Length-preserving authenticated encryption of storage blocks

Jasper Surmont

School of Science

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 21.11.2022

Supervisor

Adj. Prof. Jan-Erik Ekberg

Advisor

Msc. Arto Niemi

Aalto University
School of Science

Acknowledgements

This work would not have been possible without the support of my advisor, Arto Niemi. Not only did he read and correct many revisions, he also guided me throughout the process and supported me during my stay at the Huawei Technologies Research & Development Center. Further, I would like to thank my supervisor Jan-Erik Ekberg, who gave me the opportunity to perform my work at the R&D center and helped me during my stay there.

Finally, I would like to thank my parents and Leah for supporting me throughout this experience.

Aalto University
School of Science

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Jasper Surmont

Title Length-preserving authenticated encryption of storage blocks

Degree programme Computer, Communication and Information Sciences

Major Exchange studies in Computer Science **Code of major** SCI3042

Supervisor Adj. Prof. Jan-Erik Ekberg

Advisor Msc. Arto Niemi

Date 21.11.2022 **Number of pages** 79+1 **Language** English

Abstract

Digital storage is often protected by individually authenticating and encrypting each storage unit, usually a disk block or a memory page. This results in one ciphertext and authentication tag per unit. Where used, these tags are written to external memory locations or to different blocks within the same device, but this has two main drawbacks. First, it is not always possible to use external memory, or to allocate extra blocks to store the tags. Second, storing the tag in a different location than the ciphertext requires two IO requests for each read or write: one request for the ciphertext, another for the tag.

In this thesis, I ask and resolve the question: is it possible to use data compression to provide length-preserving storage protection, providing integrity and confidentiality, removing the need for external storage or extra blocks.

The thesis contributes to the research of block-level data protection, and analyses existing protection methods for data protection of block devices, such as dm-crypt and dm-verity in Linux, as well as RAM protections such as AMD SEV-SNP. Previous compression-based solutions are analysed and found not to be fully length-preserving.

The thesis presents LP-SP, a length-preserving storage protection method that does not need external storage or extra blocks for tags. Additionally, a prototype implementation in the device-mapper in Linux provides compression and performance measurements. These measurements result in LP-SP being especially useful in RAM and other environments with high compression rates.

Keywords Data Integrity, Compression, Encryption, FDE

Contents

Abbreviations and definitions

Abbreviations

HDD	Hard Disk Drive
SSD	Solid State Drive
KB, MB, GB, TB	Kilo-, Mega-, Giga-, TeraByte
OS	Operating System
CPU	Central Processing Unit
I/O	Input/Output
Bit	Binary Digit (0 or 1)
RLE	Run Length Encoding
MTF	Move-to-Front
OWF	One-way Function
PPT	Probabilistic Polynomial Time
CHF	Cryptographic Hash Function
SHA	Secure Hash Algorithm
AES	Advanced Encryption Standard
UUID	Universal Unique Identifier
PKCS	Public Key Cryptography Standard
MAC	Message Authentication Code
HMAC	Hash based MAC
RW, RO	Read-Write, Read-Only
dm	device-mapper
FDE	Full Disk Encryption
LUKS	Linux Unified Key Setup

Definitions

Generic
Not specific to a single thing: a generic thing can be applied to not only a single use case, but to a whole family with some characteristic.

Byte
A group of 8 bits.

Octet
See Byte.

Access time
The time it takes to complete a processor's request to get data [41].

Codeword
The result of an encoding.

Parallelism
The act of performing multiple computations simultaneously, often resulting in faster computing speeds.

XOR
Exclusive OR (symbol $\oplus$): a bitwise XOR results in 1 if only one of the two values is 1, and 0 otherwise. The XOR of two numbers is the bitwise XOR on every bit. For example:
$5_{10} \oplus 9_{10} = 0101_2 \oplus 1001_2 = 1100_2 = 12_{10}$

Independent and identically distributed variables
Every variable is an independent draw from a fixed probabilistic model, i.e. the probability of an occurrence of an event x_1 is independent whether another event x_2 happened or not: $P(x_1|x_2) = P(x_1)$ [17].

Decision Problem
A problem that can be posed as a yes-no question. The only valid outputs are YES or NO.

Deterministic algorithm
An algorithm that always provides the same output for the same input; a *predictable* algorithm. On the other hand, a nondeterministic algorithm can provide different outputs for the same input; a *non-predictable* algorithm.

Big O notation: $\mathcal{O}(\cdot)$	It describes how a function behaves if the input goes to infinity, i.e. a function with $\mathcal{O}(g(n))$ means it is limited by the function $f(n) = M \cdot g(n)$ as n goes towards infinity, with $M \in \mathbb{R}$.
Probabilistic Polynomial Time (PPT)	If a decision problem is in PPT, there exists an nondeterministic algorithm that runs in polynomial time. This algorithm returns YES on a YES input with probability greater than $\frac{1}{2}$, and returns YES on a NO input with probability less than or equal to $\frac{1}{2}$.
Negligible function	[Informal definition] A function f with input x that is smaller than any polynomial function, for sufficiently big x. [Formal definition] A function f with as property that for every integer c, there exists an integer N_c such that $\forall x > N_c$: $\lvert f(x)\rvert < \frac{1}{x^c}$

1 Introduction

The rapidly evolving digital world has caused a huge shift in the everyday life. Jobs that were slow and tedious have been replaced by automatic digital processes, old-fashioned archives with books in shelves have been transformed in massive databases, and people from all over the world can connect with anyone almost instantly. One factor that allows for such a fast growing digitalisation is the ability to store massive amounts of data in relatively small physical devices. These devices are found everywhere: desktops, laptops, phones, smartwatches, databases, TVs and even lamps. However, having so much data in such everyday devices can be dangerous. Physical protection techniques like protecting an archive in a secured building from intruders are not always possible or sufficient anymore. Instead, newer techniques try to also protect the actual data, rather than just the device itself.

In the context of storage devices, there are two types of entities: those who should have access to the data (*authorized*), and those who should not (*unauthorized*). This thesis focuses on providing *authentication*, *confidentiality* and *integrity*. Authentication ensures that authorized writers and readers can identify each other to make sure the other one is not an attacker who claims to be authorized. Confidentiality ensures that unauthorized entities do not gain any new information from analysing the protected data. It thus provides a way to store private information on an insecure disk, preventing attackers who observe the disk from extracting this private data. Integrity ensures that data is authentic, which means that the data one reads must be the original data that was written. To provide confidentiality and integrity, the data has to be transformed and/or new data needs to be created. Confidentiality can be provided by *encrypting* the message, transforming it into data that looks like gibberish. The encrypted data can be *decrypted* bu authorized entities back to the original message. Integrity and authentication can overlap, and can both be provided by creating *integrity metadata*. This integrity metadata can be used in conjunction with the message to verify integrity and authentication.

When these former techniques are applied on storage units, i.e. physical devices in a computer that store data, some drawbacks become apparent. Storage devices can be written to and read from in *blocks*: the smallest addressable size of data on the device. It can be desired to protect the data on this device block by block, called block-level protection. The techniques described above expand the data: they create integrity metadata that also needs to be stored somewhere to be able to verify the

block. However, if the block is already full with the data that needs the protection, there is no more room left in the block to store the integrity metadata. Previous solutions solved this by storing this extra data elsewhere (somewhere else in the same device, or in a different device). In some environments, these solutions are not always efficient or even possible. In those cases, weaker protection schemes can be used, which does not always meet the required security properties.

This thesis tries to find improvements in cases where previous solutions are not possible or sufficient, and where a strong protection scheme is still required. We describe LP-SP: a solution to provide block-level, length-preserving storage protection. LP-SP can be useful in any block-like storage devices, like HDD, SSD and RAM. Additionally, migration (moving processes from one system to another) might benefit from it as well.

LP-SP is able to preserve length by compressing the input data; the space that the compression frees up can then be used to store the integrity metadata. The idea to use compression is not new [63] [62]. However, not all data is compressible. Not all data is compressible, which means storing integrity metadata and IVs is not always possible. Previous solutions that use compression to provide block-level protection, still require some external, trusted storage to protect blocks that are not compressible enough (called *incompressible blocks*). LP-SP does not need any external storage, and is thus the first complete length-preserving method that provides authentication, confidentiality and integrity. A part of LP-SP is also implemented as a proof-of-concept block device in the Device-Mapper layer in Linux, and is then called dm-ci: device-mapper **c**onfidentiality **i**ntegrity. Additionally, a tool called *cisetup* is implemented that provides two functions. First, it can check disks and files for compression statistics. Second, it can format a disk to be used by *dm-ci*.

1.1 Personal contributions

The thesis work was for the most part performed while the author was an intern at Helsinki System Security Laboratory of Huawei Technologies Oy. The author's individual contributions are:

- Designed and wrote this thesis
- Wrote all of the implementation code for LP-SP and dm-ci
- Wrote all of the implementation code of cisetup
- Researched and experimented with the compression algorithms
- Researched and experimented with previous block-level storage protection methods
- Co-invented the length-reserving storage protection method and contributed to the patent application (PCT/EP2022/2080427)

2 Background

This section covers the basics of compression, cryptography, storage devices, operating systems and filesystems. Furthermore, it covers more specific topics like the device-mapper and Merkle trees.

2.1 Compression

Compression is the act of transforming data such that its size shrinks. Compression algorithms can be lossy or lossless. Lossy compression algorithms lose information during the compression, whereas the latter allow reproducing the original encoding from the compressed one. Lossy algorithms are useful where some data is unnecessary or not important. An example is a sound file, where frequencies that are not perceived by the human ear can be removed. [69]

Compressed data has a certain compression rate. The compression rate x is computed as $\frac{\text{original size}}{\text{compressed size}}$. The compression rate depends on the data, the compression methods, and their parameters. In general, lossy algorithms have a far higher compression rate than lossless algorithms. Another useful statistic is bits per character (bpc). It is computed as $bpc = \frac{\text{amount bits in compressed data}}{\text{amount characters in original data}}$. Since blocks and pages of storage units must be reproduced exactly we will only look at lossless compression.

Two main tasks arise when trying to compress data: *modelling* and *entropy coding*. The task of the former is to reveal redundancies among the data, as well as to produce elements and related statistics or probabilities for coding. The task of the latter is to then remove this revealed redundancy. An important thing to notice is that the modelling and coding techniques are independent, i.e. in some compression implementations, the modelling technique can be substituted for a different one without changing the coding technique, and equivalently for the coding technique. [69] [61]

2.2 Entropy and Noiseless source coding theorem

The noiseless source coding theorem [72] states that there is a limit on the minimal possible *expected* length of a codeword. Take note of the word *expected*: it places a lower bound on the *average* length of a codeword, but it is still possible for a single

codeword to be smaller than this length. The theorem makes use of the *entropy*, which is first explained.

Let X be a set of events $x_1, x_2, ..., x_n$ with probabilities $p_1, p_2, ..., p_n$. The *Shannon entropy* is then defined as

$$H(X) = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

It is a nonnegative number and can be seen as the average information per event. The higher a probability of an event, the less information an occurrence of this event adds[1]. The entropy is dependent on the probabilities of certain events occurring. However, exact probabilities are usually not known, requiring the probabilities to be estimated. Using different probability estimates may result in a different entropy. [72]

The theorem now states that: *one cannot encode X of length n with fewer than $H(X) \cdot n$ bits on average* [34]. The proof is left out as it's not vital to understand the theorem. What is important in this thesis is that it shows that truly random data on average can hardly be compressed. Take as an example the alphabet with 26 letters. If the data we are trying to compress is random[2], each letter has a probability of $\frac{1}{26}$. Thus, the Shannon entropy for a random sequence of letters Y is

$$H(Y) = -\sum_{i=1}^{26} p_i \log_2(p_i) = -\sum_{i=1}^{26} \frac{1}{26} \log_2(\frac{1}{26}) \approx 4.7$$

The minimum amount of bits on average to encode Y of length n is then $\lceil 4.7 \cdot n \rceil$. Assuming the input is a fixed-length binary encoding (what will always be the case in this thesis): 26 characters requires five bits for every character. Hence, in this case, for $n > 1$, the minimum amount of bits is lower than the input. This is because 26 is not a power of two, and a semi-fixed coding could be applied to compress the data slightly.

However, in this thesis, input data is a sequence of bytes (B), with values ranging from 0 until 255 and is assumed to be random. Its Shannon entropy is then

$$H(B) = -\sum_{i=0}^{255} \frac{1}{256} \log_2(\frac{1}{256}) = 8$$

Indeed, this value is equal to the length of a fixed-length binary encoding of 256 characters, so on average a random sequence of bytes is incompressible. This fact is very important and introduces a difficult problem for this thesis' research (see Section 5.2.1).

[1] As an example, take as event *someone on earth took a shower today.* The odds of this happening are incredibly high, so someone telling you: "Someone took a shower today", hardly adds any information. Alternatively, someone telling you: "Nobody on earth took a shower today", adds an enormous amount of information.

[2] Independently and identically distributed

2.3 Modelling

Modelling aims to reveal the redundancy in data. It does this by either using probability estimations, reordering or recoding of symbols, or building dictionaries of subsequences and mapping these to indices. The different modelling techniques used in this thesis are explained more in detail in their respective subsection.

2.3.1 Run-length encoding (RLE)

Opposite of what the name suggests, run-length encoding is a very intuitive *modelling* technique. It scans over the data and measures the length of each run (a sequence of symbols with the same value) and produces a combination of length-value pairs as non-negative integers [35]. Below is an example:

$$\overbrace{11111}^{5}\overbrace{000000000}^{9}\overbrace{111}^{3}\overbrace{00000}^{5}\overbrace{1111}^{4} \rightarrow (\mathbf{5},1)\,(\mathbf{9},0)\,(\mathbf{3},1)\,(\mathbf{5},0)\,(\mathbf{4},1)$$

Reverting this transformation is easy: given some RLE-transformed data x with x_i the value of the symbol at position i, the original data y is then:

$$y = \overbrace{x_1 x_1 ... x_1}^{x_0}\,\overbrace{x_3 x_3 ... x_3}^{x_2}\,\overbrace{x_5 x_5 ... x_5}^{x_4}\,...$$

RLE performs best on data with long runs, since every run of arbitrary length is transformed into two symbols. Similarly, it performs poorly on data with short runs: if the maximum run length is only one, RLE transforms every symbol into two symbols.

2.3.2 Move-to-Front

The move-to-front (MTF) [10] transformation starts with a predetermined list of possible symbols, called the *MTF list*. When working with only lower-case letters, this could be "abcdefghijklmnopqrstuvwxyz". Next, every symbol in the data is replaced with its position in the MTF list, and that symbol is moved to the front of the sequence (hence, Move-to-Front). The output of MTF is thus a sequence of integers between 0 and the length of the MTF list. An example on the word *bananaaa* is shown in Table 2.1.

Input	MTF list	Output
bananaaa	a**b**cdefghijklmnopqrstuvwxyz	1
b**a**nanaaa	b**a**cdefghijklmnopqrstuvwxyz	1
ba**n**anaaa	abcdefghijklm**n**opqrstuvwxyz	13
ban**a**naaa	n**a**bcdefghijklmopqrstuvwxyz	1
bana**n**aaa	a**n**bcdefghijklmopqrstuvwxyz	1
banan**a**aa	n**a**bcdefghijklmopqrstuvwxyz	1
banana**a**a	**a**nbcdefghijklmopqrstuvwxyz	0
bananaa**a**	**a**nbcdefghijklmopqrstuvwxyz	0

Table 2.1: MTF applied to the word bananaaa with initial MTF-array equal to the alphabet

When performed on data with low entropy, the output will usually have lower values than the input. This can provide significant benefits to certain coding algorithms, like interpolative coding (Section 2.4.4). Another side benefit is that repeated sequences will result in an output with very few different numbers. For example, The word *banbanbanban...* results in 1 1 13 2 2 2 2 2 2 2 2 2 .... Applying e.g. RLE on this output will transform it into six symbols, whereas applying RLE on the original data results in doubling of the symbols.

2.3.3 Lempel-Ziv

LZ-77 and *LZ-78* are two modelling techniques published by Abraham Lempel and Jacob Ziv in 1977 and 1978 [86], [87]. They work by building a dictionary using past data to match future sequences to the past data. Instead of writing down the original data, a reference to the previous sequence can be written. LZ-77 encodes matches by a *length-distance* pair. The length specifies how long the matched sequence is, and the distance specifies the amount of bytes behind the start of the sequence is. The algorithm starts at some position in the data. It fills a *lookahead buffer* of a certain length containing the current future data. It then tries to find a match with a length as large as possible, starting at the *window*: the window is a buffer of a certain length that contains the past data. If a match is found, the length and distance of the match is written down. If not, the *length-distance* pair is replaced by *0-x* with x being the actual byte value. The following example demonstrates the algorithm on the string *bananabanana* with a lookahead buffer size and window size of 5.

input	window	lookahead buffer	result
bananabanana		banan	(0,b)
b**a**nanabanana	b	anana	(0,a)
ba**n**anabanana	ba	nanab	(0,n)
ban**a**nabanana	ban	anaba	(2,1)
banan**a**banana	banan	abana	(1,1)
banana**b**anana	anana	banan	(0, b)
bananab**a**nana	nanab	anana	(3, 1)
bananabana**n**a	abana	na	(2, 3)

Table 2.2: LZ77 applied to the word bananabanana

2.4 Entropy coding

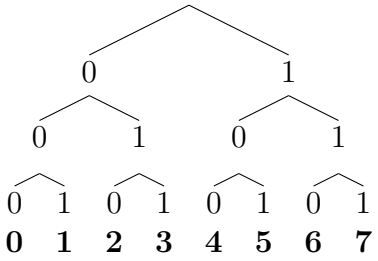
Entropy coding aims to remove the redundancy that some modelling technique(s) revealed. While many different codings exist, the family that is important in this thesis is *integer coding*. As the name suggests, it tries to represent a sequence of integers into a smaller sequence of integers. The reason integer coding is so important, is that the architecture explained in this thesis should be as generic as possible, without knowing what the source of the data is. Hence, the only thing it knows are the byte values, which should be interpreted as integers[3]. Another reason is that most modelling techniques reveal the redundancy using integers. The result of RLE, for example, are Length-Value pairs. [69]

In integer codings, a distinction has to be made between *fixed-length*, *semi-fixed-length* and *variable-length* codings. Fixed-length codes encode all integers into some representation of equal length. Semi-fixed-length does something similar, but has two possible lengths instead of one. Lastly, variable-length codings encode integers into differently sized representations.

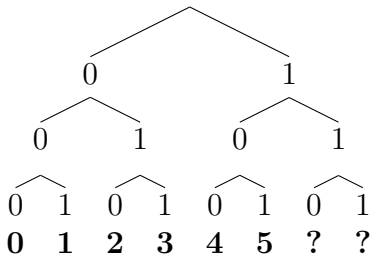
2.4.1 Fixed-length

To be able to encode integers in to a fixed-length encoding, an upper bound on the amount of different values must be known (n). One can then easily encode integers into its binary form, using $\lceil \log_2(n) \rceil$ bits. For example, for $n = 8$, the integers 0..7 are encoded as shown in the tree below (take notice of the leading zeroes): [29] [28]

[3]To give an example, another way to interpret bytes could be as ASCII characters, such that textual coding techniques can be applied. However, the data could come from any source, also non-textual sources. This would make the codings pretty useless, as they are optimized for texts.

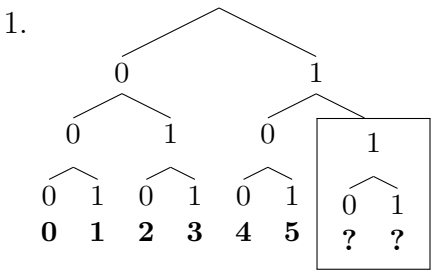


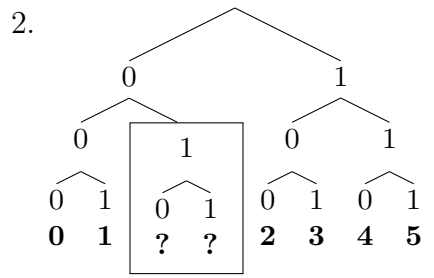
This is not limited to incremental integers starting from 0, as one can map the encodings to different values (e.g. $000 \rightarrow 12$, $001 \rightarrow 14$, $010 \rightarrow 16, ...$). This fixed-length encoding is only efficient if n is a power of two. If the upper bound would be anything else, some bit combinations would be unused. For $n = 6$, the tree below shows that the combinations 110 and 111 are unused.



