implementation automatically detects version according to the binary header.

New LUKS2 devices can be created with `-type luks2` option. For example, `cryptsetup format -type luks2 <device>`.

A.2 LUKS2 On-Disk Format

The LUKS2 header is located at the beginning (sector 0) of the block device (for a detached header on a dedicated block device or in a file). The basic on-disk structure is illustrated in Figure A.1.

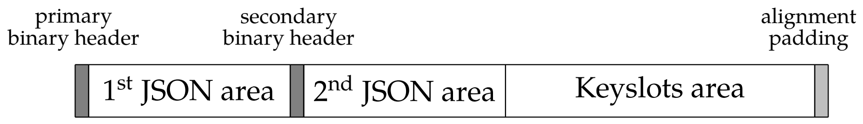

Figure A.1: LUKS2 header on-disk structure.

The LUKS2 header contains three logical areas:

- binary structured header (one 4096-byte sector, only 512-bytes are used),
- area for metadata stored in the JSON format and
- keyslot area (keyslots binary data).

The binary and JSON areas are stored twice on the device (primary and secondary header) and under normal circumstances contain same functional metadata. The binary header size ensures that the binary header is always written to only one sector (atomic write). Binary data in the keyslots area is allocated on-demand. There is no redundancy in the binary keyslots area.

A.2.1 Binary Header

The binary header is intended for a quick scanning by *blkid* and contains a signature to detect LUKS device, basic information (labels), header size and metadata checksum. Binary header in the C structure is described in Figure A.2.

All integer values are stored in the big-endian format. All strings are in the C format, and a valid header must have all strings terminated by the zero byte.

The primary binary header must be stored in sector 0 of the device. The secondary header starts immediately after the primary header JSON area (see *hdr_size* in primary header). To allow for an easy recovery, the secondary header must start at a fixed offset listed in Table A.1.

Offset (hexa) [bytes]	**JSON area** [kB]
16384 (0x004000)	12
32768 (0x008000)	28
65536 (0x010000)	60
131072 (0x020000)	124
262144 (0x040000)	252
524288 (0x080000)	508
1048576 (0x100000)	1020
2097152 (0x200000)	2044
4194304 (0x400000)	4092

Table A.1: Possible LUKS2 secondary header offsets and JSON area size.

The LUKS1 compatible fields (*magic*, *UUID*) are placed intentionally on the same offsets. The binary header contains these fields:

- **magic** contains the unique string (see C defines `MAGIC_1ST` for the primary header and `MAGIC_2ND` for the secondary header in Figure A.2).
- **version** must be set to 2 for LUKS2.
- **hdr_size** contains the size of the header with the JSON data area. The offset and size of the secondary header must match this size. It is a prevention to rewrite of a header with a different JSON area size.
- **seqid** is a counter (sequential number) that is always increased when a new update of the header is written. The header with a higher *seqid* is more recent and is used for recovery (if there are primary and sec-

```
#define MAGIC_1ST "LUKS\xba\xbe"
#define MAGIC_2ND "SKUL\xba\xbe"
#define MAGIC_L    6
#define UUID_L    40
#define LABEL_L   48
#define SALT_L    64
#define CSUM_ALG_L 32
#define CSUM_L    64

// All integers are stored as big-endian.
// Header structure must be exactly 4096 bytes.

struct luks2_hdr_disk {
  char     magic[MAGIC_L];        // MAGIC_1ST or MAGIC_2ND
  uint16_t version;               // Version 2
  uint64_t hdr_size;              // size including JSON area [bytes]
  uint64_t seqid;                 // sequence ID, increased on update
  char     label[LABEL_L];        // ASCII label or empty
  char     csum_alg[CSUM_ALG_L]; // checksum algorithm, "sha256"
  uint8_t  salt[SALT_L];          // salt, unique for every header
  char     uuid[UUID_L];          // UUID of device
  char     subsystem[LABEL_L];    // owner subsystem label or empty
  uint64_t hdr_offset;            // offset from device start [bytes]
  char    _padding[184];          // must be zeroed
  uint8_t  csum[CSUM_L];          // header checksum
  char    _padding4096[7*512];    // Padding, must be zeroed
} __attribute__ ((packed));
```

Figure A.2: LUKS2 binary header on-disk structure.

ondary headers with different seqid, the more recent one is automatically used).

- **label** is an optional label (similar to a filesystem label).
- **csum_alg** is a checksum algorithm. Metadata checksum covers both the binary data and the following JSON area and is calculated with the checksum field zeroed. By default, plain SHA-256 function is used as the checksum algorithm.
- **salt** is generated by an RNG and is different for every header (it differs on the primary and secondary header), even the backup header must contain a different salt. The salt is not used after the binary header is read, the main intention is to avoid deduplication of the header sector. The salt must be regenerated on every header repair (but not on a regular update).
- **uuid** is device UUID with the same format as in LUKS1.

- **subsystem** is an optional secondary label.
- **hdr_offset** must match the physical header offset on the device (in bytes). If it does not match, the header is misplaced and must not be used. It is a prevention to partition resize or manipulation with the device start offset.
- **csum** contains a checksum calculated with the *csum_alg* algorithm. If the checksum algorithm tag is shorter than the *csum* field length, the rest of this field must be zeroed.

The rest of the binary header (including padding fields) must be zeroed. The *version*, *UUID*, *label* and *subsystem* fields are intended to be used in the *udev* database and *udev* triggered actions. For example, a system can manage all LUKS2 devices with a specific subsystem field automatically by some external tool. The *label* and *UUID* can be used the same way as a filesystem label.

A.2.2 JSON Area

The JSON area starts immediately after the binary header (end of JSON area must be aligned to 4096-byte sector offset). Size of JSON area is determined from binary header *hdr_size* field: $JSON\ area\ size = hdr\_size - 4096$.

The area contains metadata in JSON format [117]. The JSON metadata are stored in the area as a C string that must be terminated by the zero character. The unused remainder of the area must be empty and filled with zeroes. The header cannot store larger metadata than this fixed JSON area.[6]

A.2.3 Keyslots Area

Keyslots area is a reserved space on the disk that can be allocated for a binary data from keyslots. There are stored encrypted keys referenced from keyslots metadata. The structure of stored keyslot binary data depends on the keyslot type. The *luks2* keyslots type uses the same LUKS1 binary structure.

The allocated area is defined in a keyslot by an *area* object that contains *offset* (from the device beginning) and *size* fields. Both fields must be validated to point to the keyslot area. Invalid values must be rejected.

6. For now, reference implementation uses only areas with 16 kB header (4kB binary header + 12kB JSON area).

A.3 LUKS2 JSON Metadata Format

The LUKS2 metadata allows defining objects that, according to the *type* field, defines a specific functionality. Objects that are not recognized byt the implementation are ignored, but metadata are still maintained inside the JSON metadata. Implementation must validate the JSON structure before updating the on-disk header.

The LUKS2 structure has 5 mandatory top-level objects (see Figure A.3) as follows:

- **config** contains persistent header configuration attributes.
- **keyslots** are objects describing encrypted keys storage areas.
- **digests** are used to verify that keys decrypted from keyslots are correct.
- **segments** describe areas on disk that contain user encrypted data.
- **tokens** can optionally include additional metadata, bindings to other systems – *how to get a passphrase for the keyslot*.

