New Methods in Hard Disk Encryption

Clemens Fruhwirth <clemens@endorphin.org>
Institute for Computer Languages
Theory and Logic Group
Vienna University of Technology

July 18, 2005

Abstract

This work investigates the state of the art in hard disk cryptography. As the choice of the cipher mode is essential for the security of hard disk data, we discuss the recent cipher mode developments at two standardisation bodies, NIST and IEEE. It is a necessity to consider new developments, as the most common cipher mode – namely CBC – has many security problems. This work devotes a chapter to the analysis of CBC weaknesses.

Next to others, the main contributions of this work are (1) efficient algorithms for series of multiplications in a finite field (Galois Field), (2) analysis of the security of password-based cryptography with respect to low entropy attacks and (3) a design template for secure key management, namely TKS1. For the latter, it is assumed that key management has to be done on regular user hardware in the absence of any special security hardware like key tokens. We solve the problems arising from magnetic storage by introducing a method called anti-forensic information splitter.

This work is complemented by the presentation of a system implementing a variant of TKS1. It is called LUKS and it was developed and implemented by the author of this work.

Contents

List of Figures

List of Tables

Listings

Preface

I got involved with hard disk cryptography for the first time in the year 2000. This was much too early, since the regular desktop processors were not able to fulfil my performance expectations. Two years later I repeated my attempt to encrypt my data by default, this time with loop-AES, a high performance encryption package for Linux. But due to its unrefined coding style, next to the author's dislike for cooperation, the code was never included into any mainstream Linux project.

I have always been a fan of integration. For my personal projects, I could say, I wanted my work to be part of something bigger and to be useful to the community. But to be honest, I cooperate with bigger projects like Linux distributions or the Linux kernel primarily, because it takes maintenance work away from me. Otherwise, I would have to reintegrate my work with all components when new versions are released. Therefore, I started to put my efforts into another crypto project, which was more likely to be included into mainstream Linux, named "kerneli".

With the advent of Linux 2.6 in 2003 and after having gained experience in how to lobby my code into the Linux kernel, cryptoloop was included into the new release, and the Linux users received a hard disk encryption ready kernel for the first time. cryptoloop was my port based on the loop back device driver from a former kerneli component. But the loop back driver had many unsolved problems attached, and half a year later Christophe Saout presented his superior dm-crypt implementation that was based on a new block device layer called device-mapper. Recognising this, I readjusted my road map and concentrated my efforts on dm-crypt.

In the first quarter of 2004, Jari Ruusu, the author of loop-AES, implemented the water marking attack against the CBC on-disk format both – cryptoloop and dm-crypt – were using. The attack was not taken seriously, especially not by me, as Jari Ruusu had no good reputation and was known to spread more confusion than facts. After new threat models had shown that this attack can be relevant in some situations, I invented ESSIV to remedy the problem. Unfortunately, most Linux users were not well educated with respect to cryptography and were confused from the mixture of correct and unobjective claims Ruusu was still posting to the Linux mailing list.

I started to write a full-disclosure document on all known problems my implementation had. The primary intention was to provide information to the Linux users. But this undertaking lead me much farer. I stumbled consecutively into the world of SISWG and their alternative cipher modes. Stunned by the variety of flaws I collected for CBC, I was eager to start using the new SISWG modes. Until now, I have submitted many patches to the Linux

developers for implementing my favourite of the SISWG modes, LRW-AES. Unfortunately, the Linux kernel virtual memory subsystem is a pure disaster. I refused to create an unaesthetic workaround for its limitations, as my fellow developers requested, and so I stopped my development. Unless this problems are cleared up, there will be no working LRW implementation any time soon for Linux. In fact, there will be no other hard disk solution operating with a SISWG cipher mode any time soon, as my implementation efforts were the only attempt (known to me) to implement them. So, the user has no other option than to use CBC and its probably secure (or not) variants that are included with loop-AES and TrueCrypt.

Probably more important than cipher modes is password management. A big step towards tighter security in this field is achieved with my own software I baptised LUKS. This paper presents the theoretic foundations of LUKS in Chapter 5, "Password Management", and gives a tour of its concrete implementation in Chapter 6.

The following work founds on three years of practical experience with usage and programming of encryption systems. I implemented most of the advised solutions on my own, and some of them are already available with default installations.

Contributions of this work

1. mathematical primer for Galois Field algorithms,
2. development of specialised algorithms for multiplicative sequences in Galois Fields,
3. analysis of NIST cipher modes with respect to applicability for hard disk encryption,
4. presentation of the Anti-forensic Information Splitter,
5. development and analysis of a technological growth model for quantifying PBKDF2 security,
6. Template Key Setup 1 (TKS1), a guideline for designing secure key management systems,
7. case study of a hard disk encryption system incorporating TKS1, namely Linux Unified Key Setup (LUKS).

Acknowledgements

I would like to thank Christophe Saout for his rock-stable dm-crypt implementation, Herbert Valerio Riedel for being forthcoming in my early kerneli efforts, Ernst Molitor, Arno Wagner, James Hughes, Pascal Brisset, Grzegorz Kulewski, Ulrich Kuehn, and Adam J. Richter for reviewing the full-disclosure document of CBC, which is predecessor of Chapter 4, the anonymous commentator for pointing out an improvement concerning the last block of anti-forensic storage and Laszlo Hars for correcting the false hypothesis of independent probabilities in Chapter 4. Thanks to Peter Wilson, who created many useful LaTeX

packages. Without his work, this document would not look as nice. Also, I would like to thank all users and testers that provided me with feedback for LUKS and all my other code.

Finally, I would like to thank my thesis adviser Gernot Salzer. His contribution to the quality of this work is significant due to his highly detailed review of my work.

Chapter 1

Introduction

Data encryption has been used for individual precious documents in the past. With the advent of more powerful desktop processors in the last decade, the data throughput of ciphers surpassed that of hard disks. Hence, encryption is no longer a bottle neck, and regular users become more interested in the topic of hard disk encryption.

Modern operating systems that utilise virtual memory swapping and temporary files make tracking and confining sensitive data hard. To relieve the user of controlling every program's swapping behaviour precisely, the encryption of the hard disk as a whole is a comfortable option. This paper is not about the encryption of a single file, but about the encryption of the whole hard disk in a transparent way. Usually, the file system rests on a virtual block device, which is encrypted in a way transparent to the file system and its users.

Before we discuss several cipher designs, we give a primer of the ingredients used in cryptography. This includes Galois Field theory and algorithms for its use in silicon. Readers familiar with these techniques are invited to skip Chapter 2.

A hard disk encryption designer is flooded with choices for cipher settings. The underlying cipher choice is more or less clear: AES is the dominant and most analysed cipher at the time. But when it comes to cipher modes, there is no such simple and clear decision. The NIST consideration list [NIS05] for cipher modes counts 14 cipher modes suitable for encryption. The Security in Storage Working Group, SISWG, has been founded by IEEE to find and standardise a set of cipher modes especially suited for hard disk encryption. In Chapter 3, we will take a tour of the traditional cipher modes, and the new modes proposed by NIST and SISWG.

In current hard disk encryption designs, CBC is the most used cipher mode. Unfortunately, one has to say. The list of problems associated with CBC is that long that this paper devotes the entire Chapter 4 to them. These problems are not new, but CBC is used mostly because of the lack of alternatives.

The other half of this work is devoted to password management. A feature missing from the most common designs is the ability to change passwords. All cipher blocks on the encrypted partitions are keyed with a single user password, and a password change would require the re-encryption of the whole disk. Of course this is not a desired property. By introducing key hierarchies, a chain of keys can be created to remedy the problem.

A key hierarchy requires additional storage on disk. Care has to be taken that this storage can be purged when needed. For instance, when the user decides to delete a password, the respective key in the key storage has to be destroyed securely. This is not as easy as it may seem. The firmware of a modern hard disk is programmed to preserve data. When it detects that a sector is about to become unreadable by an excessive read error rate, the firmware secures the content to a reserved disk area. The original sector is then unaccessible through the regular disk interface, so the deletion of the original content is infeasible. We will introduce a method to artificially inflate information, thus reducing the statistical chance, that the information ends up in a backup disk section. We will call this technique anti-forensic information splitting.

S-Keys, USB sticks or special key tokens can be used to carry additional key material. But most users do not want to invest extra time or money in buying such products. Thus, most encryption solutions have to rely solely on the password as key information. The problem is that user passwords are usually short, too short to contain enough entropy to match the length of the keys derived from them. By iterative hashing, the lack of entropy can be overcome, and the feasibility of dictionary attacks due to entropy weak passwords can be partially offset. As concrete solution, PBKDF2 is considered as method for password deduction.

As case study for a well-designed hard disk encryption system we will have a look at LUKS. The design of LUKS includes many concepts introduced in this work, and was solely developed by the author of this work. LUKS is short for "Linux Unified Key Setup" and tries to replace home-grown hard disk solutions with a secure standard. LUKS is not a theoretical toothless "paper tiger", but a real stable project, which can downloaded and used in production environments. Chapter 6 gives a tour of this system.

Chapter 2

Design ingredients

The subjects presented in this chapter are common buildings blocks for cryptography. Readers familiar with these concepts are invited to skip this chapter.

First, we define the three commonly used data representations, then we investigate Galois Fields along with algorithms to handle them in silicon. The chapter is finished by a discussion of Gray Codes.