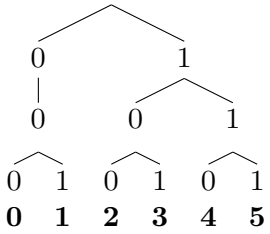
2.4.2 Semi-fixed-length

Semi-fixed-length (or phased-in) codes try to solve the shortcoming of the fixed-length approach if n is not a power of two. It creates codewords of two different lengths: $\lceil \log_2 n \rceil$ and $\lceil \log_2 n \rceil - 1$. The three most widiely known semi-fixed-length codes are: *low-short, center-short*, and *high-short*. These can all be computed using simple arithmetic and bitwise operations, without able look-ups. Low-short will encode the lower integers into the shorter length, center-short does this for the center-valued integers, and high-short does this for the highest integers. The best compression rates are thus achieved by choosing the appropriate version: low-short if primarily low-valued integers are present, and analogous to center-short and high-short. A demonstration of creating a low-short coding is shown, for $n = 6$. [61]

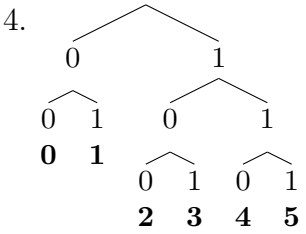




3.



4.



2.4.3 Variable-length

An unusable, naive variable-length coding is simply its *binary code* without leading zeroes (except the number 0 itself), e.g. $\beta(7) = 111, \beta(4) = 100$. However, using this technique, the decoder can not detect the end of the codeword. Imagine the binary code of some sequence of integers $x : \beta(x) = 1101$. The decoder doesn't know how many and which values x contains. In fact, there are five different solutions. Even if the decoder knows x contains for example 2 integers, there are still multiple possibilities: x could be 1 3 or 6 1.

One solution is instead to write the *unary code*, where x amount of zeroes are written, appended with a 1, e.g. $\alpha(7) = 00000001$. A decoder now knows a new integer starts after every 1. A downside is the amount of bits used, especially with big numbers: to represent 255, 256 bits (32 bytes) have to be used, whereas a fixed binary encoding of length 8 represents 255 in a single byte. [29] [28]

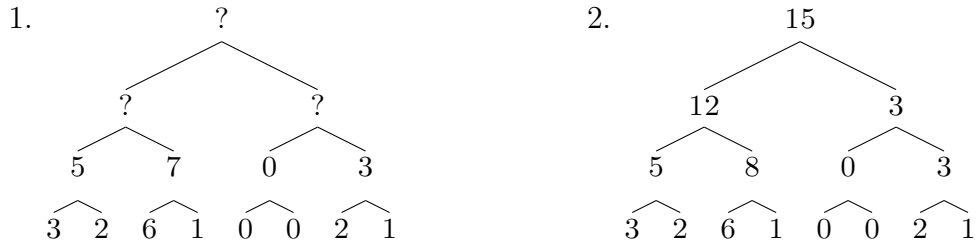
A technique that combines best of both worlds is the γ code, described by Elias [27]. First write x in binary. Then, prepend x zeroes, with x the number of bits written minus one. For example: [27]

$$\gamma(7) = \overbrace{0..0}^{x} \ \beta(7) = \overbrace{0..0}^{x} \ \overbrace{111}^{x=3-1} = 00111$$

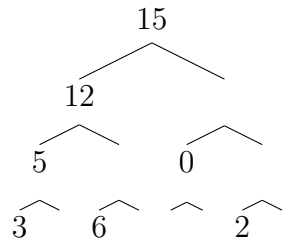
2.4.4 Interpolative coding

Interpolative coding is a technique used to calculate suitable upper bounds to use in semi-fixed-length codings when no tight upper bound is known. It should be noted that the original interpolative coding by Moffat *et al.* [57] was developed for numbers in inverted lists, which were assumed to be in strictly increasing order. Teuhola [78] expanded this coding allowing also non-increasing lists. Here, we will focus on his tree-based version.

The tree-based interpretation starts by building a tree where each parent's value is the sum of the two children's values. This is recursively done until one root value is reached. An example of the buildup tree starting from the sequence 3 2 6 1 0 0 2 1:



The goal now is to encode this tree. It can be optimized by not writing the right children and the children of zero nodes. The right children can be calculated from the left child and the parent, and the children of zero nodes are always zero too. The example tree thus looks as follows



To encode the tree, start at the root and go from left to right, before descending a level in the tree. The root value has to be variable-length encoded, for example, using the γ code (Section 2.4.3), as there is no upper bound. The rest of the values can be low/center/high-short encoded with their parent as upper bound[4]. In general, the following two rules work best: the law of large numbers states that the sum of sufficiently many numbers will tend towards the average. This means that center-short is the most appropriate for internal nodes. Secondly, in leaf nodes lower numbers usually dominate, so low-short is the better encoding there.

Parents	N/A	15	12	3	5	7	3
Value	15	13	5	0	3	6	2
Code	0001111	1101	001	00	011	110	10

The final code is 000111111010010001111010, 24 bits long. In this example, compared to the low-short encoded original sequence (1000111110100000011010, 22 bits), this is a downgrade.

2.5 Cryptography

Cryptography is the study of techniques related to aspects of information security. The terms *confidentiality*, *integrity* and *authentication* are often used, but they can have slightly different definitions depending on the context. We define these terms as follows: [54] [44].

[4]Since 0 is a possible value, the n explained in Section 2.4.2 is the parent's value + 1

1. *Authentication* provides identification between entities. Two entities communicating can identify each other and be sure that they are not talking to an attacker impersonating the entity. In this thesis, we only make a distinction between *authorized* and *unauthorized* entities. Authorized entities should have read and write access to the data, while unauthorized entities should not. Authentication thus ensures that entities can be authorized, but doesn't require different authorized entities to be distinguishable.

2. *Confidentiality* ensures that an unauthorized entity has no means of gathering information about some data. Algorithms providing confidentiality prevent unauthorized entities from learning new information about the plaintext, based on the ciphertext[5].

3. *Integrity* ensures that an authorized entity can verify that the data is *authentic*: the data has not changed since the last authorized entity wrote it. The authenticity of data fails when:

   (a) there is an unintentional modification due to an error in the system.

   (b) an unauthorized entity intentionally modified the data.

   An integrity providing scheme that can only detect (a), provides *passive integrity protection*. An integrity providing scheme that detects both (a) and (b), provides *active integrity protection*. Unless otherwise specified, active integrity protection will simply be referred to as integrity protection.

Since we only make a distinction between authorized and unauthorized entities, *active* integrity protection implicitly also provides authentication. When integrity is ensured, it is not possible for an unauthorized entity to modify data undetectably. Authorized entities thus know that when the data is authentic, the last entity that modified the data was authorized.

These security goals can be achieved using three primitives: *unkeyed*, *symmetric-key* and *public-key* (or asymmetric) primitives [54]. The symmetric-key primitives require all authorized parties to have an exact copy of the key, whereas the public-key primitives require each entity to have their own a private-public key pair. Public-key primitives are not in the scope of this thesis, and are not further discussed.

2.5.1 Encryption and ciphers

Confidentiality is usually achieved by *encrypting* a message, transforming it into an incomprehensible message. The transformation is done by passing the data (called plaintext) through a *cipher*, and the result is called a *ciphertext*. A historical example is the Caesar cipher [7]: transform every letter into the letter three places further in the alphabet. For example, applying the Caesar cipher to CRYPTOGRAPHY results FUBSWRJUDSKB. Evidently, it should be possible to revert the encryption process, so that authorized parties can read the original data. This process is called *decryption*.

[5] Plaintext and ciphertext are described more deatiled in the next section

It is easy to see that ciphertext produced by the Caesar cipher is not well-protected, as anyone who is aware of the cipher is able to decrypt the ciphertext. To prevent unauthorized parties from decrypting ciphertext while being aware of the cipher used, a key is required. As explained in the previous section, the ciphers important in thesis are symmetric: they require the same key to encrypt and decrypt the data.

It is important to know that ciphers can be broken. In fact, all ciphers can be broken by using a brute-force attack[6]: an attack where the attacker tries every possible input until it gets the wanted output [64]. Depending on the context, such a brute-force attack is deemed impossible, as it would simply require too much time and power to perform. Thus, according to Schneier [70], cryptanalists consider a cipher broken when a weakness is found that can be exploited with a complexity less than brute-force.

Symmetric-key ciphers can be further subdivided into *block* and *stream* ciphers. A block cipher takes as input some data of fixed length, called a block (not to be confused with blocks of a drive or filesystem), after which it transforms the entire block. Stream ciphers, on the other hand, do not have a restriction on length of the input data and transform every character or bit individually. A very important block cipher is the *Advanced Encryption Standard (AES)*, established by the National Institute of Standards and Technology (NIST) [25]. It is the most widely used block cipher today and has withstood analysis and attacks for over 20 years. The best known attack against AES is a key-recovery attack[7] which still requires 2^{126} (instead of brute-forcing 2^{128}) operations for 128-bit keys [76]. Most processors also have built in hardware instructions for AES, which prevents side-channel[8] attacks [59]. The detailed specification of the algorithm is not relevant to this thesis and is thus omitted, but the algorithm will often be used as an example.

2.5.2 Block cipher modes of operation

A block cipher on its own is not very useful, as the size of the data that needs encryption hardly ever matches the block size. Block ciphers are thus executed in a certain *mode of operation*. They allow for data to be of arbitrary length and can achieve more secure results. The following sections explain several modes of operations that are important and relevant to this thesis.

2.5.2.1 Electronic Code Book (ECB)

The ECB mode is the simplest and fastest mode to implement. To encrypt, it chops the plaintext in blocks and performs the cipher on each block independently. The

[6] Except data encrypted in an information-theoretically secure manner. It is a theoretical environment in which the system is secure against adversaries with unlimited computing resources and time [52].

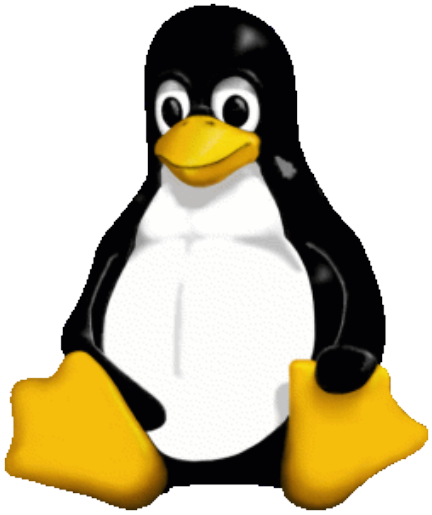
[7] A key-recovery attack is an attack where the adversary tries to recover the key used in the encryption process.

[8] A side-channel attack is an attack that exploits physical information leakages such as timing data, power consumption, or electromagnetic radiation [75].

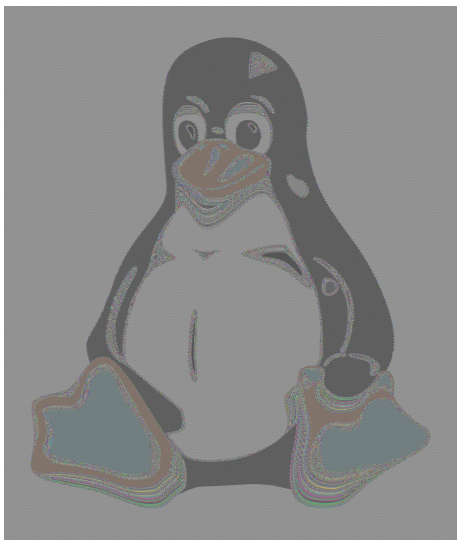
last block is padded[9] to match the block size if necessary. The resulting ciphertext is concatenation of all the ciphertexts of the blocks. Decryption is done by feeding the ciphertext as input, and performing the cipher decryption. The ECB mode using AES as its block cipher is illustrated in Figure 2.3a. [23]

Advantages	Disadvantages
Every block can be encrypted and decrypted independently, so ECB supports parallelism resulting in fast encryption and decryption speeds.	Identical plaintext input blocks with the same key are encrypted into exactly the same ciphertext. Although encrypted, patterns in the plaintext are thus visible in the ciphertext, which is not always suitable.

Figures 2.1a and 2.1b show an example of an image encrypted with AES-ECB. It clearly shows visible patterns, although the image is encrypted.



(a) Original Tux image. https://isc.tamu.edu/~lewing/linux/



(b) AES-ECB encryption of this image, clearly showing visible patterns. https://words.filippo.io/the-ecb-penguin/

Figure 2.1

2.5.2.2 Cipher Block Chaining (CBC)

The CBC mode chains the plaintext blocks with the previous ciphertext blocks. More concretely, the resulting ciphertext of every previous block is XORed with

[9]Padding: Adding bits to the data such that it matches a certain length. Most often padding is done on one side with only 0's or 1's.

the plaintext of the current block, and the initial input block is XORed with an Initialization Vector (IV). A block's ciphertext is thus dependent on all the previous blocks. This solves the problem in ECB where identical plaintext blocks are encrypted into the same ciphertext blocks. Take note that to decrypt, the XOR operation happens *after* the cipher decryption. Choosing a random IV is important, as encrypting identical plaintexts with identical IVs will result in the same ciphertext. The CBC mode using AES as its block cipher is illustrated in Figure 2.3b. [23]

Advantages	Disadvantages
Patterns in the plaintext are not visible anymore, solving the primary problem in ECB mode.	Encryption cannot be parallelised, and is thus slow. This makes it not suitable for disk encryption. Not choosing random IVs can still reveal patterns.

2.5.2.3 Counter (CTR)

The CTR mode utilises a counter R that has an initial value equal to an IV. Instead of encrypting a block of plaintext (like in ECB or CBC), the counter value is encrypted instead. The encrypted counter is then XORed with a block of plaintext, creating a block of ciphertext. The counter value is incremented by one for every block of plaintext: $R_0 = IV$, $R_1 = IV + 1$, ... Observe that decrypting is done exactly the same: *encrypting* the counter and XORing it with the ciphertext[10].

Every block can be encrypted individually, which allows CTR mode to be fully parallelised providing fast encryption and decryption. The CTR mode using AES as its block cipher is illustrated in Figure 2.3c, [23]

Advantages	Disadvantages
Both encryption and decryption are parallelisable, making CTR fast and efficient The same plaintext does not encrypt to the same ciphertext, assuming a different IV was used.	Not choosing random IVs can still reveal patterns

[10]This works because the inverse of XOR is itself: $a \oplus b = c \implies b = c \oplus a$, so $ciphertext = counter_{enc} \oplus plaintext \implies plaintext = counter_{enc} \oplus ciphertext$

2.5.2.4 XEX-based tweaked-codebook mode with ciphertext stealing (XTS)

XTS [51] [24] is a tweakable block cipher. A normal block cipher E has a key K and a message M, and its ciphertext C is

$$C = E_K(M)$$

A tweakable block cipher introduces a tweak T, such that

$$C = E_K(M, T)$$

It must not be confused with an initialization vector, as they have different properties: an IV needs to be random, whereas a tweak doesn't need to be.

XTS uses Rogaway's XEX (Xor Encrypt Xor) method [67]. Rogaway proved that if E is a secure block cipher, E'_K is also a secure block cipher, with:

$$E'_K(N, i_1, ..., i_k, M) = E_K(M \oplus \Delta) \oplus \Delta$$

$$\Delta = \alpha_1^{i_1} ... \alpha_k^{i_k} E_K(N)$$

$$\alpha_1^{i_1} .. \alpha_k^{i_k} \neq 1$$

with $\alpha_1 ... \alpha_k, N \in GF(2^n)$[11] and $i_1 ... i_k$ are integers.

It introduces a tweakable block cipher with $k + 1$ tweaks: $i_1 ... i_k$ and N.

XTS builds on top of XEX, limiting the amount of tweaks to two. It is also possible to use two different keys instead of one. In XTS, the two tweaks are the block number and offset within the block. Since $k = 1$, there is also only one α. This constant is either 2 if the elements are represented as bitstrings, and x if they are represented using polynomials instead. An overview of XTS is given in Figure 2.2, with i and j the tweaks (block number and offset).

Furthermore, XTS supports *ciphertext stealing*, which is needed when the last message is shorter than the block cipher size. Imagine the last message M_m has a length of s, and the block cipher size is n ($s < n$). The second to last block (M_{m-1}) is encrypted creating a temporary value $C = C_m || C'$. C_m is s bits long and becomes the ciphertext for M_m. Lastly, C_{m-1} becomes $E_K(M_m || C')$. The last block basically 'steals' a part of the second to last block's ciphertext.

XTS, especially in combination with AES, is a popular algorithm for Full Disk Encryption (see Section 3.2.1). The first reason for its disk encryption popularity is the length-preserving nature. The input to the encryption scheme should only include the data, key, block number and offset, which is what XTS does. Secondly, XTS protects better than other AES modes against reorder attacks (because of the tweak) and ciphertext manipulation. [51]

[11] The Galois Field $GF(2^n)$ is a finite field with 2^n elements. For example, a 4 bit number is an element of $GF(2^4)$ because it can have exactly 16 different values.

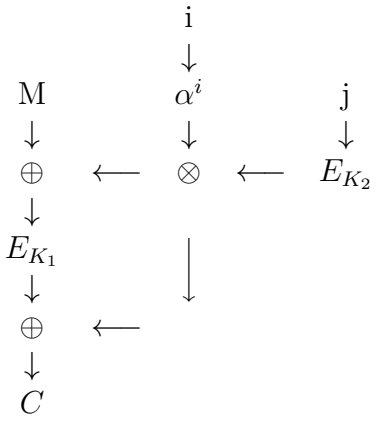


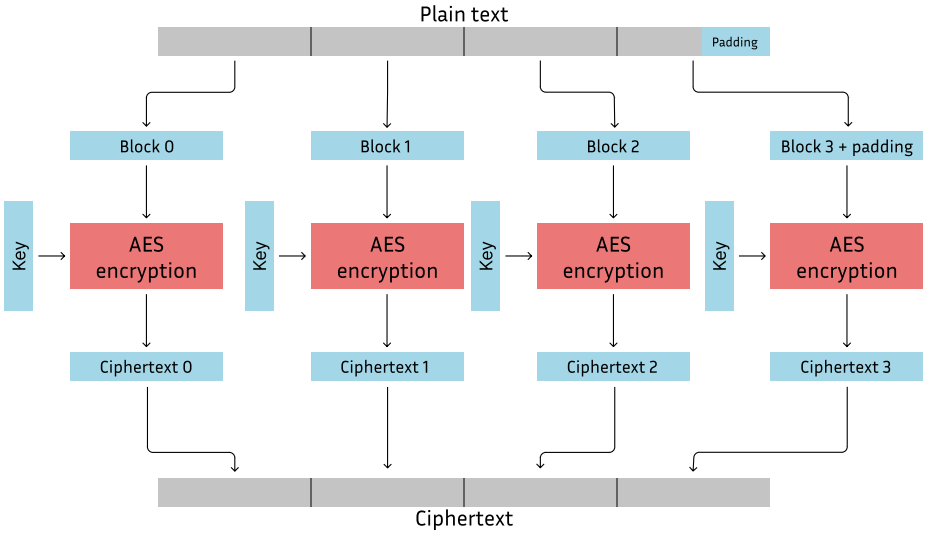
Figure 2.2: An overview of XTS with M the message, K_1 and K_2 keys, C the ciphertext, E the cipher, $\otimes$ a multiplication and $\oplus$ a XOR operation.

2.5.2.5 Galois/Counter mode (GCM)

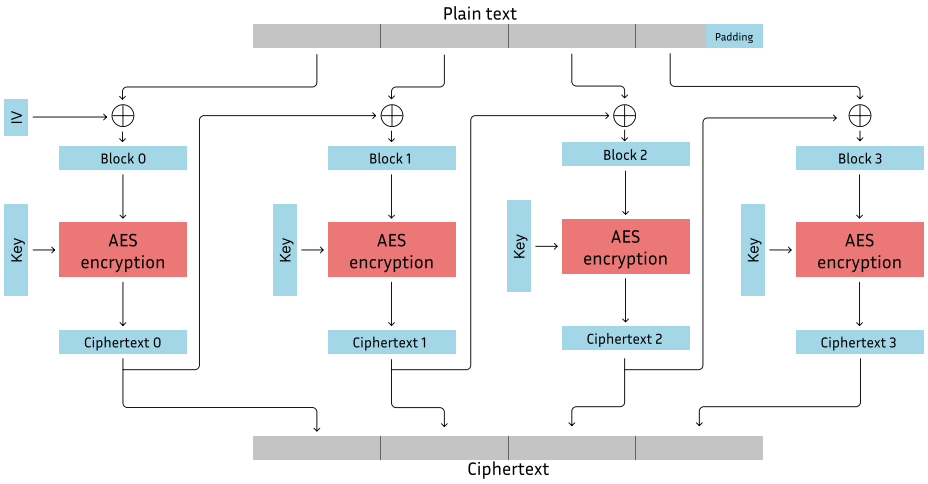
Galois/Counter [53] Mode was invented to improve performance of ciphers that also require authentication. Hardware accelerated ciphers increased the performance of other modes, like the normal Counter mode. While the encryption ciphers increased in performance, there was no standard message authentication algorithm that can keep up with the cipher. GCM solves this by computing an authentication tag using binary field multiplication, which can be implemented at a far lower cost than the counter mode.