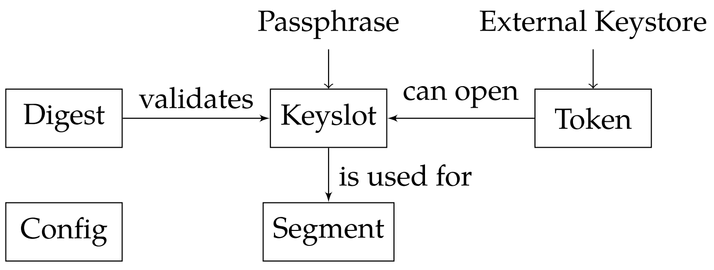

Figure A.3: LUKS2 objects schema.

Except top-level objects listed above, all JSON objects must have their names formatted as a string that represents a number in the decimal notation (unsigned integer) – for example *"0"*, *"1"* and must contain attribute *type*. According to the *type*, the implementation decides how to handle (or ignore) such an object. This notation allows mapping to LUKS1 API functions that use an integer as a reference to keyslots objects.

Binary data inside JSON (for example *salt*) are stored in the Base64 [121] encoding. JSON cannot store 64-bit integers directly, a value for an object that represents unsigned 64-bit integer (*offset* or *size*) is stored as a string in the decimal notation and later converted to the 64-bit unsigned integer. Such an integer is referenced as *string-uint64* later.

A.3.1 LUKS2 JSON Example

The following example contains full JSON metadata from the reference implementation for a LUKS2 device that is encrypted with the AES-XTS cipher, contains two keyslots and one token. The token type is *keyring* (can be unlocked by a passphrase in the keyring) and it is bound to the second keyslot.

```
{
  "keyslots":{
    "0":{
      "type":"luks2",
      "key_size":32,
      "af":{
        "type":"luks1",
        "stripes":4000,
        "hash":"sha256"
      },
      "area":{
        "type":"raw",
        "encryption":"aes-xts-plain64",
        "key_size":32,
        "offset":"32768",
        "size":"131072"
      },
      "kdf":{
        "type":"argon2i",
        "time":4,
        "memory":235980,
        "cpus":2,
        "salt":"z6vz4xK7cjan92rDA5JF806Jk2HouV008DMB6GlztVk="
      }
    },
    "1":{
      "type":"luks2",
      "key_size":32,
      "af":{
        "type":"luks1",
        "stripes":4000,
        "hash":"sha256"
      },
      "area":{
        "type":"raw",
        "encryption":"aes-xts-plain64",
        "key_size":32,
        "offset":"163840",
        "size":"131072"
      },
      "kdf":{
        "type":"pbkdf2",
        "hash":"sha256",
        "iterations":1774240,
        "salt":"vWcwY3rx2fKpXW2Q6oSCNf8j5bvdJyEzB6BNXECGDsI="
      }
    }
  },
  "tokens":{
    "0":{
      "type":"luks2-keyring",
      "keyslots":[
        "1"
      ],
      "key_description":"MyKeyringKeyID"
    }
  },
  "segments":{
    "0":{
      "type":"crypt",
      "offset":"4194304",
      "iv_tweak":"0",
      "size":"dynamic",
      "encryption":"aes-xts-plain64",
      "sector_size":512
    }
  },
  "digests":{
    "0":{
      "type":"pbkdf2",
      "keyslots":[
        "0",
        "1"
      ],
      "segments":[
        "0"
      ],
      "hash":"sha256",
      "iterations":110890,
      "salt":"G8gqtKhS96IbogHyJL0+t9kmjLkx+DM3HHJqQtgc2Dk=",
      "digest":"C9JWko5m+oYmjg6R0t/98cGGzLr/4UaG3hImSJMivfc="
    }
  },
  "config":{
    "json_size":"12288",
    "keyslots_size":"4161536",
    "flags":[
      "allow-discards"
    ]
  }
}
```

A.3.2 Keyslots Object

Keyslots object contains information about stored keys – areas, where binary keyslot data are located, encryption and anti-forensic function used, password-based key derivation function (PBKDF) and related parameters.

Every keyslot object must contain:

- **type** [string] the keyslot type (only the *luks2* is currently used).
- **key_size** [integer] the key size (in bytes) stored in keyslot.
- **area** [object] the allocated area in the binary keyslots area.
- **kdf** [object] the PBKDF type and parameters used.
- **af** [object] the anti-forensic splitter [10] (only the *luks1* type is currently used).
- **priority** [integer,optional] the keyslot priority. Here 0 means *ignore* (the slot should be used only if explicitly stated), 1 means *normal* priority and 2 means *high* priority (tried before *normal* priority).

The *luks2* keyslot type uses the same logic as LUKS1 keyslot, but allows for per-keyslot algorithms (for example different PBKDF).

The *area* object contains these fields:

- **type** [string] the area type (only the *raw* type is currently used).
- **offset** [string-uint64] the offset from the device start to the beginning of the binary area (in bytes).
- **size** [string-uint64] the area size (in bytes).
- **encryption** [string] the area encryption algorithm, in dm-crypt notation (for example `aes-xts-plain64`).
- **key_size** [integer] the area encryption key size.

The *af* (anti-forensic splitter) object contains these fields:

- **type** [string] the anti-forensic function type (only the *luks1* type compatible with LUKS1 [10] is currently used).
- **stripes** [integer] the number of stripes, for historical reasons only the 4000 value is supported.
- **hash** [string] the hash algorithm used (SHA-256).

The LUKS1 AF splitter is no longer much effective on modern storage devices. The functionality is here mainly for compatibility reasons. In future, it will be probably replaced.

The *kdf* object fields are:

- **type** [string] the PBKDF type (*pbkdf2*, *argon2i* or *argon2id* type).
- **salt** [base64] the salt for PBKDF (binary data).

For the *pbkdf2*[7] type (compatible with LUKS1) additional fields are:

- **hash** [string] the hash algorithm for the PBKDF2 (SHA-256).
- **iterations** [integer] the PBKDF2 iterations count.

For *argon2i* and *argon2id*[8] type fields are:

- **time** [integer] the time cost (in fact the iterations count for Argon2).
- **memory** [integer] the memory cost, in kilobytes. If not available, the keyslot cannot be unlocked.
- **cpus** [integer] the required number of threads (CPU cores number cost). If not available, unlocking will be slower.

A.3.3 Segments Object

Segments object contains a definition of encrypted areas on the disk containing user data (in LUKS1 mentioned as the user data payload). For a normal LUKS device, there is only one data segment present.[9]

The *segment* object contains these fields:

- **type** [string] the segment tyoe (only the *crypt* type is currently used).
- **offset** [string-uint64] the offset from the device start to the beginning of the segment (in bytes).
- **size** [string or string-uint64] the segment size (in bytes) or *dynamic* if the size of the underlying device should be used (dynamic resize).
- **iv_tweak** [string-uint64] the starting offset for the Initialization Vector (IV tweak).
- **encryption** [string] the segment encryption algorithm, in the dm-crypt notation (for example `aes-xts-plain64`).
- **sector_size** [integer] the sector size for segment (512 or 4096 bytes).
- **integrity** [object,optional] the LUKS2 user data integrity protection type.