2.1 The many faces of n

Data structures used in computers have a limited numeric domain, for two reasons: first, memory is limited by design, second, dynamic data structures require a substantial amount of overhead processing too costly for cryptography.

To manipulate data structures, we need operators. Operators are always tied to a certain data structure even if it appears that some operators exist independently from a data structure. For instance, the addition operator + defined over $\mathbb{N}$ shares many properties with the + operator defined over $\mathbb{Z}_2$ – associativity, commutativity – however + in $\mathbb{Z}_2$ is self inverse, that is addition is equal to subtraction in $\mathbb{Z}_2$. This is not true for $\mathbb{N}$.

In cryptography, we encounter integer, binary and polynomial operations, and each set of operations has its own interpretation of data, hence its own data structure. The off-the-shelf definitions for integer and polynomial operations are not usable in algorithms operating in silicon. The reason is that these operations assume that the underlying mathematical structure is infinitely large. There is little hope for finding a way to represent an infinitely large mathematical structure in a finite data structure.

Therefore, we need to refine these operations to work on structures with a finite size. Integer operations can be refined quickly: we assume all integer operations to be carried out modulo p, which possesses the form $p = 2^b$, where b stands for the numbers of bits used in the representation. This refinement converts operators defined over $\mathbb{N}$ to operators defined over $\mathbb{Z}_p$. Whenever we refer to an integer operation in this work, we actually mean its $\mathbb{Z}_p$ refinement.

With a bit of care, the same trick works for polynomial operations, but before we discuss this in the next section, we would like to define bijective mappings between the domain of integers ($\mathbb{Z}_p$), binary numbers (vector over $\mathbb{Z}_2$) and polynomials (polynomial ring over $\mathbb{Z}_2$).

If n is an integer, that is $n \in \mathbb{Z}_{2^b}$, there is exactly one element in the domain of vectors over $\mathbb{Z}_2$ associated with n. $(n_{b-1}, n_{b-2}, \ldots n_1, n_0)$ is the vector of bits, where

$$n = \sum_{i=0}^{b-1} n_i \, 2^i$$

holds true. We call this vector a binary number. This definition gives a bijective mapping between integer numbers and binary numbers. We define n_0 to be the least significant bit, while n_{b-1} is the most significant bit. We use LSB(n) to refer to n_0, and MSB(n) to refer to n_{b-1}.

We will make heavy use of Galois Fields later. Galois Field operators are defined utilising polynomial operations. As binary data structures are the native structures found in computers, we need an interpretation of a binary vector as a polynomial. We define that for the vector of bits $(n_{b-1}, n_{b-2}, \ldots n_1, n_0)$, there is an associated polynomial over the indeterminate x possessing the form

$$n_{b-1} \, x^{b-1} + n_{b-2} \, x^{b-2} + \cdots + n_1 \, x \, + n_0$$

Notice that we have defined a bijective mapping between the integer domain and domain of polynomials, by defining two bijective mappings to the intermediate domain of binary vectors. Hence, all three representations can be used completely interchangeably, as each relationship is bijective. All three representations contain the same amount of information. We will use poly(n) to denote a function that is the map from the ring of integers to the ring of polynomials, as defined by the two bijective mappings above.

Sometimes, only a subset of bits is needed. We define a bit slice operation on n, that is represented by $n\langle k, i\rangle$, $k < i$, and is defined as the integer m associated with the bits $\{m_j\}$, where

$$m_j = \begin{cases} n_j & k \leq j \leq i \\ 0 & \text{else} \end{cases}$$

An equal definition is given by

$$n\langle k, i\rangle = \sum_{j=k}^{i} n_j \, 2^j$$

2.2 Galois Field arithmetic

Before we explain the reasons for using Galois Fields, we first have to recapitulate, what the benefits are of carrying out operations in a field, and why it is desirable to choose a field as the underlying structure. Therefore, we restate the field properties.

A field is an algebraic structure $\langle A, \circ, \star\rangle$ defined over a set A and two operations. In a field, the operations $\circ$ and $\star$ have the properties of

closure: $A \times A \to A$, that means: all results of both operators are in A.

associativity: $(a \circ b) \circ c = a \circ (b \circ c)$, and $(a \star b) \star c = a \star (b \star c)$

commutativity: $a \circ b = b \circ a$, as well as $a \star b = b \star a$

Furthermore, the $\star$ operation is distributive over $\circ$.

distributivity: $(a \circ b) \star c = (a \star c) \circ (b \star c)$

For both operations, there is a neutral element ($\underline{0}$ for $\circ$, $\underline{1}$ for $\star$).

Neutral element: A neutral element with respect to an operation will have no effect on any arbitrary element, if applied with this operation. More formally,

$$\exists\, \underline{0} \in A : \forall a \in A : a \circ \underline{0} = a \tag{2.1}$$
$$\exists\, \underline{1} \in A : \forall a \in A : a \star \underline{1} = a \tag{2.2}$$

Inverse elements: Both, $\circ$ and $\star$ have neutral elements, and all elements (except the neutral of $\circ$) have inverse elements for $\circ$ and $\star$.

$$\forall a \in A : \exists b \in A : a \circ b = \underline{0} \tag{2.3}$$
$$\forall a \in A \setminus \{\underline{0}\} : \exists b \in A : a \star b = \underline{1} \tag{2.4}$$

It is important to cryptography to have inverse elements, because decryption works mostly by reverting all encryption operations, that is applying all inverse elements. The two operations $\circ$ and $\star$ are usually similar to addition and multiplication. The rational numbers $\mathbb{Q}$ form a field with respect to those operations, and without knowing, most people use the field $\langle \mathbb{Q}, +, \times \rangle$ for their daily business.

Unfortunately, the rational numbers cannot be used for computers. The cardinality of $\mathbb{Q}$ is infinite. Hence, it is impossible to use the rational numbers as a base set for a field in a computer implementation, where any representation is limited to a fixed precision. Fields with a finite number of elements are needed here.

We assume the following statement without proof:

Proposition 1 *If p is prime, then the residual classes of p form a field.*

The insight of this statement can be demonstrated, when we investigate a subset $\mathbb{Z}_p$ of the set of natural numbers $\mathbb{N}$, $\{0, 1, \ldots, p-1\}$. We define $\underset{\sim}{+}$ to be the regular integer addition, but carried out modulo p to achieve closure. The same is done for $\underset{\sim}{\times}$.

$$a \underset{\sim}{+} b = a + b \mod p$$

The subset $\mathbb{Z}_p$ in conjunction with these two operations forms a field $\langle \mathbb{Z}_p, \underset{\sim}{+}, \underset{\sim}{\times} \rangle$. This field construction works only for primes. The co-domains used in computer science are bit oriented and therefore have sizes of the form 2^η. This is no prime for any $\eta > 1$, and hence integer operations modulo 2^η do not form a field. Another approach is needed.

Proposition 2 *If $\mathbb{F}$ is a field, the set $\mathbb{F}[x]$ of polynomials over the indeterminate x with coefficients out of $\mathbb{F}$ forms a field with respect to polynomial addition and polynomial multiplication.*

If $\mathbb{F}$ is a finite field, then an infinite field is obtained by the construction method of Proposition 2, as the definition does not limit the order of the polynomial in any way. What is needed is a reduction element that limits the resulting polynomial in a similar way as a prime does for integers. An irreducible polynomial is for polynomials what a primes is for integers. Primes and irreducible polynomials are elements of a ring that cannot be constructed with the help of the multiplication operation and any combination of other ring elements. This at the same time implies that a reduction with such a polynomial will always result in an algebraic structure with uniquely defined multiplicative inverse for all its elements, namely a field.

Definition 1 *A Galois Field is a finite field defined over an arbitrary power of a prime, p^η. The notation to capture a Galois Field with cardinality p^η over the indeterminate x and a reduction polynomial r is $GF(p^\eta)|_r[x]$.*

All elements of $\mathrm{GF}(p^\eta)$ can be represented as polynomials with a maximum order $n-1$. For a field $\mathrm{GF}(p^\eta)$ to be unambiguously defined, it must be associated with an irreducible polynomial of order n, which is also known as the reduction polynomial. Given two elements $a, b \in \mathrm{GF}(p^\eta)|_r[x]$, the field operations $\oplus$ and $\otimes$ are defined as the respective polynomial operation $+$ and $\times$ carried out under mod r. Hence,

$$a \oplus b = a + b \mod r$$

$$a \otimes b = a \times b \mod r$$

The polynomial addition modulo r is equivalent to an unbounded polynomial addition. This can be verified by looking at

$$(a_{\eta-1}x^{\eta-1} + \ldots a_1 x + a_0) \oplus (b_{\eta-1}x^{\eta-1} + \ldots b_1 x + b_0) \mod (r_\eta x^\eta + \ldots r_1 x + r_0) \quad (2.5)$$

which can be reordered as,

$$(a_{\eta-1} + b_{\eta-1})x^{\eta-1} + (a_{\eta-2} + b_{\eta-2})x^{\eta-2} + \ldots + (a_1 + b_1)x + (a_0 + b_0) \mod (r_\eta x^\eta + \ldots r_1 x + r_0) \quad (2.6)$$

The order of the result of this polynomial addition is $\max\big(\mathrm{ord}(a), \mathrm{ord}(b)\big)$. If both, a and b, have an order less than n, there is no reduction of the order via the modulo operation. Hence, modulo bounded polynomial addition is equal to unbounded polynomial addition.