Its inputs are similar to the Counter mode: an IV, plaintext and key. Additionally, it can accept Additional Authenticated Data (AAD). This additional data can be data that needs to be authenticated, but not encrypted. Its outputs are the ciphertext and an authentication tag.

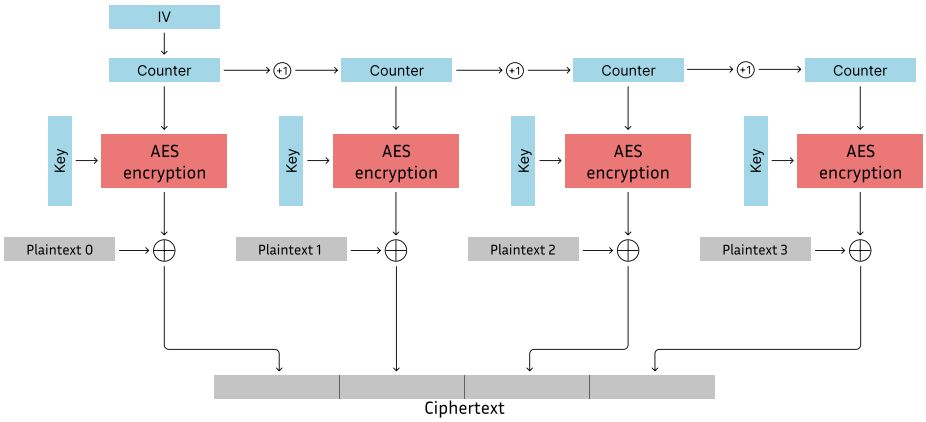
GCM is illustrated in Figure 2.3d. The start of GCM encryption is similar to normal CTR mode, initializing the counter to the IV, incrementing the counter for every block and encrypting this counter. Then, the encrypted counter is XORed with the plaintext to get the ciphertext. A small difference with the CTR mode is the first encrypted counter value that is not used for the plaintext, but kept to calculate the resulting authentication tag. After the ciphertext is calculated, the AAD is multiplied in the Galois Field $GF(2^{128})$ (refer to [53] for specifics) and XORed with the first ciphertext block. This result is then repeatedly multiplied in $GF(2^{128})$ and XORed with the next ciphertext block. The last block is instead XORed with the concatenation of the lengths of the AAD and Ciphertext, multiplied again in $GF(2^{128})$ and XORed with the first encrypted counter value. This result is the authentication tag.



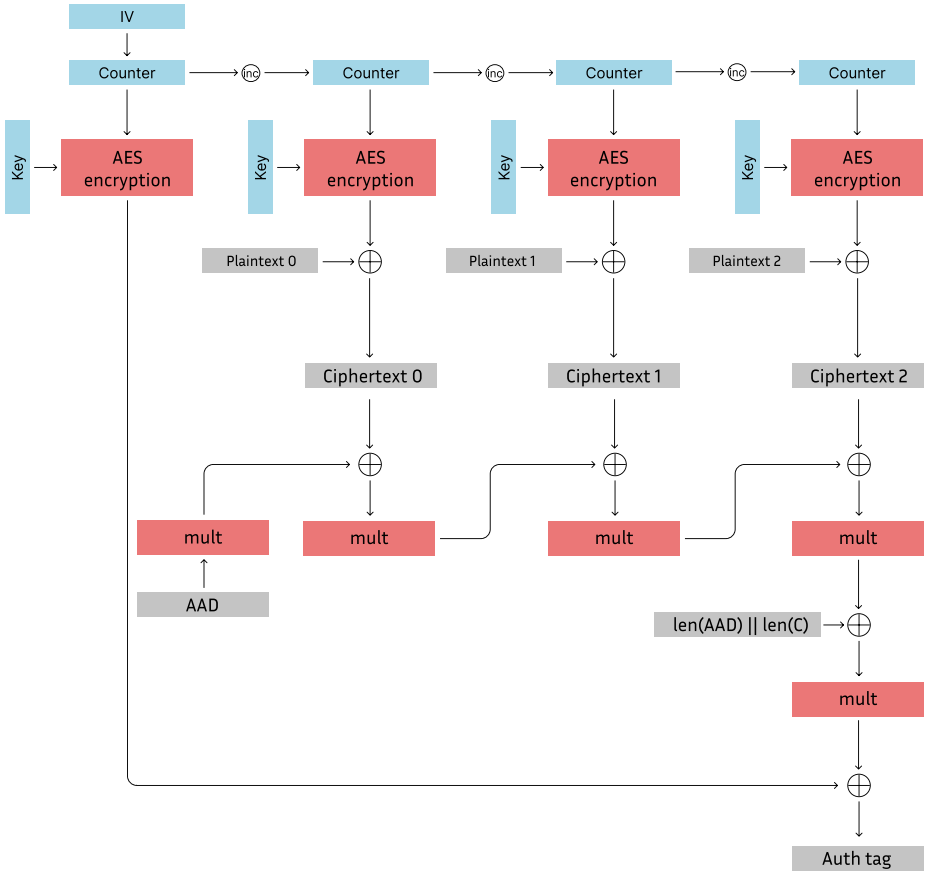
(a) Overview of encryption in AES-ECB.
For decryption, swap plaintext with ciphertext and AES encryption with AES decryption.



(b) Overview of encryption with AES-CBC.
For decryption, swap AES encryption with AES decryption and reverse the vertical arrows.



(c) Overview of encryption with AES-CTR.
For decryption, swap plaintext with ciphertext, but **keep AES encryption**.



(d) Overview of encryption with AES-GCM.
For decryption, swap the direction of plaintext and ciphertext arrows into the XOR operation.

Figure 2.3

2.5.3 One-way function

A *one-way function* (OWF) is a function that is *hard* to invert, i.e. finding the preimage x given the output $f(x)$ should be *infeasible*. To be more precise about *hard* and *infeasible*, we define a one-way function f with a security experiment shown in Figure 2.4. Here, $\mathcal{A}$ is the adversary and λ is the input length. The definition of an OWF states that the probability of this experiment returning 1 is negligible for any PPT adversary[12] [14].

Simplified, the definition states that for a one-way function f and random string x, no PPT adversary given $f(x)$ can find the original value x with a non-negligible probability.

$\underline{Exp^{OW}_{f,\mathcal{A}}(1^\lambda)}$

// sample a random binary
// string of length λ
$x \leftarrow_\$ \{0,1\}^\lambda$
$y \leftarrow f(x)$
$x' \leftarrow_\$ \mathcal{A}(1^n, y)$
if $|x'| \neq \lambda$:
  return 0
if $f(x') = y$:
  return 1
return 0

Figure 2.4: One-way function experiment

2.5.4 Cryptographic hash Function (CHF)

A hash function is a function that deterministically maps input of arbitrary size to a fixed-size output. A cryptographic hash function is hash function that has additional security properties [3]:

- It is quick[13] to compute the hash for any input.
- It is a one-way function.
- It is collision resistant, i.e. no PPT adversary exists that can find two messages M and M' with $M \neq M'$ such that they map to the same value.
- A small change in input changes the output such that it appears uncorrelated with the old hash value.

[12] A PPT adversary is an adversary that can do a task in PPT.
[13] In a cryptography context, quick means with a time complexity of $\mathcal{O}(n)$ or less.

A popular family of CHFs is the Secure Hash Algorithms (SHA) family. An example with the hash function $sha1$:

$$sha1(\text{"}abc\text{"}) = a9993e364706816aba3e25717850c26c9cd0d89d$$

CHFs are mainly used to compute a digest. Such a digest can then be used for numerous applications: integrity verification, digital signatures, password verification, etc.

Since CHFs are deterministic, they all share a common vulnerability: an attacker could precompute a table of a large amount of hashes of common data. This is called a *rainbow table*. The attacker can then try to find a match between a new hash and a hash in the table. If a match is found, the attacker knows with very high probability what the original message was.

To defend against an attack involving a rainbow table, a *salt* is added during the hashing. A salt is a random string of data, which is made public. The new hash then becomes $hash(data + salt)$. If an attacker now wants to use rainbow tables to reverse the hashes, a new table for every single salt should be made. If the salt is large enough, let's say 128 bits, 2^{128} different rainbow tables have to be constructed. Even if the creation of a reasonably sized rainbow table takes only 1 second[14], it would still take $1.08 \cdot 10^{31}$ years in total. For comparison, the universe exists approximately $10 \cdot 10^{10}$ years ([70] Section 1.7) . The time is not the only aspect that limits these brute force approaches, thermodynamics also restricts what can be done: iterating a 219-bit counter through all its states takes the same amount of energy as a typical supernova explosion. Hence, to try and construct 2^{256} rainbow tables, one would need to consume more energy than a supernova explosion, just to iterate through all the different salts (without actually computing the rainbow tables) ([70] Section 7.1).

However, if the attacker is only interested in reversing one specific hash, only a single rainbow table needs to be made. With the right infrastructure and with data that is not too long and random (e.g. a password) this could be done relatively fast. Thus, for relatively short data with low entropy, more protection is needed; e.g. rate limiting or CAPTCHAs. Rate limiting slows down the amount of incorrect authentications a user can make. For example, after three incorrect tries, the user needs to wait five minutes. This prevents attackers from brute forcing common (combinations of) passwords, as it would take too long. Alternatively or additionally, a CAPTCHA can be used. CAPTCHA, originally introduced by Ahn *et al.* [1], stands for: Completely Automated Public Turing test to tell Computers and Humans Apart. It is a test that humans can pass, but automated programs can't. They are often required when creating new accounts, like an e-mail account. In our password example, it would prevent a bot from trying a lot of possible passwords, instead requiring a human to intervene, which is considerably slower.

[14]This is an enormous underestimation used for demonstration purposes. The true time it depends on the amount of entries one wants to achieve, and the infrastructure used to construct the tables.

2.5.5 Message Authentication Code (MAC)

A Message Authentication Code (MAC), often just called *tag*, is added to messages to allow detecting whether the message has been tampered with. Using a secret key, the MAC can be computed to tag messages and verify messages using the same key. MAC algorithms need to be Unforgeable Under Chosen Message Attacks (UNF-CMA), meaning attackers cannot forge message-tag pairs, even if it has control over the messages that are being tagged. [14]

More concrete, MACs are computed over some data: files, network transmissions, programs, etc. This MAC is stored somewhere, or sent along with the data. The MAC can be recomputed at a later stage, and if the stored / sent MAC is not equal to the freshly computed MAC, it provides evidence that the data may have been tampered with. Because of the UNF-CMA property, an attacker without access to the key can not modify the data and afterwards compute the correct MAC for this new data.

In combination with encryption, the order of tagging and encrypting is important, and gives different results. The three methods include: *Encrypt-and-MAC*, *MAC-then-Encrypt* and *Encrypt-then-MAC* [8]. Encrypt-and-MAC consists of encrypting the plaintext, and computing the MAC over the plaintext as well. According to Bellare *et al.* [8], Encrypt-and-MAC is not secure. An easy example is that a MAC can leak information about the input [14]. Because neither the input, nor the MAC is encrypted, it could thus reveal information about the plaintext. In contrast, MAC-then-Encrypt first computes the MAC over the plaintext, and encrypts both the plaintext and MAC. Lastly, Encrypt-then-MAC first encrypts the plaintext, and then computes the MAC over the ciphertext. The two latter modes are similar even though Encrypt-then-MAC is generally recommended [8]. This is mainly because it is easier to theoretically prove its security properties. For example, MAC-then-Encrypt does not provide any integrity protection over the ciphertext. This vulnerability makes the MAC-then-Encrypt scheme malleable [15] if the used cipher is also malleable. However, in practice this should not happen, as a malleable cipher is already dangerous to use and should thus be avoided under any circumstance.

2.5.6 Merkle tree

In 1988, Merkle introduced the concept of what now is known as a Merkle tree. A Merkle tree is a tree in which every *leaf* is a hash of some *data*. Every *inner node* is a hash over the node's *children*. An example of a Merkle tree is given in Figure 2.5. [55]

In general, verifying if some data is still valid and hasn't been unintentionally corrupted can be done by storing the hash of each file and comparing the stored hash to a newly computed hash of the file. However, this doesn't prevent the corruption

[15] Malleability: the ability of an attacker to change some ciphertext which decrypts to a similar ciphertext, hence being able to modify a part of the plaintext [22].

by an attacker, as the attacker can replace the data, compute the hash of the malicious data, and store this hash. This is where *Merkle trees* are useful. Protecting data in an untrusted storage space simply requires to either store the root hash in a trusted environment, or digitally sign the root hash. Then, when trying to validate a file, he needs to request only a small amount of hashes from the tree to verify the authenticity of the file. Because Merkle trees are expensive to construct, they are mainly used in read only scenarios, such that the tree does not need to be rebuilt after every modification.

An example is shown in Figure 2.5. Assume someone with a stored root hash wants to validate data D1. All he needs to do is request H2 and H1 from the tree. He calculates H3 of the data he wants to validate; together with H2 he can calculate H0; together with H1 he can then calculate the root hash. If the root hashes are equal, the file was valid. If not, there was some error along the way.

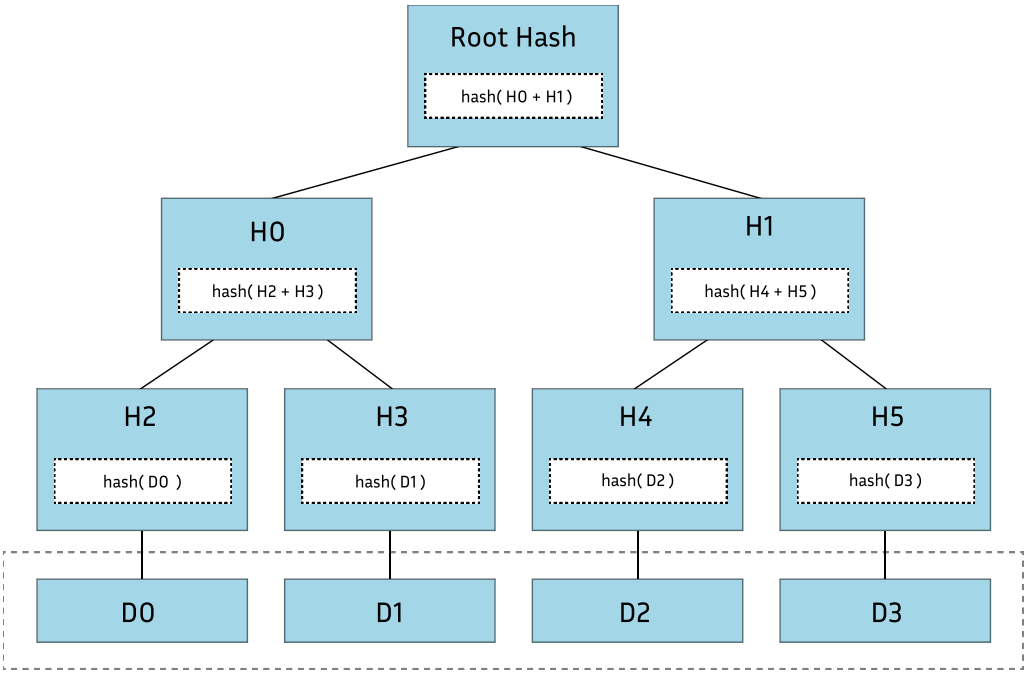


Figure 2.5: A Merkle tree example. The bottom blocks D0-D4 contain the actual data. H2-H5 contain the hashes of respectively D0-D4. H0 and H1 contain the hash of respectively H2 concatenated with H3, and H4 concatenated with H5. Lastly, the root hash is a hash over the concatenation of H0 and H1.

2.6 Storage devices

Storage devices are physical devices that store and retrieve data. They can be split up in two categories: *nonvolatile* and *volatile* devices. Nonvolatile devices retain their data when electrical power is lost, whereas volatile devices lose their data. In most computers, the volatile devices are called the memory or RAM (Random

Access Memory), and the nonvolatile devices are called disks or drives. The two most important types of disks are analyzed further.

2.6.1 Hard disk drive

A hard disk drive (HDD) [2] is one of the oldest types of disks. It's a mechanical disk which stores and retrieves data using a rotating platter and a moving platter head, much like a vinyl record. The head can move in one line over the surface of the platter, and together with the rotation of the platter itself, it can access any location on the platter. The head can make magnetic changes to the platter, which can be retrieved at any time.

HDDs are divided into sectors, meaning that data can only be read and written in multiples of the size of a sector (Figure 2.6). One of the advantages that makes HDD still relevant nowadays is the fact that the price per storage is very low. For example, in 2017, Seagate sold 8TB HDDs for 0.038$/GB [48]. Nevertheless, the HDD's technology is past its prime. One of the major disadvantages is its slow speed. The main factor in latency in HDDs is caused by the head that needs to wait until the platter has spun the correct distance before it can read the data.

Many optimizations have been made like prioritizing certain reads over others depending on the current state of the disk. There is also a significant latency difference between random access data and sequential data. Sequential data can be written or read without much extra latency in between the blocks, since the head will pass over this data as it's requested. However, for random data the head will have to stall very often, waiting for the platter to be in the right spot.

Even though HDD's are slowly being replaced by SSDs (Figure 2.7), the concept of blocks as a unit remains, and it is also used in newer generations.

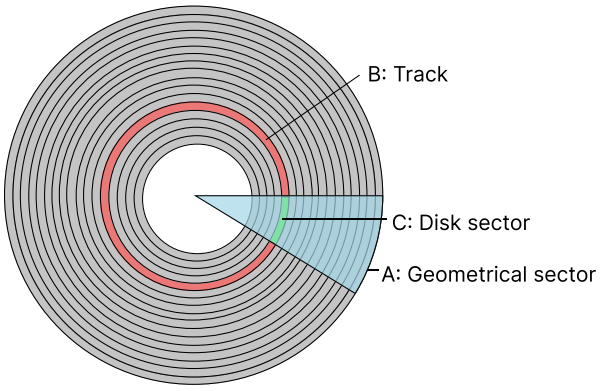


Figure 2.6: Blue: Geometrical sector, Red: Track, Green: (Disk) Sector

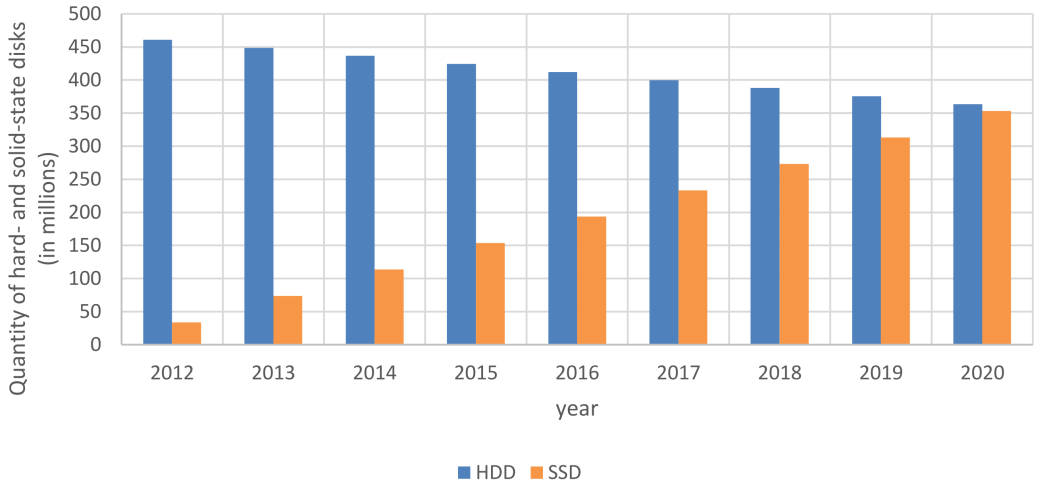


Figure 2.7: Estimation of global SSD and HDD shipments until 2017. Years 2018 until 2020 are predictions [77].