User data integrity protection is an experimental feature [4]) and requires *dm-integrity* and *dm-crypt* drivers with integrity support.

The *integrity* object contains these fields:

- **type** [string] the integrity type (in the dm-crypt notation, for example *aead* or *hmac*(*sha256*)).

7. PBKDF2 contains the time cost (iterations) that describes how many times PBKDF2 must iterate to derive the candidate key.
8. Argon2 algorithms, here used as PBKDF, are memory-hard [105] and have three costs: time, memory required and number of threads (CPUs).
9. During the data reencryption, the data area is internally divided according to the new and the old key, but only one abstracted area should be presented to the user.

- **journal_encryption** [string] the encryption type for the *dm-integrity* journal (not implemented yet, use *none*).
- **journal_integrity** [string] the integrity protection type for the *dm-integrity* journal (not implemented yet, use *none*).

A.3.4 Digests Object

The digests object is used to verify that a key decrypted from a keyslot is correct. Digests are assigned to keyslots and segments. If it is not assigned to a segment, then it is a digest for an unbound key. Every keyslot must have one assigned digest object. The key digest object also specifies the exact key size for the encryption algorithm of the segment.

The *digest* object contains these fields:

- **type** [string] the digest tyoe (only the *pbkdf2* type compatible with LUKS1 is used).
- **keyslots** [array] the array of keyslot objects names that are assigned to the digest.
- **segments** [array] the array of segment objects names that are assigned to the digest.
- **salt** [base64] the binary salt for the digest.
- **digest** [base64] the binary digest data.

The *pbkdf2* digest (similar to a kdf object in keyslot) contains these fields:

- **hash** [string] the hash algorithm for PBKDF2 (SHA-256).
- **iterations** [integer] the PBKDF2 iterations count.

A.3.5 Config Object

The *config* object contains attributes that are global for the LUKS device.

It contains these fields:

- **json_size** [string-uint64] the JSON area size (in bytes). Must match the binary header.
- **keyslots_size** [string-uint64] the binary keyslot area size (in bytes). Must match the binary header.
- **flags** [array, optional] the array of string objects with persistent flags for the device.
- **requirements** [array, optional] the array of string objects with additional required features for the LUKS device.

The *flags* can contain feature and activation flags. Unknown flags are ignored.

The reference implementation uses these flags:

- **allow-discards** allows TRIM (discards) on the active device.
- **same-cpu-crypt** compatibility performance flag for dm-crypt [62].
- **submit-from-crypt-cpus** compatibility flag for dm-crypt [62].
- **no-journal** disable data journalling for dm-integrity [120].

The *requirements* array can contain an array of additional features that are mandatory when manipulating with a LUKS device and metadata or that are required for proper device activation. If an implementation detects a string that it does not recognize, it must treat the whole metadata as read-only and must avoid device activation. These requirements markers are used for future extensions to mark the header to be not backward compatible. Currently, only the *offline-reencrypt* flag is used that marks a device during offline reencryption to prevent an activation until the reencryption is finished.

A.3.6 Tokens Object

A token is an object that can describe *how to get a passphrase* to unlock a particular keyslot. It can also contain additional user-defined JSON metadata.

The mandatory fields for every token are:

- **type** [string] the token type (tokens with *luks2*- prefix are reserved for the implementation internal use).
- **keyslots** [array] the array of keyslot objects names that are assigned to the token.

The rest of the JSON content is the particular token implementation and can contain arbitrary JSON structured data (implementation should provide an interface to the JSON metadata directly).

For example, the reference implementation of *luks2-keyring* token allows automatic activation of the device if the passphrase is preloaded into a keyring with the specified ID.

The *luks2-keyring* token type contains these fields:

- **type** [string] is set to the *luks2-keyring*.
- **keyslots** [array] is assigned to the specific keyslot(s).
- **key_description** [string] contains the ID of the keyring entry with a passphrase.

A.4 LUKS2 Operations

Basic operations of a LUKS2 device are the same as specified in LUKS1 [10]. Header update operations must be synchronized due to the redundancy of metadata and operations must be serialized to prevent concurrent processes from updating the metadata at the same time. The metadata update must be implemented in such a way that at least of one header (primary or secndary) is always valid to allow for a proper recovery in the case of a failure. These steps require an implementation of some high-level locking of metadata access.

A.4.1 Device Formatting

Initialization (formatting) of a LUKS2 device starts with generating basic metadata parameters, like *UUID* and writing both binary headers and basic metadata structures. The JSON area must always contain valid LUKS2 top-level objects. The config object must be initialized to include proper area size parameters that match the binary header. If the LUKS2 header references a user data segment, that segment must be initialized with all mandatory parameters. The keyslots, digests and tokens can be empty in this step.

A.4.2 Keyslot Initialization

The next step is allocation of new keyslot and metadata and assignment to the key digests and segments. The key stored in the keyslot and salt for the keyslot should be generated using a cryptographically secure RNG.

The PBKDF cost parameters (iterations, memory, CPU cores) that are used to derive the keyslot unlocking key from a user passphrase must be either specified by the user, or it can be benchmarked according to user needs. See the reference cryptsetup implementation [11] as an example of this approach.

Once the key is generated, a new key digest is created, and a new keyslot object is allocated and assigned to the digest (and segment). The new keyslot contains the binary keyslot area allocated according to the key size.

The size of the binary allocated area is determined according to the key size and the anti-forensic (AF) splitter output (see section 2.4 in LUKS1 [10]).

LUKS2 keyslots can store different keys with different key sizes. The allocation of binary keyslot data depends on the order of creation. Keyslot positions are no longer fixed as in LUKS1.

The last step of keyslot initialization writes the encrypted key to the allocated binary keyslot area. A user passphrase and a salt are processed by the configured PBKDF. The PBKDF output key is used for the keyslot binary area encryption algorithm. The key is split using AF splitter and encrypted by the keyslot encryption algorithm.

A.4.3 Keyslot Content Retrieval

The user provided passphrase with the salt and parameters from the header metadata are processed through the PBKDF. The derived key is used to decrypt the binary keyslot area. The decrypted content is processed (merged) in the AF splitter. The assigned key digest is calculated with the recovered candidate key. If the calculated digest and the digest in metadata match, the recovered key is valid. If the digest does not match, the provided passphrase must be rejected.

A.4.4 Keyslot Revocation

To discard a keyslot, the binary area for the keyslot must be physically overwritten (to discard the stored data). After this step, the keyslot metadata object must be removed with all bindings to digests and segments.

Note that the key digest and its binding to the segment can remain in metadata (not assigned to any keyslots). If a user has the copy of the encryption key, the validity of the key can still be verified with this digest and the device can be later still activated.

A.4.5 Metadata Recovery

The replicated metadata allows for a full LUKS2 header recovery (except binary keyslot areas) if some part of headers become corrupted. A part of the recovery can be automated, but because this process can revert some intentional changes, a user interaction is suggested.