As we will see in the next paragraphs, polynomial addition in $\mathrm{GF}(2^\eta)$ can be carried out with a single XOR operation[1]. The field's multiplication operation is not as easy to implement without further refinements.

$$c = a \otimes b \quad (2.7)$$

is defined as the polynomial multiplication,

$$c = (a_{\eta-1}x^{\eta-1} + \ldots a_1 x + a_0) \times (b_{\eta-1}x^{\eta-1} + \ldots b_1 x + b_0) \mod (r_\eta x^\eta + \ldots r_1 x + r_0) \quad (2.8)$$

[1] This is the reason why the symbols for the field operation $\oplus$ and the XOR operation are used as synonyms in literature. Many computer languages use $\hat{}$ as bit-wise XOR operator.

Compared to addition, a polynomial multiplication is more time consuming, not just for pen and paper but also for silicon. What is additionally unpleasant for silicon – when computing the unrefined equation (2.8) – is first, the space for holding coefficients has to be extended to $2n$, and second, the final polynomial division imposed by the modulo operation is non-trivial and time consuming. Fortunately, there are a few tricks that can save a lot of time when carrying out Galois Field multiplications.

The first step towards better computability is to utilise the law of distributivity to split the multiplication (2.7) into a regular pattern. We start by splitting off the coefficient of the lowest order and extracting x from the remaining rest polynomial of b.

$$\begin{aligned} a \otimes b &= a \otimes (b_{\eta-1}x^{\eta-1} + \cdots + b_1 x + b_0) \\ &= a \otimes (b_{\eta-1}x^{\eta-1} + \cdots + b_1 x) \oplus b_0\, a \\ &= \underbrace{a \otimes (b_{\eta-1}x^{\eta-2} + \cdots + b_1)}_{\text{remaining}}\, x \oplus b_0\, a \end{aligned}$$

The process is repeated for the remaining under-braced polynomial, until the following form is reached:

$$((\ldots(\overbrace{(0 \oplus b_{\eta-1}\, a)x}^{\text{most inner}} \oplus b_{\eta-2}\, a)\ldots)x \oplus b_2\, a)x \oplus b_1\, a)x \oplus b_0\, a$$

As you can see the order of x increases steadily from the most inner parenthesis to the most outer parenthesis. The pattern of this expression can be refined to a recursive definition. A recursion is a form of iterative processing, which is a well suited implementation. To avoid confusion with coefficient indices, we use a superscript instead of a subscript to denote the recursion index, and to remind us that this is not a power, we use parenthesis.

$$\begin{aligned} c^{(0)} &= 0 \\ c^{(i)} &= c^{(i-1)}\, x \oplus b_{\eta-i}\, a \end{aligned} \tag{2.9}$$

The expression $c^{(\eta)} \bmod r$ is equal to (2.8). What is unsolved until now, is the explosion of the polynomial order and the required division. This can be solved by using the following modulo laws

$$a + b \mod r = (a \mod r + b \mod r) \mod r$$

The left side of this law is equal to the definition of the $\oplus$ operation, while the right side is equal to $(a \mod r \oplus b \mod r)$. After applying this to (2.9), we obtain

$$c^{(i)} = \left(c^{(i-1)}\, x \mod r\right) \oplus (b_{\eta-i}\, a \mod r) \tag{2.10}$$

There is little to worry about the order of the last term $b_{\eta-i}\, a$ as this is a scalar multiplication having no effect on the order. Remember r is a polynomial, and it is used for reducing the order of a polynomial in the first place. The term $c^{(i-1)}x \mod r$ from (2.10) is the simplest case, where a modulo operation might happen. Before we have a look at the modulo mechanics at work here, we remember the division definition, that also defines modulo results. As given in [Mes82],

Proposition 3 *If s and r are two polynomials over a field, then there are two unique polynomials p and q, where $q < r$, so that $s = p\,r + q$.*

r is the reduction polynomial and q is the polynomial remainder we are interested in. Therefore, we transform the equation to give q as a function of s.

$$q = s - p\,r$$

Assume s to be the result of a polynomial multiplication, $a \otimes x$, that needs to undergo a modulo reduction. We fill in $s = a \otimes x$ and obtain,

$$\begin{aligned} q &= (a_{\eta-1}x^{\eta-1} + a_{\eta-2}x^{\eta-2} + \cdots + a_1 x + a_0)\,x - p\,r \\ q &= (a_{\eta-1}x^{\eta} + a_{\eta-2}x^{\eta-1} + \cdots + a_1 x^2 + a_0 x) - p\,r \end{aligned} \tag{2.11}$$

For polynomials, the requirement $q < r$ from the division algorithm's definition implies that $\mathrm{ord}(q) < \mathrm{ord}(r)$. As $\mathrm{ord}(r) = n$, the order of q has to be smaller than n. Without the subtraction, this is violated in the last equation, as there is a term $a_{\eta-1}x^{\eta}$. Therefore, we have to use the subtraction to make the coefficient for x^{η} vanish. This is obtained by

$$p = \frac{a_{\eta-1}}{r_{\eta}} \tag{2.12}$$

By defining p this way, it ensures that the coefficients for x^{η}, $a_{\eta-1}$ and r_{η}, match in the subtraction. Hence, the subtraction will cause the coefficient of x^{η} to disappear, thus the result will be a polynomial with an order of maximum $n-1$ and thus a valid element of the finite field. Equation (2.12) also implies that p degenerates into an element of the underlying field. To sum up our findings,

$$a \otimes x = (a_{\eta-1}x^{\eta} + a_{\eta-2}x^{\eta-1} + \cdots + a_1 x^2 + a_0 x) - \frac{a_{\eta-1}}{r_{\eta}}\,r \tag{2.13}$$

We have silently introduced the subtraction operation in our equations (2.11) and (2.13), which in fact does not exist in a field. The subtraction operation is the inverse operation of addition and can be synthesised by adding the inverse element associated with addition. The same applies to the quotient used in the calculation of p, (2.12), which can be synthesised by the multiplication with the multiplicative inverse element.

In practise, r is usually given as a monic polynomial, requiring $r_{\eta} = 1$, therefore p will be equal to $a_{\eta-1}$ when considering (2.12). Assuming this, a substitution of (2.13) into (2.10) yields

$$c^{(i)} \;=\; \left(c_{\eta-2}^{(i-1)}x^{\eta} + c_{\eta-3}^{(i-1)}x^{\eta-1} + \cdots + c_0^{(i-1)}x\right) \ominus c_{\eta-1}^{(i-1)}\,r \oplus (b_{\eta-i}\,a) \tag{2.14}$$

Remember that the superscript denotes the recursion index, while the subscript refers to the coefficient index. So, the term $c_{\eta-1}^{(i-1)}$ refers to the coefficient of $x^{\eta-1}$ in the $(i-1)^{\text{th}}$ recursion result. We used the symbol $\ominus$ instead of the subtraction sign to emphasise that this is a field operation. Notice that this equation does not include any mod operations and therefore, it is the well suited for the algorithms we develop in the next section.

2.3 Algorithms for GF(2^η)

The most common Galois Fields, you encounter in this work, are GF(2), GF(2^8), and GF(2^{128}). The Galois Field GF(2) corresponds to the field, which is formed by the residue classes $\mathbb{Z}_2$. The results for the field operations $\oplus$ and $\oplus$ are easily storable in a lookup table for field sizes up to 2^8. But for GF(2^{128}), we need algorithms to calculate the results.

As most Galois Fields in this paper are defined over the prime 2, it is useful to investigate its speciality for hardware. Equation (2.6) shows, a polynomial addition can be carried out by adding the coefficients of all powers of x directly. A field GF(2^η) has η coefficients out of GF(2). An addition in GF(2) is equivalent to the exclusive-or operation found in Boolean logic, more commonly denoted as XOR in computer sciences.

XOR	0	1
0	0	1
1	1	0

If the coefficients are stored in a vector and the coefficients are bits, the polynomial becomes presentable by a bit vector. In many architectures we find instructions that can carry out a XOR operation on two bit vectors matching the i^{th} element of one vector with the i^{th} element in the other vector. With the help of these instructions, it is possible to do a Galois Field addition with a single instruction. In this section, we assume that all polynomials are over the field $\mathbb{Z}_2$ and are represented by a bit vector.

Polynomial multiplication has been well prepared for computer implementations by the refinement to a recursive function, that also includes the reduction polynomial. For $p = 2$, the reduction equation (2.13) becomes even simpler as the coefficient of $x_{\eta-1}$ is either 0 or 1, that is the term $x^{\eta-1}$ is present or not. In case of $a_{\eta-1} = 0$, no subtraction has to be done at all, as a multiplication with x will not result in a term x^η. If $a_{\eta-1} = 1$, the reduction polynomial is subtracted exactly once. So the modulo reduction can be implemented as a conditional subtraction when $a_{\eta-1} = 1$ and a shift operation. Bit shift operations are available in almost any modern architecture. Subtraction can be replaced by addition, as both operations are interchangeable in GF(2). Convince yourself by looking at the XOR table.