2.6.2 Solid State Drive

A Solid State Drive (SSD) [66] is a more recent type of nonvolatile storage device, that grew a lot in popularity in the past decade. In contrast to HDDs, SSDs do not have physically spinning platters or other mechanical features. Instead, SSDs store data in semiconductor cells. Advantages of SSDs consist of higher shock resistance, lower latency, less noise, and smaller dimensions. The lower latency includes both sequential and random access data.

Even though SSDs do not have physically divided sectors, the smallest possible structure to read or write is also called a sector or a block. This size varies, but most newer disks have sector sizes of 4096 bytes.

2.6.3 Random Access Memory (RAM)

A running processor needs space to store and retrieve data. Linking the processor directly with an HDD or SSD would be very slow. Instead, computers have a memory hierarchy, stacking different storage devices. The devices closer to the processor are faster but have a smaller size. One of these layers is the Random Access Memory (RAM). Data used by processes is stored on the RAM and allows for faster data loads and stores compared to drives. RAM can be both volatile and nonvolatile. While nonvolatile RAM is cheaper per GB, as well as having more capacity, it is slower and less reliable. [41] [49]

To understand the speed difference between RAM and drives, the latency[16] of DDR3 (a type of RAM with production year 2010) is compared with an HDD produced in the same year (Seagate ST3600057). DDR3 acquired an average latency

[16]Latency: The time to complete a simple operation assuming no contention [41] p20.

of $37ns (1ns = 10^{-9}s)$, whereas the average latency of the HDD reached $3.6ms$ $(1ms = 10^{-3}s)$. These statistics do not present a perfect example of the speed differences between the two, but they do show a rough example of the speed differences between RAMs and HDDs. [41].

2.6.4 Virtual memory

Originally, programmers needed to know the amount of RAM available, and take it into consideration when the program's size would exceed the available memory. This was feasible for a little while, because programmers knew the specifications of the machine they were working on. However, the complexity of programs grew, and taking the RAM into account was not always possible anymore. [20]

This led to the introduction of *Virtual Memory*. Virtual Memory has the purpose of giving the programmer the impression as if the RAM is far bigger; it is no longer the programmer's task to account for running out of memory. Both virtual and physical memory are divided into fixed-sized units: *pages*. A Memory Management Unit (MMU) maps virtual memory addresses to physical memory addresses. Programs access their data using the virtual memory, which the MMU translates to the physical addresses. The RAM then can retrieve or store the requested data using the physical address. The virtual memory can be made far bigger than the actual available memory, which relieves the programmer from caring about the available memory. The MMU makes this possible by dynamically *swapping* out pages for new ones. A page that is swapped out is sent to the disk, because the MMU expects it won't be used soon. The newly freed memory is then assigned to the requesting process. When a process wants to access some memory that has been swapped out to the disk, a *page fault* occurs. Page faults require the MMU to swap the page from the disk back in the memory, by swapping another page out. [45]

A simplified paging example can be as follows:

1. A process requests memory
2. There is memory free and the MMU maps a page to assign it to the process
3. A process requests memory
4. There is no memory left. The MMU decides which page to swap to the disk which frees some memory. It now maps a new page to the process
5. A process wants to access a page it owns, but the page was just swapped out
6. A *page fault* occurs, the MMU swaps the page back in by swapping another page out
7. A process signals it does not need a page anymore
8. The MMU frees the page

How MMUs can manage memory the most efficient is an entire study on its own,

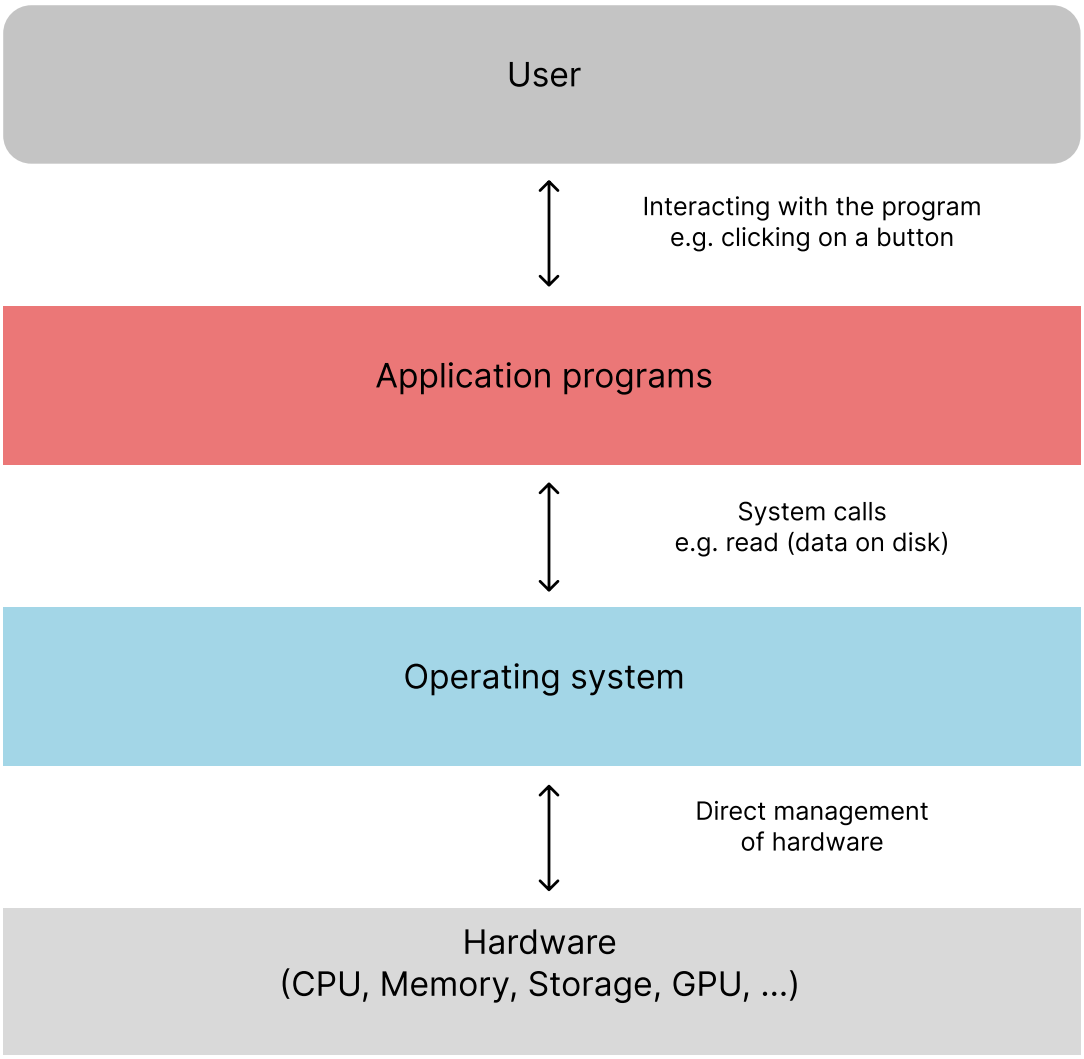


Figure 2.8: A simplified overview of the main stakeholders in a computer

and is not relevant in this thesis. The key takeaway from this section is that memory is also managed in block-like structures, because it facilitates the organization of the virtual memory.

2.7 Operating systems and Linux

A computer consists of many different hardware components: processors, disks, memory, motherboards, GPU's, network devices, etc. These parts have to somehow cooperate, which is accomplished by a computer's *Operating System* (OS). The operating system is the layer between application programs (like web browsers, games and mail apps) and the hardware. It manages the resources and assigns these resources to various applications running in the OS. Figure 2.8 shows how an OS is the bridge between application / user and hardware. [73] [5]

Operating systems can vary greatly in responsibilities, depending on the computer it should be used on. For the everyday user, the most well-known operating systems are Windows, MacOS, Linux, Android and iOS. Linux is not really one operating system, but a family of operating systems (called distributions or distros) which are all based on the Linux kernel.

The Linux kernel is an open source project[17], originally released by Linus Torvalds in 1991. It is the kernel for a vast amount of Operating Systems, for example Android and Ubuntu. Even though the everyday user might not see it, Linux-based operating systems are dominant on the market. Here are some statistics to prove it [32], [56]:

- 100% of the world's top 500 supercomputers run on Linux in 2021.
- 96.3% of the world's top 1 million servers run on Linux.
- 90% of all cloud infrastructure operates on Linux.
- 71.7% of the mobile phones worldwide use Android (based on Linux) in March 2022

Because of Linux' open source license, its wide usage both on client and server side, and the fact that the Huawei operating systems HarmonyOS and EMUI are based on it, it is chosen as the OS to work with in this thesis. Therefore, the next sections mainly focus on Linux.

2.8 Filesystem

A *filesystem* or *file system* defines a data structure and operations on this data which allows the operating system to manage how files are named, stored and retrieved from a storage device. Without it, data wouldn't be isolated and it would not be possible to tell where individual files start or end. A vast amount of types of filesystems exist, but they all share a few common purposes: [5]

- Space management
- Providing a structure (hierarchy)
- Bookkeeping of individual or a group of files
- Access control
- Maintaining integrity
- Providing utilities (like moving or deleting files)

Explaining everything about filesystems is far beyond the scope of this thesis. Hence, we will mainly examine a few specific parts about them: the basic structure of a filesystem, the bookkeeping and maintaining integrity.

[17]Kernel: The core of an operating system; it has complete control over the system and is always in memory.

The main goal of a filesystem is to organise the data on a device such that individual files can be stored and retrieved. It does this by creating directories (sometimes known as folders) in which one can either store files, or further directories, called sub-directories. This process can repeat itself to create a tree of directories and files. Linux goes further than this and implements a Virtual File System (VFS), which also unifies the physical drives into a single directory structure. Thus, in Linux, there is a root directory (named /) under which all the other files and directories reside, regardless on which physical device they are stored. Another important task is the bookkeeping of all the files on the system. Filesystems keep track of the file name, timestamps, location on the filesystem, actual physical location, size, ownership, access permissions, etc. This data and has to be stored and managed too. [5]

Early filesystems like Ext2 [15] supported these features, however, they had a major flaw. In case of power loss or other crashes, the filesystem was prone to corruption, because it can be left in an *inconsistent* state: an exceptional state in which (parts of) the filesystem is (are) not usable anymore. The consequences can range from a file not being saved, to the entire filesystem being unusable. Filesystems developed in the late 90's like Ext3 [81] and NTFS [74] [68] use *journaling* to fix this problem. *Journaling* is the procedure of first writing changes to the filesystem in a *journal*, which is located in a reserved space on the drive. If journaling is applied, the following 'bad' scenarios are possible: [65]

- The system crashes before the change is written to the journal: The change is lost, since the wanted change was stored only in a volatile device (the memory).
- The system crashes after writing to the journal, but before committing the change to the files itself: The rebooted system sees an inconsistency between the journal and actual data, and applies the change as specified in the journal.
- The system crashes while writing to the files: The rebooted system again sees an inconsistency between journal and actual data, and can apply the change again as specified in the journal.

Journaling provides an excellent solution, but also decreases write performance, as writes have to be written twice. The actual decrease in write performance depends on the type, which could be logical or physical, and implementation of the journal. In short, a physical journal keeps track of what the result should look like (thus actually storing the intended data), whereas a logical journal keeps track of the operations performed (e.g. "delete file X"). In general, a physical journal allows for more fault tolerance, but with lower write performance compared to a logical one.

2.9 Drivers, Device files and Block devices

All devices connecting to a computer must be somehow managed and understood by the OS. This is done using *device drivers*. A driver allows the OS (or other software) to use a hardware device without actually knowing the details about the device. In general, drivers allow the OS to use a predefined set of operations, and

the drivers translate it to the specifics for each device. Drivers are often made by the manufacturers of the actual device, but they can also be made by third parties. In short, using drivers forwards the responsibility of 'making a device work with an OS' to the manufacturers of the device (or others), such that the OS is less complex[18]. The example in Figures 2.9 and 2.10 show the difference between having and not having drivers. [21]

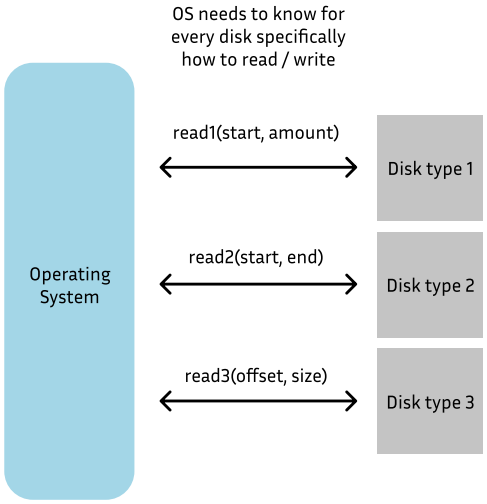


Figure 2.9: An example where an OS has no drivers

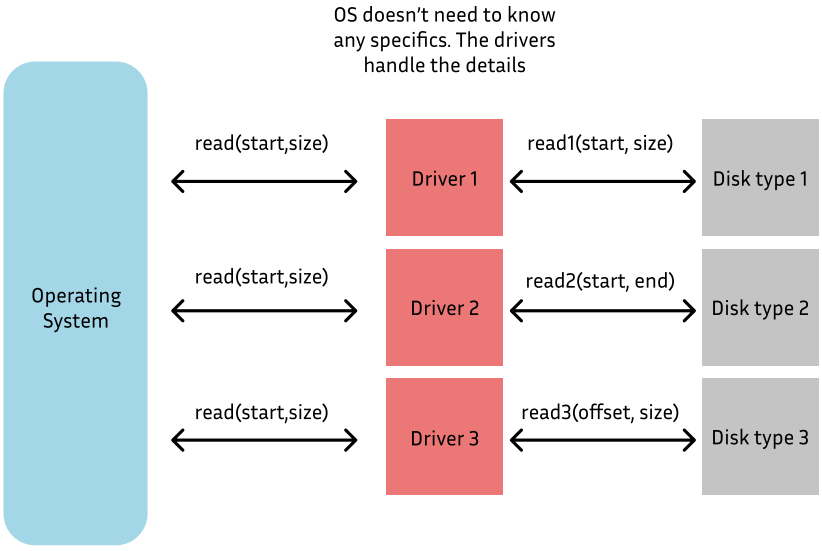


Figure 2.10: An example where the OS uses the drivers for each device

In Linux, a driver is accessed using a *device file* or *special file*. At first glance, these

[18]In the present world, not using drivers is not even feasible. Imagine that Windows had to perform an update for every new device that comes on the market. Not only would this be very expensive, it would also make Windows itself massive, containing a lot of functionality most people wouldn't even use.

files look as if they are ordinary files. However, they are used as an abstraction such that applications can use normal I/O system calls, such as *open*, *read* or *write* to interact with the driver. This simplifies many applications and leads to consistency throughout I/O mechanisms.

Two types of device files exist: *block device (files)* and *character device (files)*. Character devices are devices with which the driver communicates by sending single characters or a stream of single characters only. They usually do not store data, but instead act upon the input immediately. Devices that (indirectly) use character devices are keyboards, monitors, mice, printers, etc. These devices are not useful in the scope of this thesis, so they will be omitted. Block devices on the other hand, can be read from or written to in blocks: the smallest addressable unit of the block device, with typical values of 512, 1024 or 4096 bytes[19]. These devices usually store data. Both HDDs and SSDs are block devices. [83]

2.10 Device-mapper (dm)

The *device-mapper* is an infrastructure in Linux which provides a generic way of creating block devices on top of other block devices. In essence, it can create a virtual block device, which upon request can read, modify, or kill the request, before forwarding it to the block device layer underneath. The device-mapper allows the stacking of dm *targets*, which each implements a distinct function. [50] (appendix A)

Examples of dm targets are *dm-zero* [80] and *dm-linear* [79]. dm-zero will return only zeros for every read request, and silently drop every write request. dm-linear maps a continuous range of blocks onto a different range of blocks. A more complex target is *crypt* (Section 3.2). It transparently encrypts incoming data, and decrypts outgoing data. Because the device-mapper allows for stacking of devices, one can implement a block device as seen in Figure 2.11. The highest layer is a linear target that maps all blocks 1000 blocks further. It then forwards the mapped request one layer lower to a crypt target that will encrypt the data. Lastly, the crypt target forwards this request to another block device. This block device could be the last layer before accessing the physical drive, or it could be another layer. The fact that the device-mapper is layer agnostic, makes the device-mapper layer a very elegant architecture to handle complex transformations over data.

[19] The block size of the block device can be different from the block / sector size from the underlying physical device.

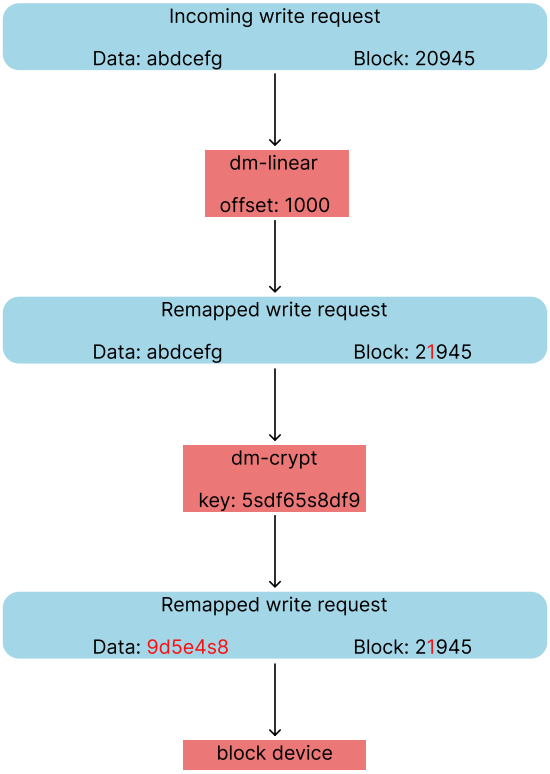


Figure 2.11: A simplified example of device mapper stacking

3 Storage protection

Before analysing previous storage protection solutions, we define storage units and protection more clearly. A *storage unit* is a device that stores data in fixed-sized units. These fixed-sized units are called *blocks* for drives, and *pages* for RAM.

Summarizing section 2.5, confidentiality prevents attackers from extracting information from the data, while authenticity assures the reader that the data was written by an authorized entity. Integrity allows the reader to detect unintentional (passive) or unauthorized modifications (active) on the data. In this thesis, active integrity protection implicitly ensures authentication.

The *protection* that is desired in these devices is *authentication*, *confidentiality* and *integrity*, as explained in Section 2.5.

The protection of storage devices must be efficient, because these devices are read from and written to frequently. These storage devices are often already a computational bottleneck compared to CPU speeds, and long latencies would decrease efficiency drastically. The protection schemes should not fail, even when an attacker can get complete access over it, as well as modify data in it.

This section first introduces file-level protection, and argues why it is not always suitable. Then, block-level protection is introduced, and its existing protection solutions are analysed. Next, page-level protection is explained, along with two existing solutions. Finally, previous research on length-preserving storage protection is described.

3.1 File-level protection

Nonvolatile storage devices usually have a filesystem stored on them. The filesystem is situated in a higher layer than the block layer, and is the layer most end users will interact with. Instead of trying to protect the device on a block level, it is also possible to protect it on a file level. In the end, what users care about is that their files stay safe, regardless of how this is done. In Linux, popular file-level protection methods are *Cryptfs* and *eCryptfs*. Android 7.0 also introduced what is called *file-based encryption* (FBE) [36].

3.1.1 eCryptfs

eCryptfs [39] [26], being the successor of Cryptfs [85], is a filesystem for Linux that can be mounted on top of any directory to protect it. It stores cryptographic data in the headers of the files, protecting every file individually. In short, eCryptfs has one main key: the File Encryption Key Encryption Key (FEKEK). This file is derived from a user's passphrase. For every file that is encrypted, it generates a random File Encryption Key (FEK), which is encrypted with the FEKEK. This Encrypted File Encryption Key (EFEK) is then stored in the metadata of the file. The FEK is used to encrypt and decrypt the file, but to get the original FEK, the FEKEK needs to be used to decrypt the EFEK. Figure 3.1 [40] shows an overview of eCryptfs.