The automatic recovery should always update both copies to the more recent version (with higher *seqid*). The metadata handler should first try to load the primary header, then the secondary header. If one of the headers is more recent, the older header is updated. If the primary header is corrupted, a scan on several known offsets for the secondary header can be performed.

A.4.6 Mandatory Requirements

While the LUKS2 format is algorithm-agnostic, some algorithm implementations are crucial for internal function.

The cryptographic backend for LUKS2 must support these algorithms:

- **SHA-1** hash algorithm (for compatibility with old LUKS1 devices).
- **SHA-256** hash algorithm, used as the default checksum for the binary header and in the PBKDF2 digest.

- **PBKDF2** password-based key derivation (for digest and backward compatibility with LUKS1).
- **Argon2i and Argon2id** memory-hard key derivation functions for new LUKS2 keyslots.
- **AES-XTS** symmetric cipher for the default keyslot encryption and the default user data encryption.

A.4.7 Conversion from LUKS1

If an existing LUKS1 device header contains enough space for the LUKS2 metadata, then it can be converted in-place to the LUKS2 format. Reference implementatio provides the `cryptsetup convert -type luks2` command.

LUKS1 [name]	**128-bit key** [sectors]	**256-bit key** [sectors]	**512-bit key** [sectors]
Header	0	0	0
Keyslot 0	8	8	8
Keyslot 1	136	264	512
Keyslot 2	264	520	1016
Keyslot 3	392	776	1520
Keyslot 4	520	1032	2024
Keyslot 5	648	1288	2528
Keyslot 6	776	1544	3032
Keyslot 7	904	1800	3536
Padding	1032	2056	4040
Data offset	2048	4096	4096
Unused sectors	1016	2040	56

Table A.2: Offsets (in 512-byte sectors) of common LUKS1 headers.

For reference, Table A.2 contains offsets of LUKS1 keyslots that can be converted to LUKS2 in-place. The resulting LUKS2 header has 12kB JSON area in all these cases. Note that the binary keyslot area is directly copied to the proper position, there is no recovery possible if the convert operation fails. Schema of area locations during conversion is illustrated in Figure A.4.

If a LUKS2 header uses compatible options with LUKS1 (PBKDF2, no integrity protection, no tokens, no unbound keys) then it can also be converted back to the LUKS1 header in-place with the `cryptsetup convert -type luks1` command.

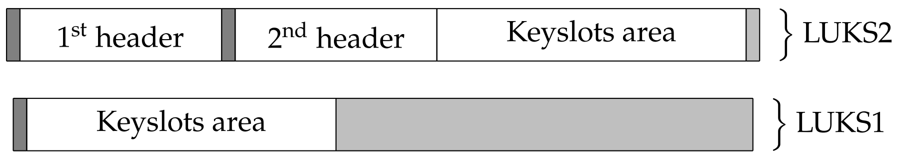

Figure A.4: LUKS1/LUKS2 areas placement.

A.4.8 Algorithm Definition Examples

The LUKS2 specification supports all algorithms that are provided by the cryptographic backend (in the Linux case by kernel dm-crypt and userspace cryptographic library). Figures A.3 and A.4 list few examples of symmetric ciphers for data encryption and PBKDF algorithms.

Algorithm in LUKS2 notation	**Description**
pbkdf2	PBKDF2 with HMAC-SHA256 [55]
argon2i	Argon2i as PBKDF (data independent) [105]
argon2id	Argon2id as PBKDF (combined mode) [105]

Table A.3: LUKS2 PBKDF algorithms.

Algorithm in dm-crypt [62] notation	**Description**
aes-xts-plain64	AES in XTS mode with sequential IV [36, 39]
aes-cbc:essiv:sha256	AES in CBC mode with ESSIV IV [62, 36]
serpent-xts-plain64	Serpent cipher with sequential IV [65]
twofish-xts-plain64	Twofish cipher with sequential IV [66]
aegis128-random	AEGIS (128-bit) with random IV, AEAD [85]
aegis256-random	AEGIS (256-bit) with random IV, AEAD [85]
morus640-random	MORUS640 with random IV, AEAD [86]
morus1280-random	MORUS1280 with random IV, AEAD [86]

Table A.4: LUKS2 encryption algorithms examples.
AEAD algorithms are experimental and require dm-integrity [120] support.

Glossary

AEAD Authenticated Encryption with Additional Data.

AF Anti-Forensic splitter defined for LUKS1. [10, 50]

Base64 Binary to text encoding scheme. [121]

blkid Utility to locate and print block device attributes.

dm-crypt Linux device-mapper crypto target. [62]

dm-integrity Linux device-mapper integrity target [120].

IV Initialization Vector for an encryption mode that tweaks encryption.

JSON JavaScript Object Notation (data-interchange format). [117]

Keyslot Encrypted area on disk that contains a key.

Length-preserving encryption Symmetric encryption where plaintext and ciphertext have the same size.

libcryptsetup Library implementing LUKS1 and LUKS2. [11]

Metadata locking A way how to serialize access to on-disk metadata updates.

PBKDF Password-Based Key Derivation Function.

RNG Cryptographically strong Random Number Generator.

Sector Atomic unit for block device (disk). Typical sector size is 4096 bytes.

TRIM Command that informs a block device that area of the disk is unused and can be discarded.

udev Device manager for Linux kernel implemented in userspace.

UUID Universally Unique IDentifier (of a block device).

Volume Key The key used for data encryption on disk. Sometimes called as Media Encryption Key (MEK).

B Author's Publications

B.1 Journals

- Milan Brož and Vashek Matyáš. The TrueCrypt On-Disk Format – An Independent View. In *IEEE Security & Privacy*, 2014, vol. 12, No 3, p. 74-77.

  [Author's contribution: 90%]

- Andrea Visconti, Ondrej Mosnáček, Milan Brož, Václav Matyáš. Examining PBKDF2 security margin — case study of LUKS. In *Journal of Information Security and Applications*.

  [Author's contribution: 20%] (Accepted for publication.)

B.2 International Conferences and Workshops

- Milan Brož, Mikuláš Patočka and Vashek Matyáš. Practical Cryptographic Data Integrity Protection with Full Disk Encryption. In *IFIP SEC 2018: ICT Systems Security and Privacy Protection*, Springer International Publishing, 2018. p. 79-93.

  [Author's contribution: 80%]

- Milan Brož. Extending Full Disk Encryption for the Future. In *Security Protocols XXV: 25th International Workshop*, Cambridge, UK, March 20-22, 2017, Revised Selected Papers. Cham: Springer International Publishing, 2017. p. 109-115.

  [Author's contribution: 100%]

- Milan Brož and Vashek Matyáš. Selecting a new key derivation function for disk encryption. In *Security and Trust Management, 11th International Workshop, STM 2015*, LNCS 9331. Switzerland: Springer International Publishing Switzerland, 2015. p. 185-199.

  [Author's contribution: 90%]

B.3 National Conferences

- Milan Brož and Ondrej Mosnáček. Password hashing reloaded. In *EurOpen 2016 sborník příspěvků*, Plzeň: EurOpen.cz, 2016. p. 15-22.