In the following function definition, it is assumed that r is a monic polynomial, so $r_\eta = 1$ at all times, which makes storing this coefficient dispensable. Omitting the coefficient for x^η from r makes r of the order $\eta - 1$, which is convenient, as in this form, it can be stored with the same precision and in the same data structure as a regular $\eta - 1$ order polynomial.

For clarity, we assume for all following function definitions that η is not passed explicitly, but is implicitly available.

Listing 2.1: xtime: multiplication with x and modulo reduction

```
xtime(a):
        if a_{η-1} = 1:
                return (a << 1) ⊕ r
        else
                return a << 1
```

xtime implements $a \otimes x$. Using this function in the recursive definition of f (2.9), we obtain a computable function for the polynomial multiplication.

Listing 2.2: base algorithm, xtime-version

```
c(a, b): return c'(a, b, η)

c'(a, b, i):
        if i = 0:
                return 0

        return xtime(c'(a, b, i-1)) ⊕ b_{η-i} a
```

c is used to start the recursion and c ' does the actual computation. The code of c ' closely resembles (2.9). We have used a simplified version for c ' here. The scalar multiplication $b_{n-i}\, a$ at the return statement needs to be expanded for a real implementation. But $b_{\eta-i}$ is either 1 or 0, so we can replace it with a conditional addition of a.

Listing 2.3: an implementable c'-recursion

```
c'(a, b, i):
        if i = 0:
                return 0

        p = xtime(c'(a, b, i-1))
        if b_{η-i} = 1:
                return p ⊕ a
        else
                return p
```

This listing contains the first implementable Galois Field multiplication code in this work. When this operation is needed infrequently, it is ok to choose this implementation. But many cipher modes in the following chapters use Galois Field multiplications extensively. Usually one of the operands is a fixed polynomial and in that case, there is another technique to save CPU cycles.

The original definition of $c = a \otimes b$ as given in (2.8) can be split up thanks to the law of distributivity.

$$\begin{aligned} c = {} & b_{\eta-1}\big(a \otimes x^{\eta-1}\big) \oplus \\ & b_{n-2}\big(a \otimes x^{\eta-2}\big) \oplus \\ & \cdots \\ & b_1\big(a \otimes x\big) \oplus \\ & b_0\, a \oplus \\ & 0 \end{aligned}$$

Also, this is equation features a regular structure, which we can put into an algorithm easily. A lookup table can store terms of the form $a \otimes x^i$, and this algorithm can add the corresponding element when b_i is 1. Given a GF(2^η), it is sufficient to store η terms of the form $a \otimes x^i$. We can calculate the lookup table via the xtime function.

$$\mathrm{lookupTable}[0] = a \tag{2.15}$$

$$\mathrm{lookupTable}[k] = \mathrm{xtime}(\mathrm{lookupTable}[k-1]) \tag{2.16}$$

The polynomial multiplication can now be programmed as:

Listing 2.4: base algorithm, lookup table version

```
c(lookupTable, b):
        p ← 0
        for i = 0 till η−1 do {
                if b_i = 1:
                        p ← p ⊕ lookupTable[i]
        }
        return r
```

This optimisation is sufficient when the Galois Field multiplication is carried out infrequently and one operand is fixed. But some cipher modes are constructed in a way, that it is likely that related results are needed. This relation can be used to save a lot of processing. Without the following techniques, cipher modes such as LRW-AES would have a serious disadvantage compared to other modes.

2.4 Sequences of multiplications in GF(2^η)

We anticipate a few things from the following chapters, so the reader understands, why it is crucial to investigate these sequences. In LRW-AES, we will see a design that will carry out a Galois Field multiplication with a fixed polynomial a and poly(n), where poly(n) stands for the corresponding polynomial to integer n as defined by the bijective mapping in Section 2.1. LRW-AES uses the result of the GF multiplication in the encryption of the n^{th} block.

Assume we use LRW-AES for hard disk encryption. When the block n is accessed, the probability is high that the block $n+1$ will be accessed too, as hard disk access is rarely carried out isolated. In practise, this is also true for much larger differences such as $n + 255$.[2]

Definition 2 *An arithmetic sequence of a polynomial multiplication is the series* $a \otimes poly(n)$, $a \otimes poly(n+1)$, $a \otimes poly(n+2)$, …, $a \otimes poly(n+k)$, *where* a *is a polynomial,* n *is an integer and* $n+k$ *is an integer addition.*

This section discusses an efficient way for generating such sequences.

When the underlying field has a size of 2^{128} and c from Listing 2.4 is used, then 128 conditional XORs have to be carried out for every single element of the arithmetic sequence. When LRW-AES accesses 256 hard disk blocks, this means 32768 conditional XORs have to be done. At the end of this chapter, we will present an algorithm that can do the same with just 383 XORs. That is almost a hundredth of the original requirement. This ratio is improving, when the sequence of multiplication results becomes longer.

2.4.1 Discussion

To get an idea of how such an algorithm might look like, we investigate an integer sequence 0–15 and the corresponding polynomials. In Table 2.5, the

[2] Linux uses a 4KB page cache structure. An LRW-AES block is 16 bytes wide, hence there are 256 blocks in 4KB, which will be read all at once.

n	$n_3n_2n_1n_0$	$a \otimes (\text{poly}(n))$	$a \otimes (\text{poly}(n+1) \oplus \text{poly}(n))$
0	0000	$a \otimes (0)$	$a \otimes (1)$
1	0001	$a \otimes (1)$	$a \otimes (x+1)$
2	0010	$a \otimes (x)$	$a \otimes (1)$
3	0011	$a \otimes (x+1)$	$a \otimes (x^2+x+1)$
4	0100	$a \otimes (x^2)$	$a \otimes (1)$
5	0101	$a \otimes (x^2+1)$	$a \otimes (x+1)$
6	0110	$a \otimes (x^2+x)$	$a \otimes (1)$
7	0111	$a \otimes (x^2+x+1)$	$a \otimes (x^3+x^2+x+1)$
8	1000	$a \otimes (x^3)$	$a \otimes (1)$
9	1001	$a \otimes (x^3+1)$	$a \otimes (x+1)$
10	1010	$a \otimes (x^3+x)$	$a \otimes (1)$
11	1011	$a \otimes (x^3+x+1)$	$a \otimes (x^2+x+1)$
12	1100	$a \otimes (x^3+x^2)$	$a \otimes (1)$
13	1101	$a \otimes (x^3+x^2+1)$	$a \otimes (x+1)$
14	1110	$a \otimes (x^3+x^2+x)$	$a \otimes (1)$
15	1111	$a \otimes (x^3+x^2+x+1)$	$a \otimes (x^4+x^3+x^2+x+1)$

Table 2.5: The many faces of n

chosen sequence is tabulated. The first thing that comes into mind, when looking at this table is that the surrounding $a \otimes (\dots)$ of the last two columns are superfluous in this presentation. This is correct, and we will later strip them with a trick, but before we do that, we need to grasp what we are actually stripping.

Our algorithm candidates, that have to compute $a \otimes \text{poly}(i)$ from a related $a \otimes \text{poly}(n)$, can utilise the elements of the lookup table to switch individual $a \otimes x^i$ elements on and off. You will see shortly, what is meant by "related".

Have a look at Line 8 of Table 2.5. The result for Line 9 can be generated with the help of a single XOR by adding $a \otimes 1$ to the result of Line 8. Likewise, Line 10 can be generated by adding $a \otimes x$ to the result of Line 8. Line 11 can be computed by adding $a \otimes 1$ to Line 10. In this tiny cutout, the saving is relatively small, but imagine that n not only consist of 4 binary digits but 128. If we use algorithm c, this implies 128 conditional XORs to generate Line 9. But when we have the result of Line 8 at hand, we get along with a single XOR. The same is true for Line 10. Compared to 128 XOR operations, this shortcut has a huge saving potential. Let us investigate the systematic of this shortcut and its prerequisites.

In this sequence, we have not considered lines 0 till 7. For $i < 8$, the differences between lines i and $i+8$ are constantly $a \otimes x^3$. Thus, Line 9 can also be generated with the help of Line 1 and the lookup table element $a \otimes x^3$. Likewise, Line 10 by using Line 2 and so on, until Line 15 is generated from Line 7.

There is another case, where we can use a similar approach. For $i < 4$, the differences between Line i and Line $i+4$ are constantly $a \otimes x^2$. For $i < 2$, we can use Line i to get to Line $i+2$ with the help of $a \otimes x$.

The reader might have already guessed the system behind this. It is possible to use i to generate $i + 2^k$ by the use of $a \otimes x^k$. This is not surprising, as

the bit-wise difference between i and $i + 2^k$ is exactly at position k. This is mirrored in the difference of $a \otimes x^k$ in the multiplication results of $a \otimes \text{poly}(i)$ and $a \otimes \text{poly}(i + 2^k)$.

Before we dive deeper into the subject, we conduct a restriction that will radically simplify our view. Assume a to be fixed to the degenerated polynomial 1. This is the identity element and any multiplication with it can be stripped of our equations. But what actually simplifies our thinking is, that the lookup table of a as defined in (2.16) will degenerate into simple binary masks. The first element of the lookup table has only one non-zero coefficient, namely a_0. Any successive element will be computed by `xtime`, that will shift this coefficient until the coefficient $a_{\eta-1}$. When these polynomials are considered in their binary representation, they will be simple binary masks of the form $0^{\cdot\eta-k}\,1\,0^{\cdot k-1}$, or in other words, a binary vector that is zero except at position k.