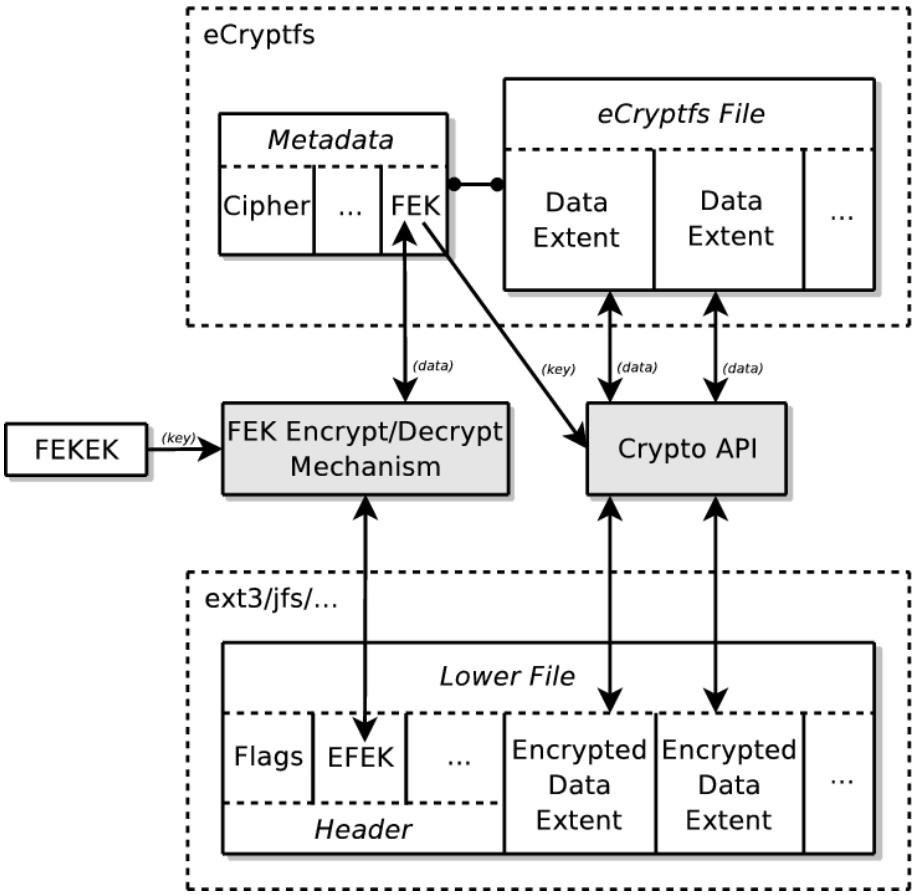


Figure 3.1: Overview of eCryptfs
Source: https://www.linuxjournal.com/article/9400
The bottom square is the normal filesystem, containing encrypted files and metadata including the EFEK. Using the eCryptfs mechanisms and Linux Crypto API, it decrypts the EFEK, and in turn the encrypted data.

3.1.2 Benefits and drawbacks of file-level protection

File-level protection is useful in the case that individual files or directories need to be protected. Furthermore, filesystems manage how data is structured and stored

(Section2.8). Thus, file-level protection solutions can allocate extra storage per file that needs to be protected, without changing the entire structure. Nevertheless, there are five drawbacks:

1. Since files are protected as one unit, reading a fragment of the file requires the entire file to be read to decrypt and verify integrity.
2. They don't work without a filesystem. This is common in RAM or swap partitions.
3. The user needs to explicitly specify which files or directories need protection. If the user wants all the data to be protected, this can be cumbersome.
4. They create extra data and are thus not length-preserving. In case storage is limited, this can be an issue.
5. The metadata of the file is not always encrypted, which can lead to data leaks [36].

Summarized, file-level protection can be inconvenient, since it is not transparent to the user. This led to the IEEE Security in Storage Working Group (SISWG) [42] announcing a call for algorithms providing block-level encryption.

3.2 Block-level protection

The SISWG [42] call for algorithms required the algorithms to be length-preserving. This led to the design of, for example, the XTS mode of operation. However, this is simply an algorithm to encrypt blocks, and is not a full block-level implementation. The next sections cover FDE, a technique to encrypt the entire disk, as well as Linux implementations of FDE and other protection schemes.

3.2.1 Full Disk Encryption (FDE)

FDE refers to the encryption of every block or sector of a disk, providing block-level confidentiality. Having block-level confidentiality implies that the complete data on the disk is confidential.

However, being restricted to the block level also has its drawbacks [47]:

1. There is no space to store additional data, such as a MAC or IV, since the entire disk needs to be encrypted.
2. Adding data integrity is slow: generic solutions require at least double the amount of read and write operations, while also reducing the available disk space [30].

In theory, simply adding FDE to a disk does not allow for strong integrity protection. Often, the notion of integrity is the *poor man's authentication*: if an adversary modifies the ciphertext, the decrypted plaintext will likely be completely different

such that the user may detect the tampering. To get stronger integrity protection, tags need to be stored on a per-block basis. Benadjila *et al.* [9] call this the *Authenticated Disk Encryption* (ADE) model. It protects against attackers aiming to break confidentiality or to successfully modify a block. In their description, ADE does not protect against replay attacks. However, Section [9].

The following sections describe implementations of FDE, as well as implementations to provide integrity.

3.2.2 Device-mapper protection targets

In Linux, block-level data protection is usually implemented in the device-mapper (Section 2.10) layer. An extra virtual block device is put in between the filesystem and the original block device. Read and Write requests are modified such that confidentiality or integrity can be achieved.

In 2012, *dm-verity* [11] was implemented in the Linux kernel. It provides active integrity protection of a **read-only** partition. To achieve this, a **Merkle tree** (Section 2.5.6) is built in which the lowest level consists of a hash of every block. Upon a read request, the hash of the block is verified using the Merkle tree. If a hash is not correct while traversing the tree, the block does not contain the original content. The reason it only supports read-only partitions is because building a new Merkle tree is an expensive operation (although implementations exist that support read-write partitions, see later). dm-verity is used in all Android devices running Android 4.4 or later [43] to protect the boot partition. Here, the root hash of the Merkle tree is used together with other metadata to create a mapping table. This table is then signed using a key, and signature is stored in a metadata block. For an attacker to successfully modify the content, he needs to have the key to correctly sign a new mapping table (which contains a new root hash). A replay attack is theoretically possible if an attacker has old data with its valid signature. However, dm-verity is supposed to be used on read-only devices, and the data is not expected to change. A replay attack would thus not accomplish anything.

More recently in 2016, *dm-integrity* [12] was introduced. It supports passive integrity protection for **read-write** partitions. Instead of creating a Merkle tree and storing it in a different device, it concatenates multiple hashes or checksums of blocks of data into a *metadata block* and stores this block inside the device itself. These metadata blocks are **interleaved** with the actual data. Since the metadata blocks consist of unsigned checksums or hashes, dm-integrity protects only against silent corruption and not active attackers (although dm-integrity can be combined with dm-crypt to also protect against attackers, see later). Since a change in data requires a change in the metadata as well, the device could end up in an inconsistent state, for example because of a system crash that happened in between the change of the actual data and metadata. dm-integrity solves this by using a *journal* (Section 2.8) which has a reserved space on the device. Before committing a write to the actual data, it is first written to the journal, after which it actually committed. After a crash the

journal can be consulted to see if something went wrong, and if so try to restore it.

The dm-verity and dm-integrity targets provided some form of data integrity. Data confidentiality can be achieved using FDE. The most popular Linux FDE solution is *dm-crypt* [82] [18]. It provides data confidentiality using transparent encryption and works by encrypting all writes before forwarding the write request to the next layer, and similarly decrypts reads from the previous layer before returning the decrypted data to the caller.

dm-crypt can be completely customised, however, the developers created and recommend using *LUKS* (Linux Unified Key Setup), which is now the standard for Linux disk encryption. It allows a user to have multiple passwords, to change or delete passwords and to facilitate compatibility among other distributions. Using LUKS is not mandatory, but it allows for more usability, automatic configuration of non-default cryptography parameters, and more. One disadvantage of using LUKS is that damage to the header could lead to permanent data-loss. Furthermore, it is obvious that the disk contains LUKS-formatted data. This can be an issue in countries where encryption is not always legal [84].

There are 2 versions of LUKS [31] [13]. Figure 3.2 shows an overview of a LUKS1 formatted disk. The data is encrypted with a *master key* (not chosen by the user, but created from a high entropy source). The partition header contains information about the used cipher, cipher mode, the key length, a UUID, a master key checksum and a salt for each key. It also contains information about the different key slots. Multiple (up to 8) passwords can be added, and the respective Key Material slots, if activated, each store the encrypted master key using its password (processed by PBKDF2 [46]). Thus, supplying one of the passwords unlocks the master key, which in term unlocks the data . This makes adding, removing and changing passwords easy. In case of such a change, it only needs to change a small bit of data, instead of having to re-encrypt the entire disk.

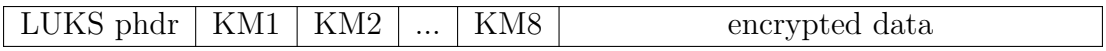


Figure 3.2: LUKS formatted disk. phdr: Partition Header, KM: Key Material

One improvement of LUKS2 over LUKS1 is the ability to also provide integrity protection. It accomplishes this by combining dm-crypt and dm-integrity [12].

dm-crypt is not the only FDE solution. Windows, for example, has BitLocker [30] which acquires the same results as dm-crypt, albeit using different techniques.

In 2017, Chakraborti *et al.* introduced *dm-x* [16]: a device-mapper target that can be seen as a mix between dm-verity and dm-integrity. It extends dm-verity with journaling and read-write capability. To achieve this read-write capability, it needs a small trusted storage, like a trusted local disk, a removable storage device or some trusted network service. This trusted storage needs to keep a 32 byte root hash of the Merkle tree, and a 32 byte secret to generate HMACs of the journal. Upon a read

request, if there's a mismatch between the Merkle tree and block hash, dm-x will throw an I/O error. Like in dm-integrity, dm-x uses a journal to ensure consistency among data and the tree after a crash. The journal design stores the old and the new hash of the data, and ensures that the journal entry is committed to the device before the data and hashes are written. Unfortunately, apart from the initial paper, no further information or implementations of dm-x seem to be available, so no tests or comparisons could be done.

Table 3.1 compares the security features of the discussed device-mapper targets.

		Confidentiality	passive Integrity	Authentication	**R**ead / **W**rite
dm-verity			×	×(because of read-only)	R
dm-integrity			×		RW
dm-crypt	LUKS1	×			RW
	LUKS2	×	×	×	RW
dm-x		×	×	×	RW

Table 3.1: Comparison of the security features of different dm targets.

3.3 Page-level protection

The solutions explained in the previous section require either read-only mode, or some permanent structure on the device itself. However, RAM devices often do not satisfy these requirements, as they are often volatile and required to be read-write. Two page-level protection methods are described below.

3.3.1 AMD SEV-SNP

AMD introduced *Secure Encrypted Virtualization* (SEV) in 2016, isolating Virtual Machines (VM) from their hypervisor. This proves to be very useful in environments like the cloud, where the intent is to build an architecture where there is no need to trust the host anymore. More recently, AMD added *Secure Nested Pages* (SNP) to SEV, adding memory integrity protection. According to AMD [4], *The basic principle of SEV-SNP integrity is that if a VM is able to read a private (encrypted) page of memory, it must always read the value it last wrote.* This should hold regardless of what happens in the memory, or even when the process is migrated to another host. [4]

This memory integrity protection is achieved by using a *Reverse Mapping Table* (RMP). It is a table containing an entry for every page that might be used by a virtual machine. These entries keep track of who owns that page. When the memory is accessed, after the translation from virtual address to physical address, the RMP is checked with the physical address as entry. If the caller does not own this page, a page fault occurs and the access is denied. Since reads do not modify data, read requests generally do not require an RMP check.

The introduction of the RMP requires new CPU instructions to modify the RMP, which allows the hypervisor to assign pages to guests, take pages back, etc.

3.3.2 Intel SGX MEE

Intel's *Software Guard Extensions* (SGX) [38], [58] provides a way, similar to AMD SEV, to create and work in a hardware-assisted Trusted Execution Environment (TEE), also called an enclave. Intel SGX also provides a technique to provide memory integrity protection, called *Memory Encryption Engine* (MEE) [37].

Typically, a CPU has a memory cache, a small storage unit that delivers faster results than the actual RAM. When the needed value is not present in the cache, the request is handled by the Memory Controller (MC), before being forwarded to the RAM. Intel MEE is an extension of this MC. The RAM is split up in three regions: the protected region, the seized region, and the general region. The protected region is to be protected by the MEE. The seized region is used by the MEE to build, update and verify the integrity tree of the protected region. Lastly, the general region is unprotected memory that is handled by the MC directly. Reads are decrypted by the MEE after fetching it from the RAM, and writes are encrypted before sending it to the RAM. The MEE also automatically updates and verifies the integrity tree.

The integrity tree consists of MAC tags and nonces. These nonces in turn consist of the address of the storage unit, and a counter. Similar to a Merkle tree, this tree has one root level. This root is considered to be stored in a trusted environment. In a simplified view, the tree consists of different levels with nonces and a tag over these nonces. Every data unit has a MAC tag and a nonce. This nonce is first verified by descending down in the tree and verifying the nonces stored at each level. If the last nonce is verified (the nonce belonging the the data unit), a new MAC tag can be computed over the data and nonce. This value is compared to the stored tag to verify integrity.

3.4 Length-preserving storage protection

The previous sections introduced techniques for data confidentiality and integrity. The techniques providing data integrity all have one thing in common: they require some extra storage to store the integrity metadata. Even more, page-level integrity protection requires specific CPU instructions and new hardware to be added. In length-preserving (LP) protection, the intent is to remove the need for extra storage while still providing confidentiality and integrity protection. This section summarizes previous research on LP protection.

Oprea et al. [63] provide a solution for block-level integrity protection for blocks stored on an untrusted storage server. It does this by storing integrity metadata on the trusted client device, and focuses on keeping the trusted space needed as small as possible. It is not a length-preserving protection solution itself, but provides

insight into how to construct one. To keep this size small, it uses the following two properties:

1. Most blocks written to a disk have low entropy.
2. An attacker modifying a block without knowing the encryption key will result in the decrypted block having high entropy (with very high chance).

Their solution exploits these properties by only computing and storing integrity metadata in case a written block has high entropy. To do this, they use a statistical test *IsRand(M)*, which returns 1 with high probability if M is a uniformly random block, and 0 otherwise.

Before encrypting a block M, *IsRand(M)* is computed. After a positive result, the client stores a hash of that block. Otherwise, nothing needs to be done. Similarly, on a read after decrypting a block, the client computes *IsRand(M)* again. If the result is positive, a hash is computed. If this hash is not present or equal to a stored hash, the block is assumed to have been tampered with. According to their findings, most written blocks have a low entropy. This means that the proposed scheme can provide data integrity with very small storage needed. However, since *IsRand* is not perfect, and a tampered block could decrypt into a low-entropy block, the scheme can sometimes (although very unlikely) fail. Another vulnerability it does not prevent is a replay attack. A block with some data creates the same hash every time. An attacker can thus eavesdrop the hash and data, and replace it at a later stage.

In a follow up paper [62], the authors go further and claim to provide data integrity with only a small, constant trusted storage. Furthermore, it aims to solve the replay attack vulnerability present in their previous paper. The approach is again based on the former two properties. In this new solution, write counters are used: a number representing the amount of writes done to that block index is stored, preventing replay attacks. Keeping track of a counter for every block index would require a relative big additional storage. Hence, the authors propose a more space-efficient method of storing the counters. According to their observations, most file accesses are sequential, meaning most write counters of adjacent blocks are correlated. Instead of implementing write counters, counter intervals are stored instead. These are stored as an array containing the indices where a new counter starts and an array containing the values of the counters for each interval. They propose two solutions: one of which is closely related to the solution proposed in this thesis. The idea is to use compression in some blocks to reduce the size of the block. The integrity metadata can then be stored inside the block itself. For incompressible blocks they use a Merkle tree in a small trusted storage. Together with the write counter (intervals), this prevents replay and reorder attacks.

4 Research question

The previous sections gave insight on how data confidentiality and integrity in various storage devices is usually achieved. In short, confidentiality is achieved by encrypting the data using a secret key, and integrity is achieved by computing integrity metadata which can be verified the next time the data is read to detect unauthorized changes. Depending on the encryption method used, an IV might also be used; this IV is then also needed during decryption. The conventional solution is to use external storage to store the integrity metadata and IV. However, this is not always efficient or even impossible.

Previous solutions aiming to provide length-preserving integrity protection by Oprea *et al.* [62] are not truly length-preserving, as they still need some external trusted storage. This begs for the following research question:

Is it possible to create a method that provides complete length-preserving integrity and confidentiality protection?

Such a method should not use any external data ——the input data should have exactly the same length as the output data. Figure 4.1 illustrates the research question.

Integrity protection schemes always produce more data than they consume because of the creation of integrity metadata. The key insight of this work is that, by compressing the input block before encryption, it is possible to make room for integrity metadata, such that the metadata can be placed in-situ in the compressed block, instead of in external storage.

Shannon's noiseless source coding theorem (Section 2.2) shows that there is a theoretical limit on the achievable compression rate, and that there exist data that cannot be compressed. Already compressed or encrypted data has a very high entropy, making it on average incompressible. Obviously, the in-situ approach does not work for incompressible blocks. The main research in this thesis thus aims to provide a solution that should work regardless of the underlying data, also being able to protect incompressible blocks.

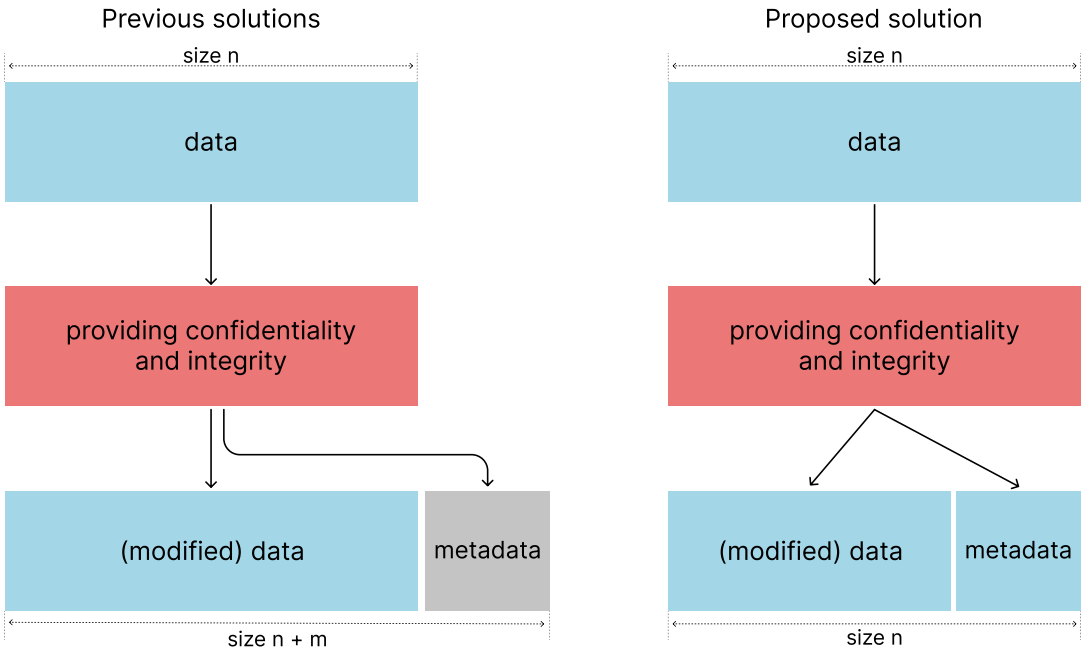


Figure 4.1: The difference between previous solutions and our proposed solution to provide confidentiality and integrity. Left: current methods providing confidentiality and integrity cause data expansion. Right: length-preserving methods providing confidentiality and integrity, studied in this thesis, use compression to make room for in-situ metadata in blocks.

5 LP-SP: Length-preserving Storage Protection

This section introduces our design, called LP-SP, which provides complete LP confidentiality, authentication and integrity protection

To make this section easier to understand, some terminology is used:

- Reader: an entity that reads blocks of data from insecure storage
- Writer: an entity that writes blocks of data to insecure storage
- Encoder: the entity that transforms a block coming from a *writer* into a different format to be stored on insecure storage
- Decoder: the entity that transforms a block coming from insecure storage back into its original format to be read by the *reader*.

5.1 Threat model and security goals

Constructing and understanding the threat model is critical to designing a secure architecture. We assume that the attacker has unlimited access to the storage unit, allowing him to perform arbitrary read and write operations. Key management is considered to be out of scope; we assume that authorized encoders and decoders have the correct key, but that unauthorized entities such as the attacker, do not. There are four types of attacks that such an attacker can do:

R1 Eavesdropping: an attacker tries to extract information by passively looking at all the changes of the device.