  [Author's contribution: 80%]

- Milan Brož. Šifrování disků a TrueCrypt (Disk Encryption and TrueCrypt). In *Europen 2013 sborník příspěvků*. Plzeň: EurOpen.cz, 2013. p. 117-128.

  [Author's contribution: 100%]

- Milan Brož. Šifrování disků (nejen) v Linuxu (Disk encryption (not only) in Linux) In *Europen 2011 sborník příspěvků*. Plzeň: EurOpen.CZ, 2011. p. 57-68.

  [Author's contribution: 100%]

B.4 Technical Reports

- Milan Brož, Mikuláš Patočka and Vashek Matyáš. Practical Cryptographic Data Integrity Protection with Full Disk Encryption Extended Version In *arXiv:1807.00309*. 2018.

  [Author's contribution: 80%]

- Milan Brož. Password Hashing Competition second round candidates — tests report (version 3). `https://github.com/mbroz/PHCtest/blob/master/output/phc`, 2015.

  [Author's contribution: 100%]

Bibliography

1. ALENDAL, Gunnar; KISON, Christian; MODG. *got HW crypto? On the (in)security of a Self-Encrypting Drive series*. 2015.
2. SAARINEN, Markku-Juhani O. Encrypted Watermarks and Linux Laptop Security. In: *Workshop on Information Security Applications, Revised Selected Papers*. 2005.
3. BROŽ, Milan; MATYÁŠ, Václav. The TrueCrypt On-Disk Format – An Independent View. *IEEE Security & Privacy*. 2014, vol. 12. ISSN 1540–7993.
4. BROŽ, Milan; PATOČKA, Mikuláš; MATYÁŠ, Vashek. Practical Cryptographic Data Integrity Protection with Full Disk Encryption. In: *SEC 2018: ICT Systems Security and Privacy Protection*. Cham: Springer International Publishing, 2018, pp. 79–93.
5. BROŽ, Milan. Extending Full Disk Encryption for the Future. In: *Security Protocols XXV - 25th International Workshop, Cambridge, UK, March 20-22, 2017, Revised Selected Papers*. 2017, pp. 109–115.
6. BROŽ, Milan; PATOČKA, Mikuláš; MATYÁŠ, Vashek. Practical Cryptographic Data Integrity Protection with Full Disk Encryption Extended Version. *ArXiv e-prints*. 2018. Available from arXiv: `1807.00309 [cs.CR]`.
7. *Password Hashing Competition*. 2015. `https://password-hashing.net/`.
8. BROŽ, Milan; MATYÁŠ, Vashek. Selecting a New Key Derivation Function for Disk Encryption. In: *Security and Trust Management - 11th International Workshop, STM 2015, Vienna, Austria, September 21-22, 2015, Proceedings*. 2015, pp. 185–199.
9. BROŽ, Milan. *Benchmarks of the PHC candidates*. 2015. `https://github.com/mbroz/PHCtest/blob/master/output/phc_round2.pdf`.
10. *Linux Unified Key Setup (LUKS1) On-Disk Format Specification, Version 1.2.3*. 2018. `https://gitlab.com/cryptsetup/cryptsetup/wikis/Specification`.
11. *Cryptsetup and LUKS utility*. 2018. `https://gitlab.com/cryptsetup/cryptsetup`.

12. *Storage Work Group Storage Security Subsystem Class: Opal*. 2014. Trusted Computing Group, `http://www.trustedcomputinggroup.org/resources/storage_work_group_storage_security_subsystem_class_opal`.

13. MÜLLER, Tilo; FREILING, Felix. A Systematic Assessment of the Security of Full Disk Encryption. In: *Transactions on Dependable and Secure Computing (TDSC)*. 99th ed. 2014, vol. PP.

14. FRUHWIRTH, Clemens. *New methods in hard disk encryption*. 2005. PhD thesis. Institute for Computer Languages Theory and Logic Group Vienna University of Technology. `http://clemens.endorphin.org/publications`.

15. KORNBLUM, Jesse D. Implementing BitLocker Drive Encryption for Forensic Analysis. *Digital Investigation*. 2009, vol. 5, no. 3, pp. 75–84.

16. MÜLLER, Tilo; FREILING, Felix C.; DEWALD, Andreas. TRESOR Runs Encryption Securely Outside RAM. In: *Proceedings of the 20th USENIX Conference on Security*. San Francisco, CA: USENIX Association, 2011, pp. 17–17. SEC'11.

17. MÜLLER, Tilo; LATZO, Tobias; FREILING, Felix. *Self-Encrypting Disks pose Self-Decrypting Risks: How to break Hardware-based Full Disk Encryption*. 2012. Technical report. Friedrich-Alexander-Universität Erlangen-Nürnberg. `https://www1.informatik.uni-erlangen.de/filepool/projects/sed/seds-at-risks.pdf`.

18. *FIPS PUB 140-2, Security Requirements for Cryptographic Modules*. 2002. U.S.Department of Commerce/National Institute of Standards and Technology.

19. *Algorithms, key size and parameters report – 2014*. 2014. Technical report. European Union Agency for Network and Information Security. `https://www.enisa.europa.eu/activities/identity-and-trust/library/deliverables/algorithms-key-size-and-parameters-report-2014/`.

20. OIKAWA, Shuichi. Integrating Memory Management with a File System on a Non-volatile Main Memory System. In: *Proceedings of the 28th Annual ACM Symposium on Applied Computing*. Coimbra, Portugal: ACM, 2013, pp. 1589–1594. SAC '13. ISBN 978-1-4503-1656-9.

21. CHEN, Feng; MESNIER, Michael P.; HAHN, Scott. A protected block device for Persistent Memory. In: *30th Symposium on Mass Storage Systems and Technologies (MSST)*. 2014, pp. 1–12.

22. TS'O, Theodore. [*LSF/MM TOPIC*] *really large storage sectors – going beyond 4096 bytes*. 2014. `http://lwn.net/Articles/582881/`.

23. PETERSEN, Martin K. *T10 Data Integrity Feature* (*Logical Block Guarding*). 2007. Linux Storage & Filesystem Workshop.

24. BAIRAVASUNDARAM, Lakshmi N.; ARPACI-DUSSEAU, Andrea C.; ARPACI-DUSSEAU, Remzi H.; GOODSON, Garth R.; SCHROEDER, Bianca. An Analysis of Data Corruption in the Storage Stack. *ACM Transactions on Storage*. 2008, vol. 4, no. 3. ISSN 1553-3077.

25. BROŽ, Milan. *TRIM & dm-crypt ... problems?* 2011. `http://asalor.blogspot.com/2011/08/trim-dm-crypt-problems.html`.

26. EMC EDUCATION SERVICES. *Information Storage and Management: Storing, Managing, and Protecting Digital Information in Classic, Virtualized, and Cloud Environments*. Wiley Publishing, 2012. ISBN 978-1-118-09483-9.

27. GUTMANN, Peter. Secure Deletion of Data from Magnetic and Solid-state Memory. In: *Proceedings of the 6th Conference on USENIX Security Symposium, Focusing on Applications of Cryptography – Volume 6*. San Jose, California: USENIX Association, 1996, pp. 8–8. SSYM'96. Online version `https://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html`.