With this restriction, Listing 2.4 degenerates into a simply copy algorithm. It scans all bits of b_i and sets the respective bits in the returned polynomial p as `lookupTable`$[i]$ will have only the i-th bit set. This simplifies the bit representations of the integer n and the polynomial $a \otimes \text{poly}(n)$ as both will be equal. So, when we talk about generating $n + 1$ from n we only have to investigate how to change the bit pattern of n to get $n + 1$. Reconsider the case, where we want to generate Line 10 from Line 8. Only the second last bit has to be switched. This is equal to x. So for the development of algorithms, it is sufficient to think of switching bits in n by the means of XOR operations. When we have finished drafting an algorithm, we can lift the restriction for a by replacing the lookup table.

Whenever the bit representations of the integer n and $a \otimes \text{poly}(n)$ give the same results, we should have no problem to immediately come up with a way how to generate $a \otimes \text{poly}(n + 1)$ by using $a \otimes \text{poly}(n)$, because all we have to do is inspect the bit-wise difference of $n + 1$ and n. We cannot use any other operation than XOR to generate $n+1$ from n, as it is the only tool when we switch to a lookup table for a later. So, all we have to do is to reverse engineer the integer addition with the help of XOR operations, when we want to generate $a \otimes \text{poly}(n + 1)$ from $a \otimes \text{poly}(n)$.

When we assume a to be equal to 1, we can strip a of the last two terms and as the rest $\text{poly}(n+1)$ and $\text{poly}(n)$ is only an indicator for another representation of the integers $n+1$ and n, we will omit poly(..) too. Hence, we will speak of generating $n + 1$ from n, when we actually mean generating $a \otimes \text{poly}(n + 1)$ from $a \otimes \text{poly}(n)$.

Integer addition features a carry logic, which we cannot easily reassemble by flipping a single bit. Flipping single bits is all we can do, because in the restricted view on the lookup table, all it contains are binary masks with a single bit set. We either avoid the carry from happening – done in the last few paragraphs – or we simulate the carry by modifying the lookup table. We present two algorithm for both approaches. Both have their advantages and disadvantages. This section will close with a hybrid version of both.

2.4.2 Algorithm for carry prevention

Following the ideas presented in the last section's discussion, we can give a recursive definition for multiplication sequence,

$$\begin{aligned} f_0 &= 0 \\ f_n &= f_{n-2^l} \oplus \mathrm{lookupTable}[l] \qquad \text{where } l = \lfloor \log_2 n \rfloor \end{aligned} \tag{2.17}$$

This recursion pattern is irregular. For the lines 8 to 15 of Table 2.5, the rounded logarithm stays constantly $l = 3$ and causes them to be generated from lines 0 till 7 by the use of x^3. How the whole table is generated with this recursive definition is depicted in Figure 2.6.[3]

If we want to generate n results, but start from a polynomial different from 0, when can simply change f_0 to – for instance – p. Thanks to the recursion p will be added to all elements of f_n. There is only one restriction for p. The $\lceil \log_2 n \rceil$ lower order coefficients (and bits) must be zero, because otherwise it would cause a carry which is not anticipated by this definition.

We formalise this requirement with the `align` function, $\mathtt{align}(p, 0) > n$, when n is the sequence length.

Listing 2.5: alignment determination

```
align(n,b):
        for i ← 0 till η − 1 do:
                if n_i ≠ b:
                        return i
        return η
```

2.4.3 Algorithm for carry simulation

A major disadvantage of the recursive definition (2.17) is that it requires results of previous calculations that might have been neither requested nor wanted.

While the difference for $(n + 1) - n$ is trivially constant in the group of natural numbers mod 2^η, the differences between the polynomials $\mathrm{poly}(n + 1)$ and $\mathrm{poly}(n)$ are not as regular. The reason for this is that subtraction in $\mathrm{GF}(2^\eta)$ is actually a field addition, and further, this operation is in fact equivalent to a XOR operation. XOR does not feature any carry elements in contrast to integer addition.

We need a way to reproduce the carry logic. Therefore, we investigate the pattern that is induced by integer carry between successive elements. In Table 2.7, we find that all differences have the form $0^{:n-k}\,1^{:k}$. Notice that the number of closing ones in the XOR difference is equal to the number of least-significant bits of n plus 1. In other words, it holds true for the index k of $0^{:n-k}\,1^{:k}$ that

$$k = \mathtt{align}(n, 1) + 1$$

We can use a second lookup table to include elements just of that form. Translated into a polynomial, these difference patterns have the form

$$a \otimes \mathrm{poly}(0^{:n-k+1}\,1^{:k+1}) = \bigoplus_{i=0}^{k} a \otimes x^i$$

[3] In fact, we are talking about $a \otimes x^i$ when we replace the lookup table later.

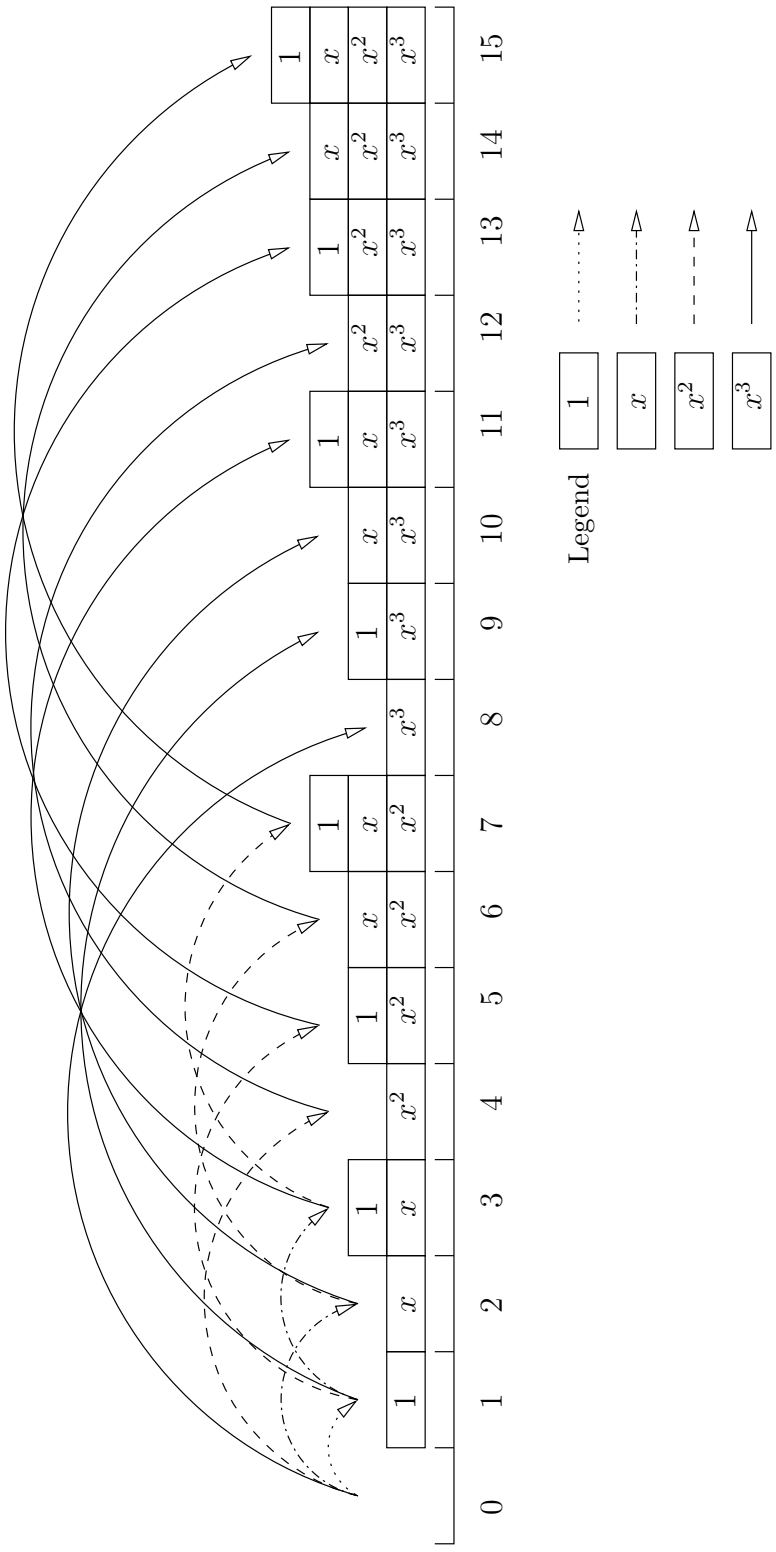


Figure 2.6: Aligned optimise

n	$n_3n_2n_1n_0$	$n \oplus (n+1)$
0	0000	0001
1	0001	0011
2	0010	0001
3	0011	0111
4	0100	0001
5	0101	0011
6	0110	0001
7	0111	1111
8	1000	0001
9	1001	0011
10	1010	0001
11	1011	0111
12	1100	0001
13	1101	0011
14	1110	0001
15	1111	-

Table 2.7: binary difference table

Listing 2.6: Single step via negative logic

```
singleStep(NArray, n, pos):
        NArray[pos] ← NArray[pos-1] ⊕
                          lookupTableNeg[align(n,0)]
```

By using elements of this lookup table, it is possible to compute steps like 3 to 4, or 7 to 8 from Table 2.7 in a single XOR operation. It works by choosing the appropriate carry pattern by determining the numbers of least significant 1 bits. This yields the elegant recursive definition,

$$f_{n+1} = f_n \oplus \mathrm{lookupTableNeg}[\mathrm{align}(n,1)] \tag{2.18}$$

We term this lookup table to utilise negative logic, as most of the elements are used to cancel out already present elements. The most striking example of this is the step from 7 to 8.