R2 Reordering: an attacker swaps blocks around.

R3 Modification: an attacker modifies stored blocks, replacing, inserting, deleting or truncating the data.

R4 Replay: an attacker overwrites a block with another block that was previously there.

For example, an eavesdrop attack can be launched by an attacker aiming to get someone's password. Users sometimes store passwords in plaintext on their computer.

Even if the user doesn't do this, passwords may also be temporarily in memory. Since memory can sometimes be swapped out to the disk, both cases might expose the password unprotected on the disk. If an attacker has access to the disk, it is possible to eavesdrop and extract the password from other data.

Another example, assume a storage device in an energy company contains information about its users that owe the company money. An attacker that has access to the device can capture the data right after he pays the company (at that time, the debt is 0). In a later stage, when his debt has grown, he can overwrite the data with the data he earlier captures. Even though he does not know the actual plaintext, he can succeed in reducing his debt to 0 again.

By achieving the security properties of Section 2.5, R1, R2 and R3 can be achieved. The use of encryption (applied correctly) ensures protection against R1. R2 can be achieved by using a tweakable or authenticated block cipher, using the block number as tweak or AAD respectively. To protect against R3, tags are created of the blocks which can be used to validate their authenticity. Protecting against R4 is usually done by including nonces or write counters: values that get updated every time the data updates.

In Section 5.4 we discuss how and which of these threats are prevented.

5.2 Architecture

The main goal of LP-SP is to store integrity metadata in-situ. On a write, it needs to create the integrity metadata, compress the block, store the integrity metadata inside the compressed block and encrypt the block. Similarly, on a read, it needs to decrypt and decompress the block, and validate the integrity of the decompressed block.

First, we explain how incompressible blocks are handled in Section 5.2.1. Then, Section 5.2.2 describes the procedure of reading and writing blocks when LP-SP is used. Next, Section 5.2.3 illustrates the format of a block after being compressed by LP-SP. Finally, Section 5.2.4 describes possible improvements that might increase the efficiency and security of LP-SP.

Then, we explain the used compression algorithms in Section 5.3.1. The encoder needs to choose which, if any, compression algorithms and combinations thereof to use. Section 5.3.2 describes this more in detail.

5.2.1 Handling of Incompressible blocks

Since Shannon's noiseless source coding theorem (Section 2.2) states that not all data is compressible, LP-SP should be prepared to always handle incompressible blocks. For example, the data that LP-SP processes might already be encrypted or compressed. Since cryptographic and compressed data generally have high entropy, those blocks will often be incompressible. LP-SP needs to also provide confidentiality

and integrity for incompressible blocks. To achieve this, two problems need to be solved:

1. How does the decoder know whether a block was compressed or not?
2. Where and how is the integrity metadata of incompressible blocks stored?

The first problem is discussed in Section 5.2.1.1, the second problem is discussed in 5.2.1.2.

5.2.1.1 Magic number as compressibility indication

A **magic number** is a sequence of bytes which should occur rarely in the blocks. If it is injected in the header, it can be used to identify compressible blocks by checking the presence of the magic number. If it's present it must mean the block was compressed.

The weakness of the magic number approach that an incompressible block containing that magic number on the correct position[1] will be falsely signaled as compressed. The decoder will try to decompress the block. If not already failed, the integrity check will fail, resulting in the decoder alarming an error. The odds of this happening can be decreased by increasing the magic number size. This, in turn, requires the block to be more compressible, as the header size has to increase.

The choice of a magic number is not trivial. It needs to be a sequence of bytes that has a low probability of occurring in an incompressible block. We can summarize the four ways a block might be incompressible:

(1) The block contains already encrypted data.

(2) The block contains (pseudo)random data (e.g. the result of a random number generator).

(3) The block contains already compressed data.

(4) The block contains none of the above, and is incompressible by 'coincidence'.

Based on these four cases, we can make an informal decision on what the magic number should be. Because encrypted data is supposed to look random, we assume that data in cases (1) and (2) is independent and identically distributed (i.i.d)[2]. In this case, the choice of magic number has no effect at all, because every combination is equally likely to occur. Case (4) is assumed to be a coincidence, e.g. a file format that usually is compressible, except not in this one case. Since there is no knowledge over the data, there is not much to say on the effect of the magic number. Finally, in case (3) we know the data was compressed. In this case, it can be assumed that the data will often have a high entropy: it does not make a lot of sense that the output of a compression algorithm has low entropy, because it could be compressed

[1] Correct position: At the position where a compressible block would store the magic number

[2] This is not the case, but we need to make approximations or assumptions in order to get some knowledge on what the magic number should be.

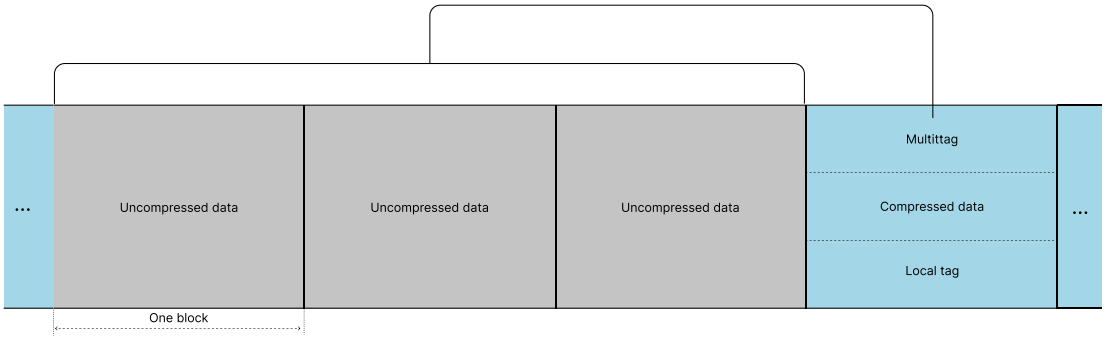


Figure 5.1: An illustration of how a multitag of a compressed block covers the plaintext of the preceding chain of incompressible blocks. The blocks in this figure are already decrypted.

again. As a result, we believe it is a good idea to use as magic number a sequence of null bytes.

5.2.1.2 Storing the tag of an incompressible block in the next compressible block

If a block compresses enough, the tag of a preceding incompressible block can be stored in it. Incompressible blocks can thus store their integrity metadata in the next compressible block. The problem with this solution it it puts a higher constraint on the compressibility requirement[3]

To solve this, instead of computing a tag for every incompressible block individually, we concatenate the plaintext of a chain (sequence) of incompressible blocks and compute the tag over the concatenated blocks. We call the resulting tag the *multitag* of the incompressible chain. The multitag is added to the block's header along with the block's own tag. It means that a compressible block always needs to compress the same amount, regardless of the amount of preceding incompressible blocks. It also prevents complete failure, where there are not enough compressible blocks to store the meta-data of all the other blocks, assuming there is at least one compressible block. Figure 5.1 illustrates this method.

[3]Adding another tag to a compressed block requires it to be recompressed to a smaller size. It can happen that this is not possible anymore, rendering this block as incompressible. The next compressed block will then get an even higher compressibility requirement, etc.

5.2.2 Reading and writing a block with LP-SP

Figure 5.2a shows the procedure of **reading** a block in LP-SP. It is best explained with an example:

Let's say you have three uncompressed blocks (B1, B2 and B3) in a row, followed by a compressed block (B4). The reader wants to read B2 (so $i = 2$).

1. Read and decrypt B2. It is not compressed. Go to part B
2. Read and decrypt B3. It is not compressed. Concatenate the plaintext of B3 with B2. Go back to B
3. Read and decrypt B4. It is compressed. Read multitag X. Compute the tag over the concatenation of blocks B2 and B3 and compare with multitag X. Since multitag X also depends on C1, it will not match. Go to block $i - 1$ (B1) and go to part C
4. Read and decrypt B1. It is not compressed. Concatenate the plaintext of B1 with B3 and B2. Compute a new tag over B1, B2 and B3. This is equal to multitag X, so (decrypted) B2 is returned.

Figure 5.2b show the procedure of **writing** a block in LP-SP. Parts B and C are best explained using two examples.

The first example covers part B. Let's say you have three uncompressed blocks (B1, B2, B3) in a row, followed by a compressed block (B4). The writer wants to write B to number 2 ($i = 2$), and B is compressible.

1. Compute the local tag, compress B and store the local tag in-situ.
2. Read all preceding incompressible blocks by decrypting, checking for compression and, if uncompressed, moving on the the preceding block.
3. Concatenate all the decrypted blocks just read (B1) and compute multitag $X1$ over it.
4. Copy multitag $X1$ into B, encrypt B and write B to block number 2.
5. Starting from 3 ($i + 1$), read all the following incompressible blocks using the same procedure as (2), but in the other direction.
6. Concatenate the decrypted blocks just read (B3) and compute multitag $X2$ over it.
7. Decrypt the first compressible block B4 and copy multitag $X2$ into B4.
8. Encrypt B4 again and write it again to its position 4.

The second example covers part C. Let's say you have three uncompressed blocks (B1, B2, B3) in a row, followed by a compressed block B4. The writer wants to write B to number 2 ($i = 2$), and B' is incompressible.

1. Read all the blocks in the incompressible chain except for B2. This is like step 2 in the previous example, but going both directions.
2. Concatenate all the decrypted blocks just read as well as B and compute the multitag over it.
3. Decrypt the first following compressible block B4 starting from number 3 (i+1) and copy the multitag into B_c.
4. Encrypt both B and B_c and write them to their respective numbers (2 and 4)

5.2.3 Compressed block format

Blocks that are compressed must store various data. This data can be split up in six parts:

1. The compression algorithm choice
2. The compression data
3. The magic number
4. The local tag
5. The multitag
6. The compressed data

Figure 5.3 illustrates such a block. The compression algorithm choice is most likely only one byte, as one byte has the ability to differentiate 256 different algorithms. Next, the compression data is found which is used by some compression algorithms to be able to decompress the block. This should also not take too many bytes (the current implementation of dm-ci uses three bytes, see later). Further, the magic number is stored, used by the decoder to determine if a block was compressed or not. The fourth and fifth part are the tags that provide integrity. The lengths of these can vary based on the algorithm used. Usual values are between 16 and 32 bytes. These five parts form the *header*. The size of the header will be referred to as HS (Header Size).

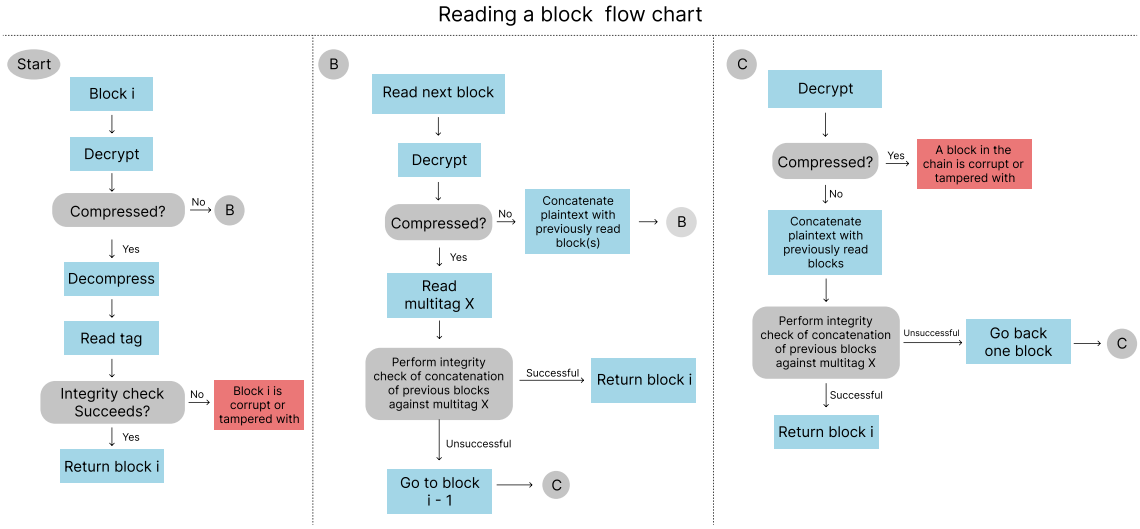
For a block to be compressible, it needs to be able to compress HS bytes, such that the resulting size of the compressed data is $block\ size - HS$.

5.2.4 Possible improvements to LP-SP

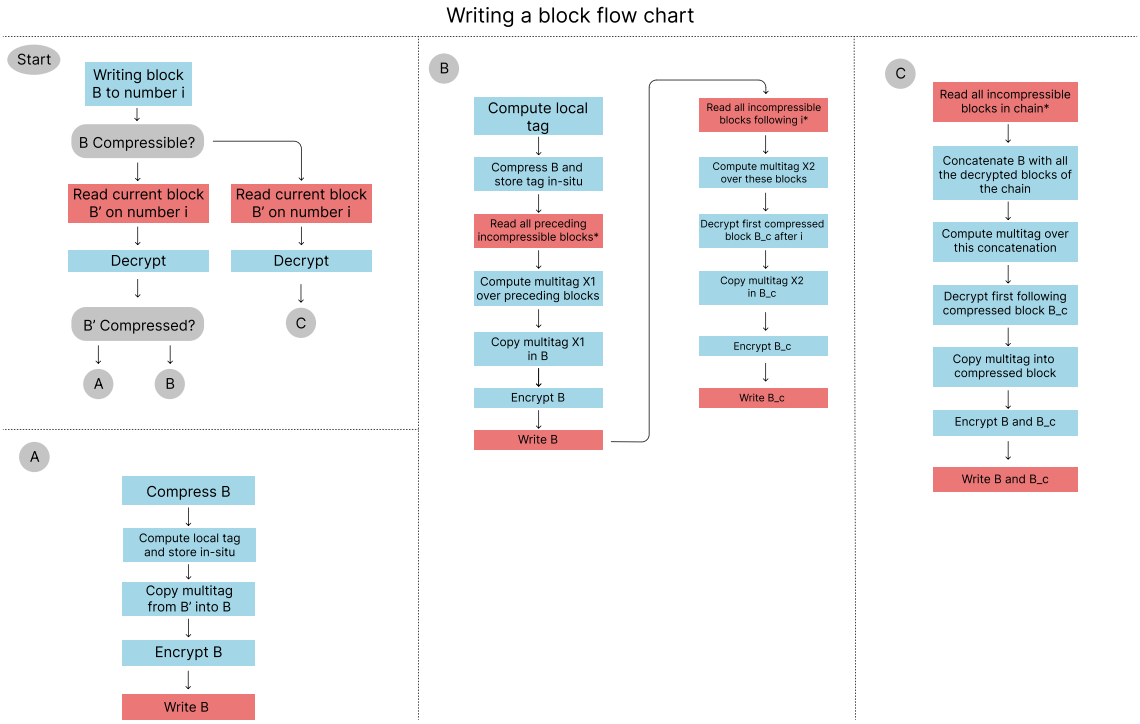
The proposed solution has three drawbacks:

(1) When the decoder falsely identifies a block as compressed, the integrity check will fail.

(2) There is not yet a way to store individual IVs in incompressible blocks.



(a) Procedure of reading block number i in LP-SP. Refer to Section 5.2.2 for a detailed example.



(b) Procedure of writing a block to number i in LP-SP. The blocks with an asterisk are a condensed version of: Read the next/previous block, decrypt it and check if it is compressed. Repeat until you find a compressed block. Refer to Section 5.2.2 for a detailed example.

Figure 5.2

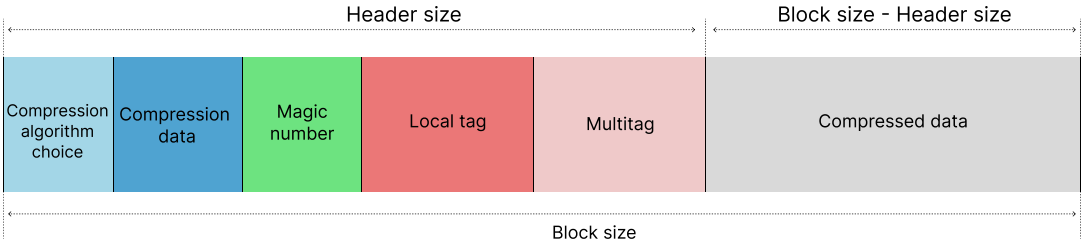


Figure 5.3: The format of a compressed block by LP-SP. It contains five parts; compression algorithm choice and compression data are needed to correctly decompress the block. The local tags is needed to verify integrity of the block itself, the multitag is needed to verify integrity of the preceding blocks.

(3) Long chains of incompressible blocks require a lot of extra reads every time one block of the chain gets read. Similarly, writes are very inefficient for long chains.

To improve (1), the decoder can assume that when an integrity check fails, the block was a false positive. It then looks for the next compressed block to find the tag. This next compressed block might also be a false positive, and the decoder can again assume this was a false positive. This process can thus repeat itself a specified amount of times, where the user can choose a trade off between correctness and speed. It should be noted a sufficiently long magic number makes the odds of having two false positive blocks in a row extremely low. As an example, let's assume that the incompressible blocks contain i.i.d data and that the magic number is 10 bytes long. Then, 10 bytes can have $256^{10} \approx 1.21 \cdot 10^{24}$ different values, which makes the odds of an i.i.d. block having the magic number extremely unlikely.

For (2), the technique used to combine tags for incompressible blocks cannot be used for IVs. Optimally, every block needs their own IV. One solution is to have all the blocks in the chain use the same IV (stored in the next compressible block). The incompressible chain is then seen as one big protected block, since both the IV and tag cover the entire chain. This way, when one incompressible block is updated, the IV must change and the other incompressible blocks in the chain need to be re-encrypted. Because this solution has not been explored further, the security of this solution is also not known.

Finally, for (3), two improvements can be made: *caching & buffered writes* and using a *different block numbering*. They are explained in their own section.

5.2.4.1 Caching and buffered writes

Depending on the use case, our block sequence might match the sequence of whatever application is using our architecture. As an example, imagine we have a block sequence $b_0\, b_1\, b_2\, b_3 \cdots$ A filesystem on top of our architecture also has some sequence of blocks: $b'_0\, b'_1\, b'_2\, b'_3 \cdots$. If our sequence matches the filesystem's sequence, writing a 4-block file to $b'_i\ b'_{i+1}\ b'_{i+2}\ b'_{i+3}$ results into the filesystem writing to our blocks

$b_j\, b_{j+1}\, b_{j+2}\, b_{j+3}$. In case this file consists of four incompressible blocks, we now have an incompressible chain of length four.

However, since filesystems most likely read complete files and not random blocks, the next time b_j is read, we can assume that $b_{j+1}\, b_{j+2}\, b_{j+3}$ will be read too. Thus, it would be very beneficial to temporarily store the result of the integrity check of this chain. A sequential read to b_{j+1} will not require any extra reads, as the integrity check has recently been validated. The process of temporarily storing data to increase performance is called *caching*.

A similar improvement can be performed for writes: instead of immediately writing every block, writes can be buffered to see if the following writes are sequential. If they are, we can compute the tags together, after which all the sequential blocks are written.

Caching and buffered writes are only useful if:

1. Our block sequence matches the block sequence of the layer above
2. Data is mostly re-read and written sequentially

5.2.4.2 Using a different block sequence

This improvement takes an entirely different approach than the one explained above. Here, we try to have a complete different block sequence than the layer above. If we were to use a deterministic pseudo-random block sequence, sequential reads by the upper layer will seem like total random reads to our layer. For example, reads / writes by the upper layer from block 1 til 4 are transformed into reads / writes by ours layer on blocks 9, 186, 301, 84. This way, clustered incompressible blocks on the upper level (think of an encrypted file or a compressed video) are not longer chained together on our layer. The odds of having a compressible next to one of these incompressible blocks are far higher, and the average incompressible chain length will be lower.

5.3 Implementation

A proof-of-concept of a part of LP-SP was implemented in Linux (kernel version 5.19.0) for the device-mapper layer. The implementation is called dm-ci (device-mapper target providing **C**onfidentiality and **I**ntegrity). More specifically, it implements the read and write operations on compressible blocks. Ultimately, dm-ci should also implement the read and write operations on inompcressible blocks. However, due to time constraints, it was not possible to make dm-ci fully functional before the end of the thesis deadline. Even though most of the incompressible block logic is implemented, it is not yet possible to read/write such blocks.