28. GUTMANN, Peter. Data Remanence in Semiconductor Devices. In: *Proceedings of the 10th Conference on USENIX Security Symposium – Volume 10*. Washington, D.C.: USENIX Association, 2001. SSYM'01.

29. INSTITUTE, American National Standards. *ANSI INCITS 482–2012, Information Technology – ATA/ATAPI Command Set – 2* (*ACS-2*). 2012.

30. WEI, Michael; GRUPP, Laura M.; SPADA, Frederick E.; SWANSON, Steven. Reliably Erasing Data from Flash-based Solid State Drives. In: *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*. San Jose, California: USENIX Association, 2011, pp. 8–8. FAST'11. ISBN 978-1-931971-82-9.

31. FERGUSON, Niels. *AES-CBC + Elephant diffuser: A disk encryption algorithm for Windows Vista*. 2006. Available also from: `http://download.microsoft.com/download/0/2/3/0238acaf-d3bf-4a6d-b3d6-0a0be4bbb36e/BitLockerCipher200608.pdf`. Technical report. Microsoft Corporation.

32. *Linux Kernel: Add support for the Speck block cipher*. 2018. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=da7a0ab5b4babbe5d7a46f852582be06a00a28f0`.

33. BIGGERS, Eric. [*RFC PATCH 0/9*] *crypto: HPolyC support*. 2018. `https://marc.info/?l=linux-crypto-vger&m=153359499015659`.

34. CROWLEY, Paul; BIGGERS, Eric. *HPolyC: length-preserving encryption for entry-level processors* [Cryptology ePrint Archive, Report 2018/720]. 2018. `https://eprint.iacr.org/2018/720`.

35. ROGAWAY, Phillip. *Evaluation of Some Blockcipher Modes of Operation*. 2011. Technical report. University of California, Davis. `http://web.cs.ucdavis.edu/~rogaway/papers/modes.pdf`.

36. *FIPS Publication 197, The Advanced Encryption Standard (AES)*. 2001. U.S. DoC/NIST.

37. SCHNEIER, Bruce. *Applied Cryptography (2nd Ed.): Protocols, Algorithms, and Source Code in C*. New York, NY, USA: John Wiley & Sons, Inc., 1995. ISBN 0-471-11709-9.

38. ROSENDORF, Dan. *Bitlocker: A little about the internals and what changed in Windows 8*. 2013.

39. DWORKIN, Morris J. *SP 800-38E. Recommendation for Block Cipher Modes of Operation: The XTS-AES Mode for Confidentiality on Storage Devices*. Gaithersburg, MD, United States, 2010. Technical report. National Institute of Standards & Technology.

40. *IEEE 1619-2007 – Standard for Cryptographic Protection of Data on Block-Oriented Storage Devices*. IEEE, 2008.

41. ROGAWAY, Phillip. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In: *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5–9, 2004, Proceedings*. Springer, 2004, vol. 3329, pp. 16–31. Lecture Notes in Computer Science. Available also from: `http://www.iacr.org/cryptodb/archive/2004/ASIACRYPT/426/426.pdf`.

42. *Public Comments on the XTS-AES Mode*. 2008. `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/XTS/collected_XTS_comments.pdf`.

43. BROŽ, Milan. *Data integrity protection with cryptsetup tools*. 2018. FOSDEM18, `https://fosdem.org/2018/schedule/event/cryptsetup/`.

44. FERGUSON, Niels; SCHNEIER, Bruce; KOHNO, Tadayoshi. *Cryptography Engineering: Design Principles and Practical Applications*. Wiley Publishing, 2010. ISBN 978-0-470-47424-2.

45. RUUSU, Jari. *Re: gonzo cryptography; how would you improve existing cryptosystems?* 2005. `https://www.mail-archive.com/cryptography@metzdowd.com/msg05189.html`.

46. HALEVI, Shai; ROGAWAY, Phillip. A Parallelizable Enciphering Mode. In: *Topics in Cryptology – CT-RSA 2004*. Springer Berlin Heidelberg, 2004, vol. 2964, pp. 292–304. Lecture Notes in Computer Science. ISBN 978-3-540-20996-6.

47. HALEVI, Shai. *EME*: extending EME to handle arbitrary-length messages with associated data* [Cryptology ePrint Archive, Report 2004/125]. 2004.

48. HALEVI, Shai; ROGAWAY, Phillip. A Tweakable Enciphering Mode. In: *Advances in Cryptology – CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. Springer, 2003, vol. 2729, pp. 482–499. Lecture Notes in Computer Science. Available also from: `http://www.iacr.org/cryptodb/archive/2003/CRYPTO/1385/1385.pdf`.

49. PESLYAK, Alexander; MARECHAL, Simon. *Password security: past, present, future (with strong bias towards password hashing)*. 2012. Available also from: `http://www.openwall.com/presentations/Passwords12-The-Future-Of-Hashing/Passwords12-The-Future-Of-Hashing.pdf`.

50. FRUHWIRTH, Clemens. *TKS1 – An anti-forensic, two level, and iterated key setup scheme*. 2004. `http://clemens.endorphin.org/publications`.

51. CANETTI, Ran; DWORK, Cynthia; NAOR, Moni; OSTROVSKY, Rafail. Deniable Encryption. In: *CRYPTO*. Springer, 1997, vol. 1294, pp. 90–104. Lecture Notes in Computer Science. ISBN 3-540-63384-7.

52. SKILLEN, Adam; MANNAN, Mohammad. On Implementing Deniable Storage Encryption for Mobile Devices. In: *The Network and Distributed System Security Symposium*. The Internet Society, 2013.

53. TURAN, Meltem Sönmez; BARKER, Elaine B.; BURR, William E.; CHEN, Lidong. *SP 800-132. Recommendation for Password-Based Key Derivation: Part 1: Storage Applications*. Gaithersburg, MD, United

States, 2010. Technical report. National Institute of Standards & Technology.

54. BIRYUKOV, Alex; KHOVRATOVICH, Dmitry; GROZSCHAEDL, Johann. *Tradeoff cryptanalysis of password hashing schemes*. 2014. `https://www.cryptolux.org/images/5/57/Tradeoffs.pdf`.

55. KALISKI, Burt. *PKCS #5: Password-Based Cryptography Specification Version 2.0* [RFC 2898 (Informational)]. IETF, 2000. Request for Comments, no. 2898. Available also from: `http://www.ietf.org/rfc/rfc2898.txt`.

56. DÜRMUTH, Markus; GÜNEYSU, Tim; KASPER, Markus; PAAR, Christof; YALCIN, Tolga; ZIMMERMANN, Ralf. Evaluation of Standardized Password-Based Key Derivation against Parallel Processing Platforms. In: *Computer Security – ESORICS 2012*. Springer Berlin Heidelberg, 2012, vol. 7459, pp. 716–733. Lecture Notes in Computer Science. ISBN 978-3-642-33166-4.

57. PERCIVAL, Colin; JOSEFSSON, Simon. *The scrypt Password-Based Key Derivation Function.* [RFC 7914]. IETF, 2016. Request for Comments, no. 7914.