The essence of this recursion can be encoded into the `singleStep` function of Listing 2.6.[4] It is assumed that NArray is the array of calculations done so far and has $pos - 1$ computed elements. The element at position $\mathrm{pos} - 1$ contains the result for $a \otimes \mathrm{poly}(n-1)$, and in general, the element at position $\mathrm{pos} - i$ contains the result $a \otimes \mathrm{poly}(n-i)$ (when $\mathrm{pos} - i \leq 0$). The array element NArray[pos] is the target element, that should be filled with the new polynomial multiplication result for $a \otimes \mathrm{poly}(n)$.

The `singleStep` function gives a very compact and implementable algorithm for generating sequences of multiplication results. However, it still depends on the base algorithm c to generate the first result in NArray[pos]. The base algorithm c is still operating with the regular lookup table. To spare the

[4] This algorithm uses the identity $\mathrm{align}(n+1,0) = \mathrm{align}(n,1)$.

need for two lookup tables, the algorithm in Listing 2.4 can be restated to use the negative-logic lookup table as well.

Listing 2.7: Base algorithm for negative logic

```
c(n):
        r ← 0, s ← 0
        for i ← (η − 1) till 0 do {
                if b_i = s:
                        r ← r ⊕ lookupTableNeg[i]
                        s ← s ⊕ 1
        }
        return r
```

What is suboptimal about the `singleStep` function is, that it needs to redetermine the alignment at every call. The expectation for the number of iterations I of the loop in Listing 2.5 is,

$$\mathbb{E}\, I = \sum_{i=1}^{\eta} i\; p(i) = \sum_{i=1}^{\eta} i\; \frac{1}{2^i} = 2 - \frac{\eta + 2}{2^\eta}$$

Furthermore,

$$\lim_{\eta \to \infty} \mathbb{E} I = 2$$

A multiplication according to the c function given in Listing 2.4, requires η bit checks and on average $\eta/2$ conditional XORs. The `singleStep` function does much better with 2 bit checks on average and 1 XOR operation. Although these results are satisfactory, a final optimisation can be made.

2.4.4 A hybrid algorithm

The advantage of avoiding carries is that the algorithm does not have to reselect a polynomial from the lookup table by `align`, as l in (2.17) is constant most of the time. The disadvantage is that in order to utilise the performance gain due to a non-changing lookup polynomial it is necessary to generate a series of results at once. Furthermore, the algorithm needs results of previous calculations, and a proper alignment of them.

The advantage of the carry simulation is that it works without all these prerequisites. It requires only $n - 1$ to be present to generate n. The disadvantage is this algorithm must inspect n at every step. This also implies that $n + 1$ must be calculated via an integer addition, something which does not have to be done for the carry prevention algorithm.

In a hybrid of these two algorithms, we try to get the best of both worlds. We want to be able to start from any given n no matter what alignment it has. We also want to use the speed of the carry prevention algorithm if a large set of results is requested.

Before we can use the carry prevention algorithm, we have to give an implementation for it first. The recursion in (2.17) is using the regular lookup table. With the alternative implementation of c in Listing 2.7, the implementation of (2.17) would be the only component left that requires the regular lookup table. We like to refine this recursion to use the negative lookup table, because so the implementation is not forced to maintain two separate lookup tables.

Equation (2.17) can be rewritten by adjusting the indices of the recursion and changing the lookup table.

$$f_n = f_{2^{l+1}-n} \oplus \text{lookupTableNeg}[l] \tag{2.19}$$

How this recursion pattern unfolds is visualised in Figure 2.8. In contrast to the version with the positive lookup table, the previous sequence results are traversed backwards. This way we use the negative logic to cancel out any unwanted polynomial coefficients.

In (2.19), $l = \lfloor \log_2 n \rfloor$ does not change for every step of n. An algorithm using this equation can rely on the fact that the rounded values $\log_2$ do not change at every step unless $n = 2^i$ for any $i \in \mathbb{N}$. The implementation makes use of this fact by directly controlling i.

Listing 2.9: carry prevention

```
fromAlign(NArray, n, pos, l):
    if l > align(n,0):
        return error, alignment violation

    for i ← 0 till l−1 do:
        poly ← lookupTableNeg[i]
        for k ← 2^i till 2^(i+1) − 1 do:
            NArray[pos+k] ←
                NArray[pos+2^i − k] ⊕ poly
    }
```

The algorithm is called `fromAlign`, because it requires n to be properly aligned. Its parameter[5] l specifies that $2^l - 1$ elements should be generated. Second, there are no $\log_2$ computations in this algorithm. Because of the rounding, that takes place via the ceiling function for l, a loop can be constructed from $n = 2^i$ to 2^{i+1}, where the rounded $\log_2 n$ is guaranteed to stay constant. Hence, we do not need to compute $\log_2 n$ as it is implicitly fixed to the current value of i.

We give an example of an arithmetic multiplication sequence to visualise, how `fromAlign` and `singleStep` can be mixed to produce a hybrid version to take advantage of both of them. Assume that 21 results of the form $a \otimes \text{poly}(n), a \otimes \text{poly}(n+1) \ldots$ are requested, where n is equal to 1101100.

generate from	generate what	method	number of results
-	1101100	base algorithm	1
1101100	1101101 to 1101111	`fromAlign`	3
1101111	1110000	`singleStep`	1
1110000	1110001 to 1111111	`fromAlign`	15
1111111	10000000	`singleStep`	1

The base result $a \otimes \text{poly}(1101100)$ is generated by the base algorithm. Then 3 elements can be generated by `fromAlign` at once, then a `singleStep` call is

[5] A real implementation of this algorithm differs insofar that an additional variable is added to control the precise length of elements, not as l aligned to 2^l. Because adding these checks complicates the control structure considerably, it is left out for pedagogic reasons here.