To make a disk work with dm-ci, all the blocks should be put in the format that is expected by the decoder. This is why a disk (on which the user wants to retain

its data) needs to be initialized: integrity metadata is computed and compressed blocks are compressed. Device-mapper targets are kernel modules, and are thus not suggested to use when iterating an entire disk. Instead, apart from dm-ci, a userspace program was written, called *cisetup* (it can be seen as a similar tool to *cryptsetup* for dm-crypt).

The total lines of code (LoC) for both *cisetup* and *dm-ci* were calculated using *cloc* [19]: at the time of submission, *cisetup* contained 1618 LoC and *dm-ci* contained 2517 LoC, both excluding comments or blank lines.

First, we explain the implementation of the compression algorithms. Then, we describe the *cisetup* tool. Finally, we explain the implementation of dm-ci.

5.3.1 Compression

Most modelling and coding methods were explained in section 2.1. This section covers the implementation and its optimizations, as well as introducing a new methods: *replacement*.

5.3.1.1 Run-length encoding

Run-length encoding is relatively easy to implement. A difference with the example in Section 2.1 is that instead of storing the length-value pairs as $L_0V_0L_1V_1L_2V_2\cdots$, we will store them as $L_0L_1L_2\cdots V_0V_1V_2\cdots$. The reason for this is that this output has some redundancy that can be exploited by applying an MTF transform on the Length and Value sequences.

To be able to revert this transform, all we need to know is the amount of runs (the amount of length-value pairs). Reverting is then done by reading each length(l) and its value(v) and repeating v l amount of times.

5.3.1.2 Replacement

If a block has a sequence of HS bytes that all have the same value, the original first HS bytes can be moved to the start of this sequence. Then our header is injected to the start of this block. Since we have some storage in our header to store decompression information, we can store the offset and value of this sequence. On decompression, the data found at the offset is copied back to the start of the block, and the HS bytes at this offset get their original value again. An example of this procedure with a sequence of numbers 8 and offset 86 is shown here.

1.

Bytes	$0 \dots 85$	$86 \dots 86+HS$	$86+HS+1 \dots end$
Content	data	8	data

2.

Bytes	$0 \dots HS-1$	$HS \dots 85$	$86 \dots 86+HS$	$86+HS+1 \dots end$
Content	our data	original data	original header	original data

5.3.1.3 Move-to-Front

If the bytes are interpreted as unsigned integers, their values lay between 0 and 255 inclusive. Thus, the starting MTF-array is simply an array with incrementing values until 255: $[0, 1, 2, ..., 255]$. The rest of the implementation is very straightforward. However, when the previous transform was RLE (section 5.3.1.1), we know that every value in the value vector is different from the previous value, hence the value-to-be-written will never be 0. We can thus execute MTF as usual, but the resulting values can be decremented by 1: on decoding, we simply increment the read value by 1 and decode the values as normal. This will lower the average bit value, and smaller integers get smaller codewords in interpolative coding.

5.3.1.4 LZ-77

Implementing LZ-77 has to be done carefully, as its performance can decrease dramatically by choosing the wrong lookahead-buffer size and window size. Compared to a generic implementation, the constraints on our data does not provide any optimizations. Tests show that increasing the buffer size and window size over 20 and 30 respectively slows down the application too much to be efficient. More information about the tests and results can be found in Section 6.

5.3.1.5 Fixed-length binary

Some models might reduce the amount of output symbols enough, such that a simple fixed-length coding can be applied. The easiest length is 8 bits (a byte) as it is easier to implement, and most outputs of the models that can be fixed-length encoded have an output between 0 and 255. This can especially happen with RLE, as sparse data will require very little symbols. However, a run with a length of 256 or more can not be encoded like this, as the maximum value of a byte is 255. However, the moment a run has length HS or more, the replacement technique can be applied. Thus, there is no need to worry about runs of length greater than 255.

5.3.1.6 Gamma coding

The γ coding explained in Section 2.1 needs a small adjustment: it is not able to encode the value 0. Since we need to be able to encode 0, we increment every byte value by one before applying the γ coding. A table of the first few numbers with their γ encoding is shown in Figure 5.4.

5.3.1.7 Interpolative coding

An implementation of interpolative coding without any specific optimizations is used, which is straightforward to implement.

Number	Number + 1 in binary	Code
0	1	1
1	10	010
2	11	011
3	100	00100
4	101	00101
5	110	00110
6	111	00111
7	1000	0001000
...	...	...

Figure 5.4: The modified γ coding of the first few integers.

5.3.2 Algorithm choice

It is clear that when multiple algorithms compress enough, the fastest one should be chosen. For most algorithms, predicting the compression rate is almost the same amount of work as actually performing the algorithm. Hence, the implementation in this thesis has a static hierarchy of algorithm combinations that it will try before proceeding to a different combination. This hierarchy is chosen based on speed and compression rate and can be seen in Figure 5.5. It evidently prioritises the fastest and most successful combinations. These measurements were tested, and their results are analyzed in Section 6. Furthermore, we do not intend to compress a block more than it is needed. This was chosen because LP-SP does not gain anything from blocks compressing more than needed (see Section 5.2.1). On the contrary, it might even decrease performance.

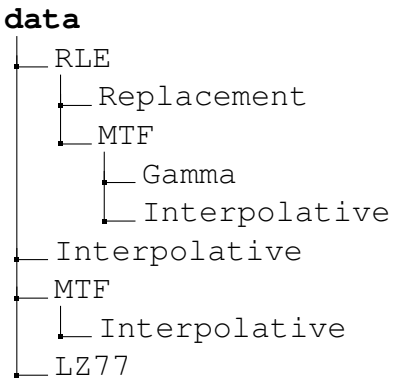


Figure 5.5: The hierarchy of algorithms, with priority order from top to bottom and left to right. For example, RLE + Replacement has priority over RLE + MTF + Golomb; RLE + MTF + Interpolative has priority over LZ77.

One nuance that should be noted is the prioritisation of RLE over Replacement. This is merely an optimization choice: RLE checks the length of every run, and thus also notices when a run has sufficient length to perform Replacement. While Replacement individually can be checked a little bit faster, it is not worthwhile to

perform it first, as on failure RLE will have to pass over all the runs again.

5.3.3 cisetup

Cisetup is a command-line interface tool that can perform two main operations:

1. Check a file or partition for compression rates (*check*)
2. Format a file or partition to be used by dm-ci (*format*)

To achieve this, it contains the implementation of all the compression algorithms explained in the previous section. They were made with a focus on scalability: adding new models or codings should not require a lot of boilerplate code, and apart from the actual algorithm itself, not a lot needs to change.

Check works by iterating over all the blocks in a file or partition to scan every block for their compressibility rate. The user has the ability to disable certain algorithm combinations[4]. The results are then shown per algorithm combination. Furthermore, *check* also prints some information about incompressible block chains like the average length, the standard deviation, the maximum length and the median length. These statistics are very important to maximize dm-ci's performance, and could be used to apply an improvement to reduce the chain lengths as specified in Section 5.2.4. An example of the program's *check* output can be seen in Figure 5.6.

```
Stats for '/dev/nvme0n1p4' (6320896 blocks)

|   nb blocks   |   percentage   |   combination                  |
|---------------|----------------|--------------------------------|
|   1668679     |   26.40        |   RLE                          |
|   1116517     |   17.66        |   Replacement                  |
|   923635      |   14.61        |   RLE -> MTF -> Golomb         |
|   1359281     |   21.50        |   RLE -> MTF -> Interpolative  |
|   0           |   0.00         |   RLE -> Alphabet-Stripping    |
|   0           |   0.00         |   RLE -> Interpolative         |
|   0           |   0.00         |   Interpolative                |
|   31662       |   0.50         |   MTF -> Interpolative         |
|   0           |   0.00         |   LZ77                         |
|---------------|----------------|--------------------------------|
|   5099774     |   80.68        |   Total compressible           |

Total amount of incompressible blocks:  1221122
Incompressible block sequences had lengths with the following stats:
        Average: 14.90        Standard deviation: 264.16      Maximum: 38716
```

Figure 5.6: An example of cisetup's check method. The test drive /dev/nvme0n1p4 consists of 6320896 blocks, of which 80.68% were compressible. RLE was the most chosen algorithm, while RLE -> MTF -> Interpolative was the second most chosen.

[4]Combination: a sequence of compression models and codings

Format also iterates over all the blocks, but actually applies and writes the compressed and encrypted blocks to the device. Ultimately, this should work no matter what the original state of the disk is. However, because *cisetup* is not the main focus of this thesis, cisetup's format command is currently only able to initialize disks to all zeroes, losing the original data. Nevertheless, implementing this feature should not prove too difficult, as most of the logic used in dm-ci can be reused.

5.3.4 dm-ci

As said before, dm-ci currently only functions with compressible blocks, which means that even though most incompressible logic is implemented, it is omitted in this explanation.

The implementation of *dm-ci* contains two design choices that simplified the implementation:

1. The usage of the Linux kernel's crypto API [60] is used for cryptographic function calls. Most functions can be executed both synchronously and asynchronously. In the current dm-ci implementation, all the crypto API calls are doing synchronously. However, an efficient implementation that makes use of concurrency most likely requires the API to be called asynchronously.
2. dm-ci uses AES-XTS as encryption cipher, and $HMAC(SHA1)$ to create the integrity tags. These were selected because these primitives were already available in the Linux Crypto API. However, the recommended cipher to use is AES-GCM, which also generates the integrity metadata.

dm-ci is a kernel module. It uses the device-mapper framework, which provides multiple useful functions, like *constructor*, *map* and *destructor*. The *constructor* / *destructor* functions are executed when the target is initialized / destroyed. Initialization happens when the user uses the *dmsetup* tool to set up the device-mapper. Targets are destroyed when the computer shuts down or the user manually removes the target. Finally, *map* gets executed every time an I/O request is sent to the target. These requests are most often read/write requests which can be modified, cancelled, forwarded, etc. before being sent to the layer below.

The structure that the Linux kernel provides to execute I/O requests is called *bio*: **b**lock **I/O**. These bio requests consist of an array of *bio_vec*. A *bio_vec* refers to a page, with an offset and length, basically representing a continuous range of blocks. The page of a *bio_vec* contains the read data after a read request, or the to-be-written data during a write request. The most important members of the bio struct are summed up:

- **bi_io_vec**: The array of *bio_vec*s.
- **bi_end_io**: The function that should be executed when the request is finished.
- **bi_iter**: An iterator that iterates over *bi_io_vec*:

- **bi_flags**: Indicators that contain the direction of the bio (read / write), and other flags important for the lower levels executing the bio

The *map* function gets executed whenever a bio request is submitted to the dm target. The target can then modify most of the parameters of the bio request, including the *bio_vec*s belonging to this bio. Additionally, the target can create new bio requests in case it needs to read / write data from / to another block. Because the behaviour of dm-ci is very different for reads and writes, they are split up in their separated for clarity.

5.3.4.1 Reads

The *map* function intercepts the read request before it is actually executed, hence, the read data is not yet available when this function is called. Instead, the *end_io* method of the request needs to be changed to a custom made function, and the original *end_io* method needs to be stored in memory. This *end_io* method gets executed when the read request finished, and thus when the read data is available. Now, in the custom function, the read data can be decrypted using the kernel's crypto API.

The plaintext's tag is calculated (also using the kernel's crypto API) and compared with the saved tag. In case these are different, a *DM_MAPIO_KILL* is returned in the *map* function, which will throw an error to the higher layers. No matter if the reads were valid or not, in the end, the *end_io* method should be set back to the original one. Otherwise, it will cause a deadlock in the layer above (the layer above is waiting for its *end_io* function to complete, but it will never do).

5.3.4.2 Writes

Opposed to reads, writes need to be modified before the request is actually executed. Because the current implementation only allows compressible blocks, we know that the multitag of the block-to-be-overwritten is empty. This block can thus be overwritten without analysing it first. When the write request enters the *map* function, its tag is first computed. Then, it is compressed and its tag is put into the compressed block. Finally, the block is encrypted and written.

5.4 Security of LP-SP

Section 5.1 explained the possible attacks on a storage unit. This section will evaluate if LP-SP reaches the security goals, and possible improvements.

LP-SP is a form of FDE, in which it encrypts every block individually. FDE implies confidentiality, and thus also prevents attacker from eavesdropping (R1). However, LP-SP currently does not provide a clear solution on how to store IVs of incompressible blocks. The proposed solution where incompressible blocks all use the same IV

(Section 5.2.4) is not proven secure, and may thus reveal patterns or allow for other attacks.

To protect against modification (R2), every block is integrity protected by a tag. This tag is present in the block itself in case the block was compressible, or in the next compressible block. LP-SP cannot guarantee the storage of individual tags for every block. If an incompressible chain has length two or higher, then the entire chain will be protected by one tag. It still prevents an attacker from modifying data in one of the incompressible blocks. However, it has the downside of some blocks losing independence, which, according to Benadjila *et al.* [9], is not optimal.

A reordering attack (R3) is prevented by using a tweakable cipher like AES-XTS (a), or using an authenticated encryption cipher like AES-GCM with the block number as AAD (b). In the case of (a), swapping blocks will result in a different plaintext, which will fail the integrity check. In the case of (b), the tag can even be verified before decryption. The tag containing the old block number will not match a newly computed tag using the new block number as AAD.

Finally, LP-SP does not provide a countermeasure to replay attacks. All the data to protect a compressible block is stored inside the block itself. An attacker can thus copy an old compressed block to the same position at a later stage. However, if the incompressible chain before the compressible block has since been updated, the multitag of the replayed block will not be valid anymore. In some way, the incompressible blocks sometimes prevent a replay attack, but this is not a security guarantee. According to Benadjila *et al.* [9], to prevent replay attacks, some trusted external storage is needed. The goal of LP-SP is not to use any external storage. Hence, the current design of LP-SP can not prevent replay attacks.

6 Empirical evaluation of dm-ci

LP-SP has been implemented as a device-mapper target in Linux. Thus, to be able to compare, benchmark and evaluate dm-ci, the tests were done on Linux, often comparing with other device-mapper targets that achieve confidentiality and data integrity as well. The most used targets are dm-crypt, dm-integrity and dm-verity. Furthermore, LP-SP is very dependent on the compressibility rates of the underlying data. This varies greatly depending on the context. Multiple tests were done to test and visualise the compressibility rates of entire disks, partitions and memory dumps of processes. The following sections go more in depth on the tests performed and their results.

All the tests were performed on a virtual machine (Virtualbox) running Ubuntu 20.04 with Linux kernel 5.19.0, 4096MB of memory, 33MB of virtual memory and a 4.4GB SATA drive. The host machine has an AMD Ryzen 9 5900X CPU and 32GB of Memory.

6.1 Speed performance

Disk bandwidth was tested with four different dm targets: dm-crypt, dm-integrity, dm-verity and dm-ci. Both dm-verity and dm-integrity were set up using the default parameters, providing only integrity protection. Dm-crypt uses LUKS2, providing both confidentiality and integrity protection using HMAC(SHA256). The most interesting comparison for dm-ci is thus dm-crypt, as their security properties are most related. Note that dm-ci has biased results, because of two reasons:

1. The tests were only executed with compressible blocks, since dm-ci does not allow incompressible blocks yet. Incompressible blocks likely form the main latency in dm-ci. Nevertheless, the test results can be used to approximate how dm-ci would perform in environments with high compression rates.
2. The other targets, especially dm-crypt, have been in de Linux kernel for a long time. They have contain many hours of development and are thus written more optimised. On the contrary, dm-ci was implemented on a relatively short time, without much focus for optimisations.

The following tests were performed using the *fio* [6] tool. The tests use direct IO and the libaio engine, and were run for two minutes.

Test 1 was a sequential read, using an IO queue depth of 16. This means that the queue with requests for the disk can have 16 requests, which gives the disk the opportunity to use parallelisation to improve the bandwidth.

Test 2 was also a sequential read, using an IO queue depth of only one.

Test 3 was a random read, using an IO queue depth of 16.

Table 6.1 shows the results of these tests. Unsurprisingly, dm-integrity is the fastest in all three tests. The second fastest in all tests is dm-verity. Both of these results are normal, as neither target encrypts the data. Dm-verity is slower than dm-integrity because dm-integrity does not need to traverse a Merkle tree. Instead, it calculates a simple checksum over the data, requiring very little work. Dm-ci and dm-crypt have closer results, except for test 3, where dm-ci is significantly faster.

Target	Test 1	Test 2	Test 3
dm-crypt	83.1	48.5	63.8
dm-ci	86.0	47.5	91.9
dm-integrity	173	71.2	170
dm-verity	108	59.8	113

Table 6.1: The bandwidth (MB/s) of the four device-mapper targets. The three tests are explained in the section above.

6.2 Compression

Compression algorithms are tested both on compression rates in different contexts, as well as speed. Throughout the compression-related tests, the compression rates displayed are calculated as $\frac{amount\ compressible\ blocks}{total\ amount\ blocks}$, which may be lower than the compression rate formula explained in Section 2.1. The tool used to calculate the compression rates is *cisetup*, as explained in the previous section.

The compression tests are split up in three parts. First, we take a look at the performance of each compression algorithm. Next, we test the compression rate of files and directories. Lastly, we look at the compression rate of memory of running processes. Based on this information, a decision can be made which compression algorithms to use and which order to try them.

6.2.1 Performance

The *cisetup* tool has a method built-in that logs the average time it took to process a block. We can check the speed of each combination by disabling all the combinations except for one, and letting the tool iterate over incompressible data, forcing the algorithm to run until its very end. Table 6.2 shows the results.

RLE and Replacement are very fast, requiring only $19\mu s$ per block. These are exactly the same value because replacement only gets executed once RLE finds a run that is

long enough, essentially doing exactly the same when checking incompressible data. Furthermore, we see that Interpolative and RLE -> Interpolative are rather fast too. Then, the three-stage combinations RLE -> MTF -> Golomb/Interpolative take about 200μs. Lastly, we see that LZ77 is very slow compared to all the other combinations. The two parameters of LZ77 (lookahead buffer size & window size) can be reduced to improve the speed.

Algorithm	Time (μs) / block
RLE	19
Replacement	19
RLE -> MTF -> Interpolative	195
RLE -> MTF -> Golomb	215
RLE -> Interpolative	88
Interpolative	40
MTF -> Interpolative	128
LZ77	1139

Table 6.2: Average time per block per compression algorithm

6.2.2 Compression rates: Files and directories

Figure 6.1 shows an overview of the compressibility rate of common directories and common file formats using all the compression algorithms mentioned here. Uncompressed files like TXT files and source code have extremely high compression rates, reaching 99% and higher. Furthermore, binary files (the content of /usr/bin and /usr/sbin) also reach 98% and higher. On the contrary, the already compressed files (PDF, PNG, JPG, MP4 and MP3) are hardly compressible. The */boot* directory has a compression rate of 54%. This can be explained by the *vmlinuz* and *initramfs* files present in that directory, which are two relatively big compressed files, occupying about 50% of the directory. Lastly, the entire disk has a compression rate of approximately 80%.

Table 6.3 shows the detailed result of the entire disk. The top 4 combinations were able to compress about 80% of the blocks. With the exception of MTF -> Interpolative, the other combinations could not compress a single block. This might be because the combination is simply not suitable, or because another combination was already able to compress it.

Table 6.5 shows some statistics on the left about the incompressible chain lengths of the disk. The average is almost 15, which is relatively big compared to the fact that 80% of the blocks are compressible. The standard deviation, maximum and median tell us that this average is so high because of a few extremely long chains. These chains are most likely part of a compressed or encrypted file.