58. PERCIVAL, Colin. *Stronger Key Derivation via Sequential Memory-Hard Functions*. 2009. `http://www.tarsnap.com/scrypt/scrypt.pdf`.

59. *Android Secure Boot*. 2014. `https://source.android.com/devices/tech/security/secureboot/index.html`.

60. *TrueCrypt*. 2018. `https://en.wikipedia.org/wiki/TrueCrypt`.

61. *VeraCrypt*. 2018. `https://www.veracrypt.fr`.

62. *dm-crypt: Linux device-mapper crypto target*. 2018. `https://gitlab.com/cryptsetup/cryptsetup/wikis/DMCrypt`.

63. HORNUNG, Alex. *tc-play*. 2014. `https://github.com/bwalex/tc-play`.

64. CZESKIS, Alexei; HILAIRE, David J. St.; KOSCHER, Karl; GRIBBLE, Steven D.; KOHNO, Tadayoshi; SCHNEIER, Bruce. Defeating Encrypted and Deniable File Systems: TrueCrypt V5.1a and the Case of the Tattling OS and Applications. In: *Proceedings of the 3rd Conference on Hot Topics in Security*. San Jose, CA: USENIX Association, 2008, 7:1–7:7. HOTSEC'08.

65. ANDERSON, Ross; BIHAM, Eli; KNUDSEN, Lars. *Serpent: A Proposal for the Advanced Encryption Standard*. `https://www.cl.cam.ac.uk/~rja14/serpent.html`.

66. SCHNEIER, Bruce; KELSEY, John; WHITING, Doug; WAGNER, David; HALL, Chris; FERGUSON, Niels. *The Twofish Encryption Algorithm: A 128-bit Block Cipher*. John Wiley & Sons, Inc., 1999. `https://www.schneier.com/academic/twofish/`.

67. AOKI, Kazumaro; ICHIKAWA, Tetsuya; KANDA, Masayuki; MATSUI, Mitsuru; MORIAI, Shiho; NAKAJIMA, Junko; TOKITA, Toshio. Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms — Design andAnalysis. In: STINSON, Douglas R.; TAVARES, Stafford (eds.). *Selected Areas in Cryptography*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 39–56.

68. DOLMATOV, Vasily. *GOST R 34.12-2015: Block Cipher "Kuznyechik"* [RFC 7801]. RFC Editor, 2016. Request for Comments, no. 7801. Available also from: `https://rfc-editor.org/rfc/rfc7801.txt`.

69. KRIOUKOV, Andrew; BAIRAVASUNDARAM, Lakshmi N.; GOODSON, Garth R.; SRINIVASAN, Kiran; THELEN, Randy; ARPACI-DUSSEAU, Andrea C.; ARPACI-DUSSEA, Remzi H. Parity Lost and Parity Regained. In: *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. USENIX Association, 2008.

70. TÜRPE, Sven; POLLER, Andreas; STEFFAN, Jan; STOTZ, Jan-Peter; TRUKENMÜLLER, Jan. Attacking the BitLocker Boot Process. In: *Proceedings of the 2nd International Conference on Trusted Computing*. Berlin, Heidelberg: Springer-Verlag, 2009. Trust '09.

71. MARTIN, Thomas Anthony; JONES, Andy; ALZAABI, Mohammed. The 2016 Analysis of Information Remaining on Computer Hard Disks Offered for Sale on the Second Hand Market in the UAE. *Journal of Digital Forensics, Security and Law*. 2016, vol. 4, no. 4.

72. ERIK RIEDEL, Mahesh Kallahalla; SWAMINATHAN, Ram. A Framework for Evaluating Storage System Security. In: *Proceedings of the 1st USENIX Conference on File and Storage Technologies*. USENIX Association, 2002, vol. 2.

73. TISCHER, Matthew; DURUMERIC, Zakir; FOSTER, Sam; DUAN, Sunny; MORI, Alec; BURSZTEIN, Elie; BAILEY, Michael. Users Really Do Plug in USB Drives They Find. In: *2016 IEEE Symposium on Security and Privacy (SP)*. 2016.

74. SHEINWALD, Dafna; SATRAN, Julian; THALER, Patricia; CAVANNA, Vincente. *Internet Protocol Small Computer System Interface (iSCSI) Cyclic Redundancy Check (CRC)/Checksum Considerations* [RFC 3385]. RFC Editor, 2015. Request for Comments, no. 3385.

75. DIJK, Marten van; RHODES, Jonathan; SARMENTA, Luis F. G.; DEVADAS, Srinivas. Offline Untrusted Storage with Immediate Detection of Forking and Replay Attacks. In: *Proceedings of the 2007 ACM Workshop on Scalable Trusted Computing*. ACM, 2007. STC '07.

76. GJØSTEEN, Kristian. Security Notions for Disk Encryption. In: *ESORICS 2005: 10th European Symposium on Research in Computer Security*. Springer Berlin Heidelberg, 2005.

77. KHATI, Louiza; MOUHA, Nicky; VERGNAUD, Damien. Full Disk Encryption: Bridging Theory and Practice. In: *Topics in Cryptology – CT-RSA 2017: The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14–17, 2017, Proceedings*. Springer International Publishing, 2017.

78. BELLARE, Mihir; NAMPREMPRE, Chanathip. Authenticated Encryption: Relations Among Notions and Analysis of the Generic Composition Paradigm. *Journal of Cryptology*. 2008, vol. 21, no. 4.

79. ROGAWAY, Phillip. Authenticated-encryption with Associated-data. In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. Washington, DC, USA, 2002.

80. DWORKIN, Morris J. *SP 800-38D. Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. Gaithersburg, MD, United States, 2007. Technical report. National Institute of Standards & Technology.

81. BÖCK, Aaron Zauner; DEVLIN, Sean. Nonce-Disrespecting Adversaries: Practical Forgery Attacks on GCM in TLS. *IACR Cryptology ePrint Archive*. 2016, vol. 2016.

82. BERNSTEIN, Daniel J. *ChaCha, a variant of Salsa20*. 2008. `https://cr.yp.to/chacha/chacha-20080120.pdf`.

83. NIR, Yoav; LANGLEY, Adam. *ChaCha20 and Poly1305 for IETF Protocols* [RFC 7539]. RFC Editor, 2015. Request for Comments, no. 7539.

84. *CAESAR: Competition for Authenticated Encryption: Security, Applicability, and Robustness*. 2018. `https://competitions.cr.yp.to/caesar.html`.

85. WU, Hongjun; PRENEEL, Bart. *AEGIS, A Fast Authenticated Encryption Algorithm (v1.1)*. 2016. Technical report. `https://competitions.cr.yp.to/round3/aegisv11.pdf`.

86. WU, Hongjun; HUANG, Tao. *The Authenticated Cipher MORUS (v2)*. 2016. Technical report. `https://competitions.cr.yp.to/round3/morusv2.pdf`.

87. GUERON, Shay; LINDELL, Yehuda. *GCM-SIV: Full Nonce Misuse-Resistant Authenticated Encryption at Under One Cycle per Byte* [Cryptology ePrint Archive, Report 2015/102]. 2015.