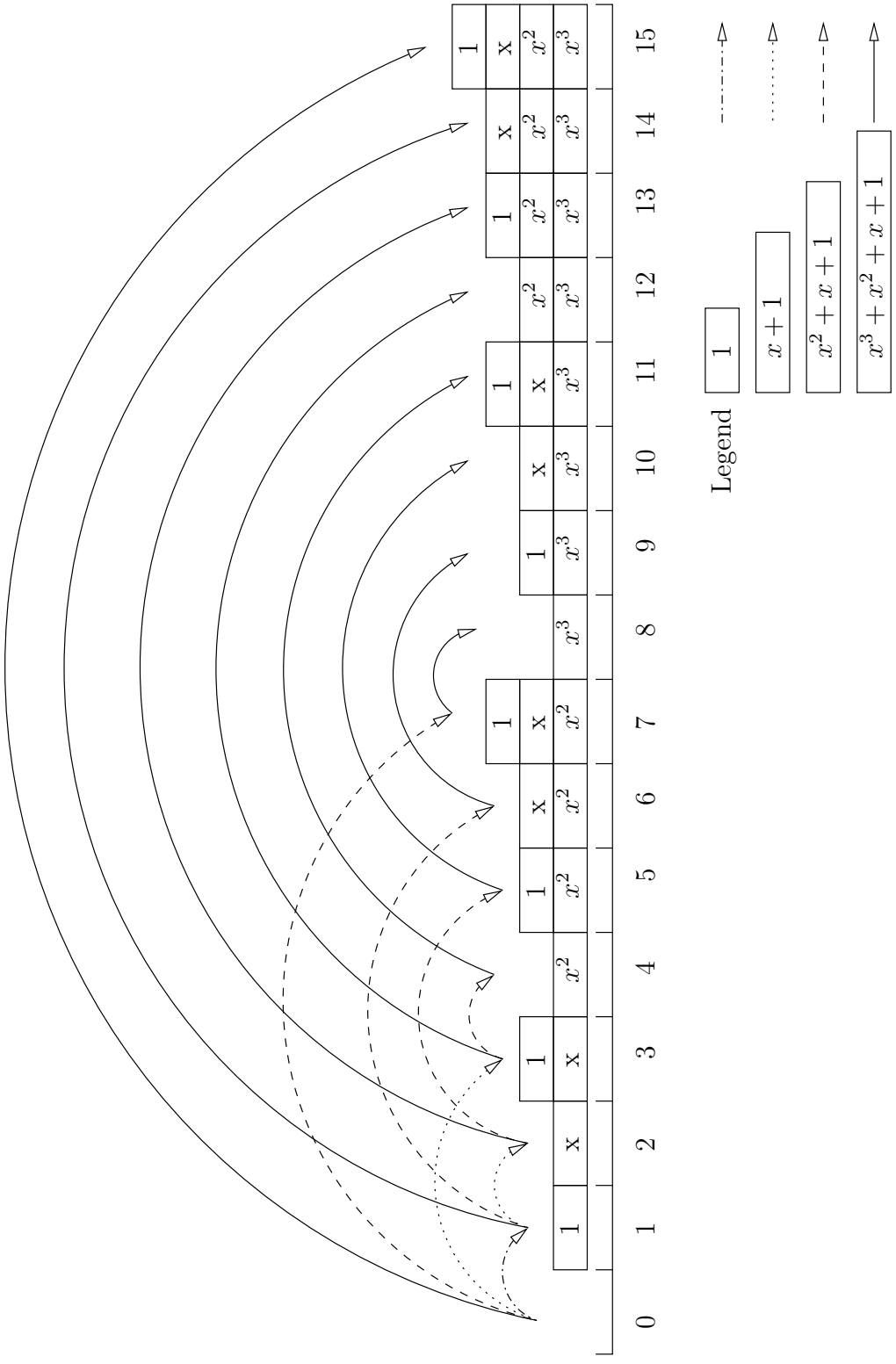


Figure 2.8: Aligned negative logic

Listing 2.10: Multiplication sequence (hybrid)

```
GFMulSeq(NArray, n, length):
2       NArray[0] ← c(n)
        pos ← 1
4       if pos < length:
                return
6
        align ← align(n+pos,0)
8
        if(align != 0):
10              fromAlign(NArray,n+pos, pos, align)
                pos ← pos + 2^align
12
        while(true):
14              if pos < length:
                        return
16
                align ← align(n+pos,1)
18              singleStep(NArray,n+pos,pos)
                pos ← pos + 1
20
                if pos < length:
22                      return

24              align ← align(n+pos,0)
                fromAlign(NArray, n+pos, pos, align)
26              pos ← pos + 2^align − 1
```

necessary. The next 15 elements can be generated by fromAlign, as the last 2^4 bits of 1110000 are zero. At 1111111 a singleStep call is needed to reach 100000000.

The switching process sketched in the table above can be formalised in the algorithm of Listing 2.10. First, a base result is obtained in Line 2. Then a check is made, if sufficient elements have been generated. The check is made regularly at lines 14 and 21. Line 9 determines, if the first step is a fromAlign call (Line 10) or from singleStep (Line 18). The loop's task is to call singleStep and fromAlign alternately, and advance the position pointer correctly. The elements of NArray are aligned so that NArray$[i]$ contains $a \otimes \text{poly}(n+i)$.

This presentation is a compromise between optimisation and comprehension. A real algorithm would keep extra counter variables for $n + pos$ and $length - pos$. Also notice that this algorithm will eventually generate too much results, as the fromAlign cannot be limited precisely in the given implementation.

This section has digged deep into possible optimisations for Galois Field multiplications when they are carried out in silicon. Even more optimisations can be found in my real implementation of these algorithms, for instance, that alignment determination is limited to η checks in total, as it can be assured that the alignment is monotonously increasing.

$n_3n_2n_1n_0$	$g_3g_2g_1g_0$	poly(g)	difference to prev. column
0000	0000	0	–
0001	0001	1	1
0010	0011	$x+1$	x
0011	0010	x	1
0100	0110	x^2+x	x^2
0101	0111	x^2+x+1	1
0110	0101	x^2+1	x
0111	0100	x^2	1
1000	1100	x^3+x^2	x^3
1001	1101	x^3+x^2+1	1
1010	1111	x^3+x^2+x+1	x
1011	1110	x^3+x^2+x	1
1100	1010	x^3+x	x^2
1101	1011	x^3+x+1	1
1110	1001	x^3+1	x
1111	1000	x^3	1

Table 2.11: Gray Code difference table

Before we will dive into cipher modes in the next chapter, we will investigate another method to speed up Galois Field multiplications. The approach is to generate a permutation $\{h_n\}$ of $\mathbb{N}$ that reduces the step from $a \otimes \text{poly}(h_n)$ to $a \otimes \text{poly}(h_{n+1})$ to a single XOR operation.

2.5 Gray Code

Definition 3 *A Gray Code is a sequence h_i, such that*

$$d(h_i, h_{i+1}) = 1 \qquad \forall i > 0$$

where $d(x, y)$ is the Hamming Distance between x and y.

When we use a vector representation of bits, then this definition implies that successive elements of a sequence must differ by exactly one bit. To demonstrate the implications, Table 2.11 contains a permutation of the entries of Table 2.5 ordered as a valid Gray Code.

The advantage of using Gray Codes becomes visible immediately when the last column is considered. The difference between any element has the form x^i, which is much more regular than the differences of Table 2.5.

Gray Codes are not unique and their actual sequence depends on the method of generation. One of the generation methods is reflexive copying. How this methods works is depicted in Figure 2.12. First, a sequence is taken that is known to be a Gray Code. Then it is copied twice, once normally, and the second time reflexive. Then two elements are chosen with the Hamming Distance 1. The first element is prepended to the normal copy, and the second element to the reflexive copy. The obtained sequence is a valid Gray Code.

In the figure, the element prepended to the normal copy is 0 and the other is 1. The initial Gray Code sequence is also {0,1}. These are arbitrary choices.

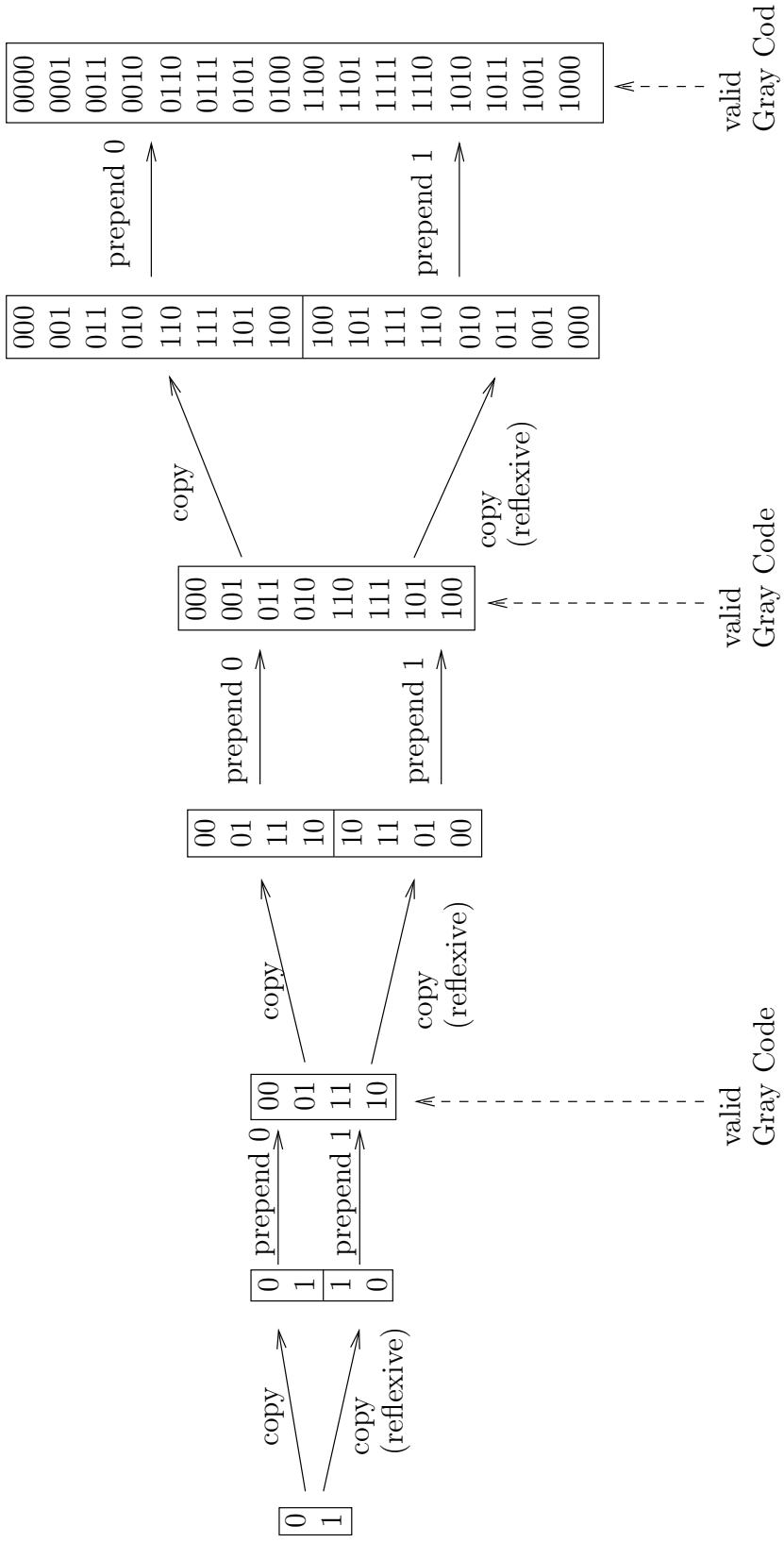


Figure 2.12: Reflexive Gray code

Choosing 1 to be the first element, and 0 to be the second element for the reflexive part would also yield a valid Gray Code. However, there is no gain from considering other permutations, and so we shall concentrate on the method depicted in the figure, which generates the binary reflexive Gray Code[6].