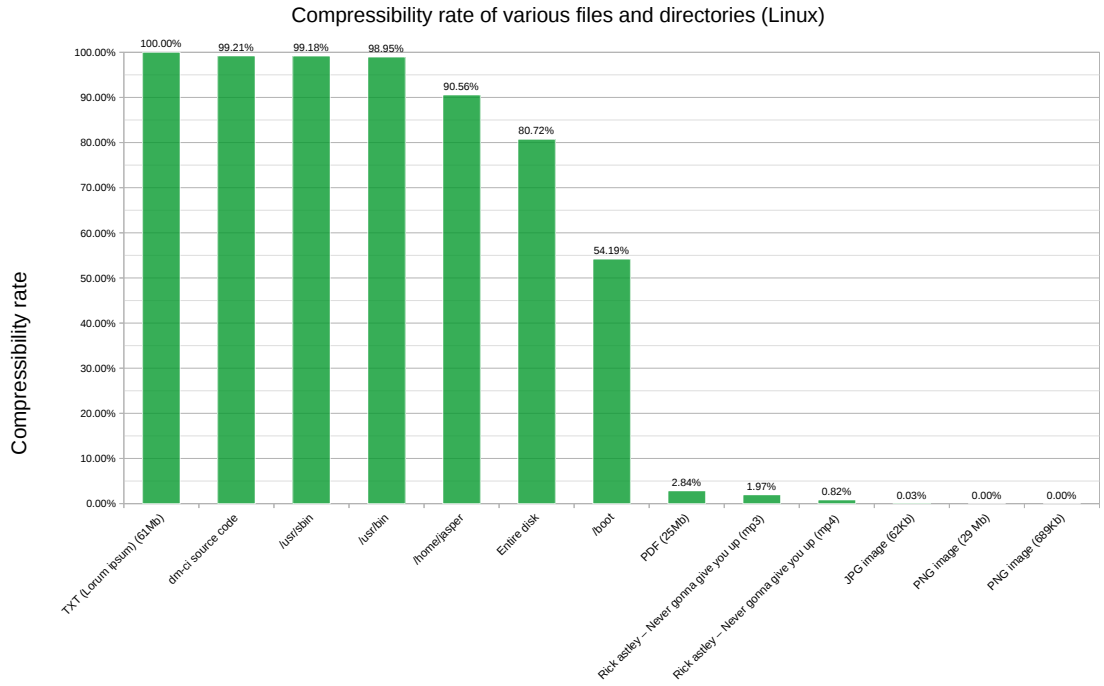


Figure 6.1: Compressibility rates of common directories and file formats in Linux.

Number of blocks	Compression rate (%)	Combination
1668679	26.40	RLE
1116517	17.66	Replacement
923635	14.61	RLE -> MTF -> Golomb
1359281	21.50	RLE -> MTF -> Interpolative
0	0	RLE -> Interpolative
0	0	Interpolative
31662	0.5	MTF -> Interpolative
0	0	LZ77
5099774	80.68	**Total compressible**

Table 6.3: The detailed result of compression rates on the entire disk.

6.2.3 Compression rates: Memory

The memory of running processes was tested by dumping the memory to a file (using *gcore* [33]), and then using *cisetup* to generate statistics of the dump. Figure 6.2 visualises the results of a few processes. Its results are very impressive, as all of the processes have compressibility rates higher than 99%. Table 6.4 shows the detailed information of the Gnome calculator. The reason for the extremely high Replacement compression rate is most likely the fact that the memory in a process is very sparse: its different sections are very far apart, and a lot of null bytes are present throughout the memory. However, this does not mean every process will compress this well. It is possible that an encryption program is encrypting a big

file, which temporarily stores the result in memory.

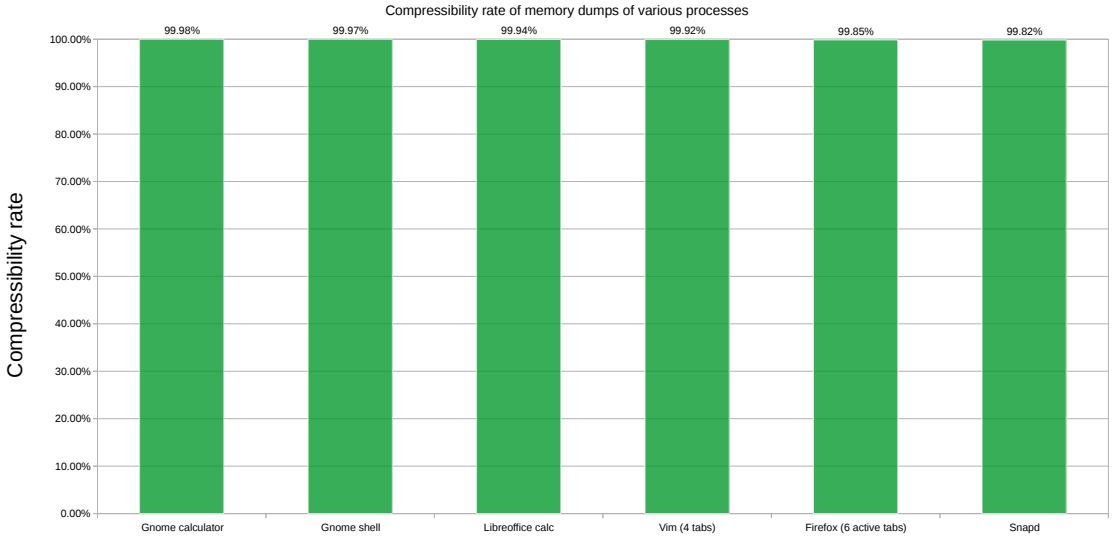


Figure 6.2: Compressibility rates of the memory of common processes in Linux.

Number of blocks	**Compression rate (%)**	**Combination**
3934	2.83	RLE
131579	94.76	Replacement
3025	2.18	RLE -> MTF -> Golomb
284	0.2	RLE -> MTF -> Interpolative
138822	99.98	**Total compressible**

Table 6.4: The detailed result of compression rates of the gnome calculator memory dump. The combinations with 0% compression rate are left out.

Incompressible chain statistics	Entire Disk	Gnome calculator memory
Average	14.90	1.88
Std Deviation	264.16	1.23
Median	2	1
Maximum	38716	5

Table 6.5: The incompressible chain length statistics of left: the entire disk and right: the gnome calculator memory dump.

7 Conclusion

In this thesis, I have investigated the research question: is it possible to create a method that provides complete length-preserving integrity and confidentiality protection? The answer to the question is *yes*.

The thesis presented a design for block-level length-preserving storage protection, called LP-SP. Previous solutions providing block-level storage protection need external storage to store metadata in. This external storage is not always easy or possible to provide. LP-SP succeeds in providing data confidentiality and integrity without the need of any external storage. LP-SP achieves this by compressing blocks using fast and non-statistical compression algorithms. Then, the integrity metadata (tag) that is calculated over the uncompressed data is stored in the space that was just freed by the compression. Incompressible blocks are protected by calculating a tag over all the incompressible blocks in a sequence, and storing this multitag in the next compressible block.

A device-mapper target *dm-ci* was implemented as a proof-of-concept of LP-SP. Another tool, called *cisetup* was implemented to facilitate checking disk statistics on compression rates, as well as giving useful information about the length of incompressible chains. Cisetup also facilitates formatting a disk, such that it can be used by dm-ci.

Two different types of tests were performed. The speed performance tests showed that, in an environment with no incompressible blocks, dm-ci had a relatively high bandwidth. It was faster than dm-crypt, which also provides integrity and confidentiality. The compression rate analysis was performed on different types of data, and showed promising results especially for memory pages. Both of these results hint towards memory being very suitable to use LP-SP in.

Bibliography

[1] L. von Ahn, M. Blum, N. J. Hopper, and J. Langford, "Captcha: Using hard ai problems for security," in *Advances in Cryptology — EUROCRYPT 2003*, E. Biham, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 294–311, ISBN: 978-3-540-39200-2.

[2] A. Al Mamum, G. Guo, and C. Bi, *Hard Disk Drive, Mechatronics and Control*. Boca Raton: CRC Press, 2007, ISBN: 9781315222134.

[3] S. Al-Kuwari, J. H. Davenport, and R. J. Bradford, "Cryptographic hash functions: Recent design trends and security notions," *IACR Cryptol. ePrint Arch.*, p. 565, 2011. [Online]. Available: http://eprint.iacr.org/2011/565.

[4] AMD, "Strengthening vm isolation with integrity protection and more," *White Paper, January*, 2020.

[5] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, 1.00. Arpaci-Dusseau Books, Aug. 2018.

[6] J. Axboe. "Flexible I/O tester." version 3.32. (2022).

[7] F. Bauer, *Decrypted Secrets*, 4th ed. Berlin: Springer, 2007, ISBN: 978-3-540-24502-5.

[8] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *Advances in Cryptology — ASIACRYPT 2000*, T. Okamoto, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 531–545, ISBN: 978-3-540-44448-0.

[9] R. Benadjila, L. Khati, and D. Vergnaud, "Secure storage—confidentiality and authentication," *Computer Science Review*, vol. 44, p. 100 465, 2022, ISSN: 1574-0137.

[10] J. Bentley, D. Sleator, and R. Tarjan, "A locally adaptive data compression scheme," *Communications of the ACM*, vol. 29, pp. 320–330, 1986.

[11] M. Broz. "Dm-verity: Device-mapper block integrity checking target." (2020), [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMVerity (visited on 03/22/2022).

[12] M. Broz, M. Patocka, and V. Matyas, "Practical cryptographic data integrity protection with full disk encryption extended version," *CoRR*, vol. abs/1807.00309, 2018.

[13] M. Brož, "LUKS2 on-disk format specification," Tech. Rep., 2022.

[14] C. Brzuska and V. Lipiäinen. "Companion to cryptographic primitives, protocols and proofs." (2021), [Online]. Available: https://github.com/cryptocompanion/cryptocompanion (visited on 04/08/2022).

[15] R. Card, T. Ts'o, and S. Tweedie, "Design and implementation of the second extended filesystem," in *Proceedings of the First Dutch International Symposium on Linux, 1995*, 1995.

[16] A. Chakraborti, B. Jain, J. Kasiak, T. Zhang, D. Porter, and R. Sion, "Dm-x: Protecting volume-level integrity for cloud volumes and local block devices," in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, ser. APSys '17, Mumbai, India: Association for Computing Machinery, 2017, ISBN: 9781450351973.

[17] A. Clauset, "A brief primer on probability distributions," in *Santa Fe Institute*, 2011.

[18] J. Cogswell. "How to encrypt a linux file system with DM-Crypt." (2015), [Online]. Available: https://www.linux.com/training-tutorials/how-encrypt-linux-file-system-dm-crypt/ (visited on 03/21/2022).

[19] A. Danial. "Cloc(1) - linux man page." (2022), [Online]. Available: https://linux.die.net/man/1/cloc.

[20] P. J. Denning, "Virtual memory," *ACM Comput. Surv.*, vol. 2, no. 3, pp. 153–189, 1970, ISSN: 0360-0300.

[21] D. Dhamdhere, *Operating Systems*, 1st ed. USA: McGraw-Hill, Inc., 2008, ISBN: 0072957697.

[22] D. Dolev, C. Dwork, and M. Naor, "Non-malleable cryptography," in *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, ser. STOC '91, New Orleans, Louisiana, USA: Association for Computing Machinery, 1991, pp. 542–552, ISBN: 0897913973.

[23] M. J. Dworkin, "Recommendation for block cipher modes of operation :" Tech. Rep., 2001.

[24] M. Dworkin, "Nist special publication 800-38e," *NIST Special Publication*, vol. 800, no. 38E, 38E, 2009.

[25] M. Dworkin, E. Barker, J. Nechvatal, *et al.*, *Advanced encryption standard (AES)*, en, 2001.

[26] "eCryptfs - ArchWiki." (2022), [Online]. Available: https://wiki.archlinux.org/title/ECryptfs.

[27] P. Elias, "Universal codeword sets and representations of the integers," *IEEE transactions on information theory*, vol. 21, no. 2, pp. 194–203, 1975.

[28] P. Fenwick, *Universal Codes, in K. Sayood (ed.): Lossless Compression Handbook*. San Diego, CA: Academic Press, 2003, pp. 55–78.

[29] P. Fenwick, *Introduction to Computer Data Representation*. Sharjah, U.A.E.: Bentham Science Publishers, 2018.

[30] N. Ferguson, *AES-CBC+ elephant diffuser: A disk encryption algorithm for windows vista*, 2006.

[31] C. Fruhwirth. "Luks1 on-disk format specification." (2018), [Online]. Available: https://mirrors.edge.kernel.org/pub/linux/utils/cryptsetup/LUKS_docs/on-disk-format.pdf (visited on 04/05/2022).

[32] N. Galov. "Linux statistics and facts - linux rock!" (2022), [Online]. Available: https://webtribunal.net/blog/linux-statistics/.

[33] "GCORE." (2021), [Online]. Available: https://www.man7.org/linux/man-pages/man1/gcore.1.html.

[34] M. Goemans. "Shannon's noiseless coding theorem." (2015), [Online]. Available: https://math.mit.edu/~goemans/18310S15/noiseless-coding.pdf.

[35] S. Golomb, "Run-length encodings (corresp.)," *IEEE transactions on information theory*, vol. 12, no. 3, pp. 399–401, 1966.

[36] T. Groß, M. Ahmadova, and T. Müller, "Analyzing android's file-based encryption: Information leakage through unencrypted metadata," in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ser. ARES '19, Canterbury, CA, United Kingdom: Association for Computing Machinery, 2019, ISBN: 9781450371643.

[37] S. Gueron, *A memory encryption engine suitable for general purpose processors*, Cryptology ePrint Archive, Paper 2016/204, 2016.

[38] A. S. Guevara Noubir, "Trusted code execution on untrusted platforms using intel sgx," *Virus bulletin*,

[39] M. A. Halcrow, "ECryptfs: An enterprise-class encrypted filesystem for Linux," in *Proceedings of the 2005 Linux Symposium*, vol. 1, 2005, pp. 201–218.

[40] M. Halcrow. "eCryptfs: A Stacked Cryptographic Filesystem | Linux Journal." (2007), [Online]. Available: https://www.linuxjournal.com/article/9400.

[41] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[42] IEEE. "Security in Storage Working Group." (2006), [Online]. Available: https://web.archive.org/web/20070102043342/http://siswg.org/.

[43] *Implementing dm-verity: Android open source project*. [Online]. Available: https://source.android.com/docs/security/features/verifiedboot/dm-verity.

[44] "Information Security Management," International Organization for Standardization, Geneva, CH, Standard, 2014.

[45] B. Jacob and T. Mudge, "Virtual memory: Issues of implementation," *Computer*, vol. 31, no. 6, pp. 33–43, 1998.

[46] B. Kaliski, "PKCS #5: Password-based cryptography specification version 2.0," RFC Editor, RFC 2898, Sep. 2000. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2898.txt.

[47] L. Khati, N. Mouha, and D. Vergnaud, "Full disk encryption: Bridging theory and practice," in *Topics in Cryptology – CT-RSA 2017*, H. Handschuh, Ed., Cham: Springer International Publishing, 2017, pp. 241–257, ISBN: 978-3-319-52153-4.

[48] A. Klein. "The Cost of Hard Drives Over Time." (2017), [Online]. Available: https://www.backblaze.com/blog/hard-drive-cost-per-gigabyte/.

[49] S. Lai, "Non-volatile memory technologies: The quest for ever lower cost," in *2008 IEEE International Electron Devices Meeting*, IEEE, 2008, pp. 1–6.

[50] S. Levine, *Logical Volume Manager Administration*. Red Hat, Aug. 2020.

[51] L. Martin, "Xts: A mode of aes for encrypting hard disks," *IEEE Security & Privacy*, vol. 8, no. 3, pp. 68–69, 2010.

[52] U. Maurer, "Information-theoretic cryptography," in *Advances in Cryptology — CRYPTO '99*, M. Wiener, Ed., ser. Lecture Notes in Computer Science, vol. 1666, Springer-Verlag, Aug. 1999, pp. 47–64.

[53] D. McGrew and J. Viega, "The galois/counter mode of operation (gcm)," *submission to NIST Modes of Operation Process*, vol. 20, pp. 0278–0070, 2004.

[54] A. Menezes, *Handbook of Applied Cryptography*. Boca Raton: CRC Press, 1997, ISBN: 9780429466335.

[55] R. C. Merkle, "A digital signature based on a conventional encryption function," in *Advances in Cryptology — CRYPTO '87*, C. Pomerance, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1988, pp. 369–378, ISBN: 978-3-540-48184-3.

[56] "Mobile operating system market share worldwide." (2022), [Online]. Available: https://gs.statcounter.com/os-market-share/mobile/worldwide.

[57] A. Moffat and L. Stuiver, *Information Retrieval*, vol. 3, no. 1, pp. 25–47, 2000.

[58] S. Mofrad, F. Zhang, S. Lu, and W. Shi, "A comparison study of intel sgx and amd memory encryption technology," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '18, Los Angeles, California: Association for Computing Machinery, 2018, ISBN: 9781450365000.

[59] K. Mowery, S. Keelveedhi, and H. Shacham, "Are AES x86 cache timing attacks still feasible?" In *Proceedings of the 2012 ACM Workshop on Cloud Computing Security Workshop*, ser. CCSW '12, Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 19–24, ISBN: 9781450316651.

[60] S. Mueller and M. Vasut. "Linux kernel crypto API." (2022), [Online]. Available: https://www.kernel.org/doc/html/latest/crypto/index.html.

[61] A. Niemi and J. Teuhola, "Interpolative coding as an alternative to arithmetic coding in bi-level image compression," in *SCC 2015; 10th International ITG Conference on Systems, Communications and Coding*, 2015, pp. 1–6.

[62] A. Oprea and M. K. Reiter, "Integrity checking in cryptographic file systems with constant trusted storage," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, ser. SS'07, Boston, MA: USENIX Association, 2007.

[63] A. Oprea, M. K. Reiter, and K. Yang, "Space-efficient block storage integrity," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2005, San Diego, California, USA*, The Internet Society, 2005.

[64] C. Paar and J. Pelzl, *Understanding cryptography: a textbook for students and practitioners*. Springer Science & Business Media, 2009, p. 7.

[65] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis and evolution of journaling file systems.," in *USENIX Annual Technical Conference, General Track*, vol. 194, 2005, pp. 196–215.

[66] A. R. Rahiman and P. Sumari, "Solid state disk: A new storage device for video storage server," in *2008 International Symposium on Information Technology*, IEEE, vol. 4, 2008, pp. 1–8.

[67] P. Rogaway, "Efficient instantiations of tweakable blockciphers and refinements to modes ocb and pmac," in *Advances in Cryptology - ASIACRYPT 2004*, P. J. Lee, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 16–31, ISBN: 978-3-540-30539-2.

[68] T. Sammes and B. Jenkinson, "The new technology file system," *Forensic Computing*, pp. 215–275, 2007.

[69] K. Sayood, *Introduction to data compression*, 5th ed. Morgan Kaufmann, 2017.

[70] B. Schneier, *Applied cryptography*, 2nd ed. John Wiley & Sons, 1996.

[71] B. Schneier, "A self-study course in block-cipher cryptanalysis," *Cryptologia*, vol. 24, no. 1, pp. 18–33, 2000.

[72] C. E. Shannon, "A mathematical theory of communication," *Bell Systems Technical Journal*, vol. 27, pp. 379–423, 1948.

[73] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts, 10e Abridged Print Companion*. John Wiley & Sons, 2018.

[74] D. A. Solomon and H. Custer, *Inside Windows NT*. Microsoft Press Redmond, 1998, vol. 2.

[75] F.-X. Standaert, "Introduction to side-channel attacks," in *Secure Integrated Circuits and Systems*, I. M. Verbauwhede, Ed. Boston, MA: Springer US, 2010, pp. 27–42, ISBN: 978-0-387-71829-3.

[76] B. Tao and H. Wu, "Improving the biclique cryptanalysis of AES," in *Information Security and Privacy*, E. Foo and D. Stebila, Eds., Cham: Springer International Publishing, 2015, pp. 39–56, ISBN: 978-3-319-19962-7.

[77] P. Tecchio, F. Ardente, M. Marwede, C. Clemm, G. Dimitrova, and F. Mathieux, "Analysis of material efficiency aspects of personal computers product group," *Luxembourgh. doi*, vol. 10, p. 89 220, 2018.

[78] J. Teuhola, "Interpolative coding of integer sequences supporting log-time random access," *Information Processing & Management*, vol. 47, no. 5, pp. 742–761, 2011, Managing and Mining Multilingual Documents, ISSN: 0306-4573.

[79] The kernel development community. "Dm-linear." (2022), [Online]. Available: https://docs.kernel.org/admin-guide/device-mapper/linear.html.

[80] The kernel development community. "Dm-zero." (2022), [Online]. Available: https://docs.kernel.org/admin-guide/device-mapper/zero.html.

[81] S. C. Tweedie *et al.*, "Journaling the linux ext2fs filesystem," in *The Fourth Annual Linux Expo*, Durham, North Carolina, 1998.

[82] A. Wagner and M. Broz. "DM-crypt README." (2020), [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMCrypt (visited on 03/21/2022).

[83] B. Ward, *How Linux works: What every superuser should know*. No Starch Press, 2021.

[84] "World map of encryption laws and policies." (), [Online]. Available: https://www.gp-digital.org/world-map-of-encryption/.

[85] E. Zadok, I. Badulescu, and A. Shender, "Cryptfs: A stackable vnode level encryption file system," Technical Report CUCS-021-98, Computer Science Department, Columbia University, Tech. Rep., 1998.

[86] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977, ISSN: 1557-9654.

[87] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.