88. SIVATHANU, Gopalan; WRIGHT, Charles P.; ZADOK, Erez. Ensuring Data Integrity in Storage: Techniques and Applications. In: *Proceedings of the 2005 ACM Workshop on Storage Security and Survivability*. Fairfax, VA, USA: ACM, 2005, pp. 26–36. StorageSS '05. ISBN 1-59593-233-X.

89. CHAKRABORTY, Debrup; LÓPEZ, Cuauhtemoc Mancillas; SARKAR, Palash. *Disk Encryption: Do We Need to Preserve Length?* [Cryptology ePrint Archive, Report 2015/594]. 2015.

90. HOLT, Keith. *End-to-End Data Protection Justification*. 2003. Available also from: `http://www.t10.org/ftp/t10/document.03/03-224r0.pdf`. T10 Technical Committee proposal letter.

91. BUTLER, Kevin; MCLAUGHLIN, Stephen; MCDANIEL, Patrick. Disk-enabled authenticated encryption. In: *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 2010.

92. KAMP, Poul-Henning. GBDE: GEOM Based Disk Encryption. In: *Proceedings of the BSD Conference 2003 on BSD Conference*. San Mateo, California: USENIX, 2003.

93. DAWIDEK, Pawel Jakub. *FreeBSD GELI encryption system*. 2011. `http://svnweb.freebsd.org/base/head/sys/geom/eli/g_eli_integrity.c?view=co`.

94. DOWDESWELL, Roland C. *Initial Analysis of GBDE*. 2003. `http://www.imrryr.org/~elric/cgd/gbde-analysis2.pdf`.

95. *Linux mainline kernel archive*. 2018. `https://kernel.org`.

96. KARIO, Hubert. *RAID doesn't work!* 2018. `https://securitypitfalls.wordpress.com/2018/05/08/raid-doesnt-work`.

97. IEEE Standard for Authenticated Encryption With Length Expansion for Storage Devices. *IEEE Std 1619.1-2007*. 2008.

98. MOSNÁČEK, Ondrej. *Optimizing authenticated encryption algorithms*. 2018. Master's thesis. Masaryk University, Faculty of Informatics, Brno. Supervised by Milan BROŽ. `https://is.muni.cz/th/qvh3t/`.

99. AXBOE, Jens. *Flexible I/O Tester*. 2017. `https://github.com/axboe/fio`.

100. GUERON, Shay; LANGLEY, Adam; LINDELL, Yehuda. *AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption*. IETF, 2018. Internet-Draft.

101. ZHANG, Yupu; RAJIMWALE, Abhishek; ARPACI-DUSSEAU, Andrea C.; ARPACI-DUSSEAU, Remzi H. End-to-end Data Integrity for File Systems: A ZFS Case Study. In: *Proceedings of the 8th USENIX Conference on File and Storage Technologies*. USENIX Association, 2010.

102. BERTONI, G.; DAEMEN, J.; PEETERS, M.; ASSCHE, G. Van. *Sponge functions* [Ecrypt Hash Workshop 2007]. 2007.

103. AUMASSON, Jean-Philippe; NEVES, Samuel; WILCOX-O'HEARN, Zooko; WINNERLEIN, Christian. BLAKE2: Simpler, Smaller, Fast As MD5. In: *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*. Banff, AB, Canada: Springer-Verlag, 2013, pp. 119–135. ACNS'13. ISBN 978-3-642-38979-5.

104. HATZIVASILIS, George; PAPAEFSTATHIOU, Ioannis; MANIFAVAS, Charalampos. *Password Hashing Competition - Survey and Benchmark* [Cryptology ePrint Archive, Report 2015/265]. 2015.

105. BIRYUKOV, Alex; KHOVRATOVICH, Dmitry. *Argon and Argon2*. 2015. Available also from: `https://password-hashing.net/submissions/specs/Argon-v2.pdf`.

106. ALWEN, Joel; BLOCKI, Jeremiah. *Efficiently Computing Data-Independent Memory-Hard Functions* [Cryptology ePrint Archive, Report 2016/115]. 2016. `https://eprint.iacr.org/2016/115`.

107. THOMAS, Steven. *battcrypt (Blowfish All The Things)*. 2014. Available also from: `https://password-hashing.net/submissions/specs/battcrypt-v0.pdf`.

108. FORLER, Christian; LUCKS, Stefan; WENZEL, Jakob. *The Catena Password-Scrambling Framework*. 2015. Available also from: `https://password-hashing.net/submissions/specs/Catena-v5.pdf`.

109. SIMPLICIO, Marcos A. Jr.; ALMEIDA, Leonardo C.; ANDRADE, Ewerton R.; SANTOS, Paulo C. F. dos; BARRETO, Paulo S. L. M. *The Lyra2 reference guide*. 2015. Available also from: `https://password-hashing.net/submissions/specs/Lyra2-v3.pdf`.

110. PESLYAK, Alexander. *yescrypt – a Password Hashing Competition submission*. 2014. Available also from: `https://password-hashing.net/submissions/specs/yescrypt-v0.pdf`.

111. PORNIN, Thomas. *The MAKWA Password Hashing Function*. 2014. Available also from: `https://password-hashing.net/submissions/specs/Makwa-v0.pdf`.

112. THOMAS, Steven. *Parallel*. 2014. Available also from: `https://password-hashing.net/submissions/specs/Parallel-v0.pdf`.

113. WU, Hongjun. *POMELO: A Password Hashing Algorithm*. 2014. Available also from: `https://password-hashing.net/submissions/specs/POMELO-v1.pdf`.

114. GOSNEY, Jeremi M. *The Pufferfish Password Hashing Scheme*. 2014. Available also from: `https://password-hashing.net/submissions/specs/Pufferfish-v0.pdf`.

115. COX, Bill. *Added multi-threading support to test suite* [PHC mailing list archive]. 2015. `http://article.gmane.org/gmane.comp.security.phc/2915`.

116. THOMAS, Steve. *PHC: survey and benchmarks*. 2015. `http://article.gmane.org/gmane.comp.security.phc/2550`.

117. *The JSON Data Interchange Format*. 2013. Technical report. ECMA. `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

118. BIRYUKOV, Alex; DINU, Daniel; KHOVRATOVICH, Dmitry; JOSEFSSON, Simon. *The memory-hard Argon2 password hash and proof-of-work function*. Internet Engineering Task Force, 2018. Available also from: `https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-argon2-03`. Internet-Draft. Internet Engineering Task Force. Work in Progress.

119. *Linux Kernel documentation, memory overcommit accounting*. 2018. `https://www.kernel.org/doc/Documentation/vm/overcommit-accounting`.

120. *dm-integrity: Linux device-mapper integrity target*. 2018. `https://gitlab.com/cryptsetup/cryptsetup/wikis/DMIntegrity`.

121. JOSEFSSON, Simon. *The Base16, Base32, and Base64 Data Encodings* [RFC 4648]. RFC Editor, 2006. Request for Comments, no. 4648. `https://www.ietf.org/rfc/rfc4648.txt`.