The sequence of natural numbers m_i in binary form can be associated bijectively with their binary reflected Gray Code, $h_i = \text{brgc}(m_i)$. The Gray Code $g = \text{brgc}(n)$ is generated from the binary representation of number n. Given a sequence of bits $n_k n_{k-1} \ldots n_1 n_0$ for the number n, the bits $g_k g_{k-1} \ldots g_1 g_0$ of the reflexive binary code are generated by

$$g_i = n_{i+1} + n_i \mod 2$$

The inverse can be computed by

$$n_i = \sum_{j=i}^{\infty} g_j \mod 2$$

Table 2.11 is generated with these equations. The verification of these equations is left as an exercise to the reader.

[6]Sequence number A014550 in "The On-Line Encyclopedia of Integer Sequences" [AT&T]

Chapter 3

Cipher Modes

Hardly anyone uses a cipher as published. If this is the case, the cipher is said to be used in ECB mode, Electronic Codebook Mode, but this mode does not fully qualify as a cipher mode, because ECB rather denotes the absence of any further design.

Normally, a cipher serves as a building block in an encryption design. The way a cipher is utilised is called cipher mode. The advantage of a cipher mode can be to obfuscate plaintext similarities, to relieve the user of block size restrictions, or provide additional authentication capabilities. These benefits come for a small computational price, and this is the reason why these designs are so common.

This chapter will first have a look at the well known, and well analysed, traditional cipher modes. The second part will examine more recent cipher design, most notably the submissions to the "NIST - Modes Of Operation" standardisation process and the work of the SIS Working Group of IEEE. Despite the minor role of authentication algorithms, they are relevant for some attack models. Thus, we will give the reader a brief overview of what is available with respect to this topic.

Notation

We utilise the regular definition of a cipher. A cipher is an invertible function $\mathcal{E} : \mathbb{P} \times \mathbb{K} \rightarrow \mathbb{C}$. $\mathbb{P}$ is the set of possible plaintexts, $\mathbb{K}$ the set of possible keys, and $\mathbb{C}$ the set of possible ciphertexts. The inverse function with respect to the argument $\mathbb{P}$ is called $\mathcal{D}$, and fulfils

$$P = \mathcal{D}(\mathcal{E}(P, K), K) \qquad \text{for all } P \in \mathbb{P}, \text{ and } K \in \mathbb{K} \tag{3.1}$$

K will be omitted if clear from context. If not, the key will be added as subscript to $\mathcal{E}$, so $\mathcal{E}(P, K) = \mathcal{E}_K(P)$.

The following convention is used to distinguish between block cipher en/decryption operations and cipher mode en/decryption operations: $\mathcal{E}$ and $\mathcal{D}$ denote block cipher operations, while $\mathbf{E}$ and $\mathbf{D}$ represent cipher mode operations. For block size restricted modes, the operands of $\mathbf{E}$ and $\mathbf{D}$ are an integral multiple of those of $\mathcal{E}$ and $\mathcal{D}$. If a cipher mode symbol is used to refer to a specific mode, the mode's name is added as superscript, for instance, $\mathbf{D}^{CBC}$.

We use $c^{:n}$ to refer to a string of length n consisting only of c, for instance the all-zero block $0^{:n}$.

3.1 Traditional cipher modes and their shortcomings

The following cipher modes are fairly simple, and have been in use for a long time. Their ideas serve as building blocks for other more sophisticated cipher designs. If we aim to understand the motivations of these sophisticated designs, we have to have a look at the traditional designs to see, where they succeed and where they fail.

3.1.1 CBC: Cipher Block Chaining

The most common mode might be CBC, which stands for Cipher Block Chaining. This design's main intent is to overcome the shortcoming that in regular ECB mode two identical plaintext blocks encrypt to the same ciphertext, and hence ECB reveals plaintext similarities. CBC creates an interdependency among the cipher blocks by XORing the preceding cipher block into the current block's plaintext. This operation modifies the plaintext and thus modifies the ciphertext. Figure 3.1 is a visualisation of this mode.

The formal definition of CBC is[1]:

$$\begin{aligned} \text{Encryption:} \qquad \mathbf{E}^{CBC} &: \mathbb{P}^n \times \mathbb{K} \to \mathbb{C}^n \qquad (3.2)\\ P &= P_1 \times P_2 \times \cdots \times P_n \\ C_i &= \mathcal{E}_K(P_i \oplus C_{i-1}) \\ \mathbf{E}^{CBC}(P) &= C_1 \times C_2 \times \cdots \times C_n \\ \text{Decryption:} \qquad \mathbf{D}^{CBC} &: \mathbb{C}^n \times \mathbb{K} \to \mathbb{P}^n \qquad (3.3)\\ C &= C_1 \times C_2 \times \cdots \times C_n \\ P_i &= \mathcal{D}_K(C_i) \oplus C_{i-1} \\ \mathbf{D}^{CBC}(C) &= P_1 \times P_2 \times \cdots \times P_n \end{aligned}$$

As for block ciphers, we want decryption to be the inverse operation of encryption, and so we demand

$$\mathbf{D}(\mathbf{E}(P)) = P \qquad \text{for all } P \in \mathbb{P}^n \qquad (3.4)$$

For CBC, this requirement can be proved quickly. Using the encryption definition (3.2),

$$C_i = \mathcal{E}(P_i \oplus C_{i-1})$$

we apply $\mathcal{D}$ and obtain

$$\mathcal{D}(C_i) = \mathcal{D}\big(\mathcal{E}(P_i \oplus C_{i-1})\big)$$

Because of (3.1), we can strip off $\mathcal{D}\big(\mathcal{E}(\dots)\big)$ of the right side

$$\mathcal{D}(C_i) = P_i \oplus C_{i-1}$$

Thanks to the field properties of any GF(2) field we find the operation $\oplus$ to be self-inverse. Applying it, yields

$$\mathcal{D}(C_i) \oplus C_{i-1} = \Big(P_i \oplus C_{i-1}\Big) \oplus C_{i-1}$$

[1] The symbol for the Cartesian product will be relaxed to the concatenation symbol || in further definitions, because this precise formalism is not required to make definitions clear, and because ciphertext as well as plaintext is usually supplied in a concatenated form.

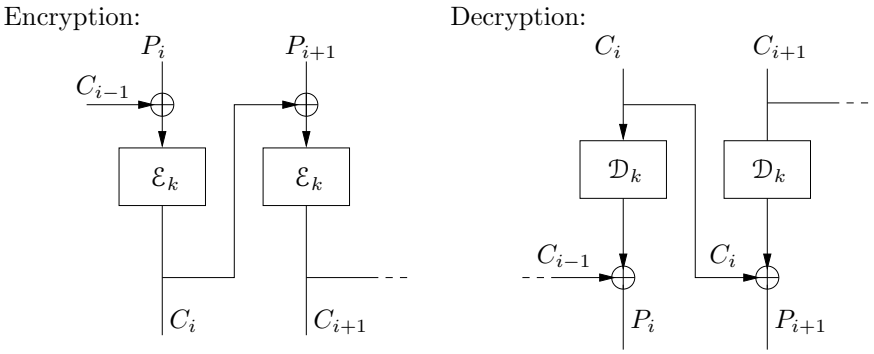


Figure 3.1: CBC Mode

and because of the law of associativity, and the neutral element $\underline{0}$, the left hand side can be refined to

$$P_i \oplus \Big(C_{i-1} \oplus C_{i-1}\Big) = P_i \oplus \underline{0} = P_i$$

If there is a proof of Equation (3.4) in a regular cipher mode paper, it is not carried out in such detail. However, we have chosen this level of detail to demonstrate that the field properties are nothing that could be omitted, and thus we justifiably devoted the pages of Section 2.2 to their investigation.

The recursive definitions (3.2) and (3.3) need an initial value for C_0. C_0 is not computed, but set to an initialisation vector to stop the recursion. In cryptography, this value is abbreviated by IV.

If encryption is used for communication purposes, an IV should be randomly generated and agreed upon by all participating parties[2]. The IV does not have to be kept secret. But this is different from the permission to transmit the IV unencrypted. Some modes are sensitive to manipulation of the IV, and when an IV is transmitted unauthenticated and unencrypted it is a target for potential manipulation. When negotiating the IV, the receiver has to make sure that the IV is authentic. Ways for secure CBC IV negotiation are described in [FS03].

What is also important about a cipher mode design is its parallelisation characteristic. This is especially interesting for hard disk encryption, because usual hard ciphers cannot keep pace with the I/O demands of modern RAID arrays. To make encryption transparent performance-wise, it is necessary to parallelise encryption among a cluster of cipher hardware. CBC is not parallelisable. You can see in Figure 3.1 that the recursion requires all preceding blocks to be enciphered. For any non-parallelisable cipher, it is possible to define an m-way interleaved version, so that plaintext block P_n is dispatched to $(n \mod m)^{\text{th}}$ encryption unit. Unfortunately, an m-way interleaved implementation and an n-way interleaved implementation is not compatible unless $m = n$.

Fault tolerance can be beneficial or undesired. CBC is fault tolerant to some extent. An error in the transmission of the CBC ciphertext causes the corresponding plaintext block to be completely garbled. But as you can see

[2]In fact, the IV should be unique and unpredictable for CBC, but if the IV domain is sufficiently large, it is acceptable to chose the IV at random.

from the decryption definition (3.3), a cipher block C_i appears twice, once involved in the decryption of P_i and once in the decryption P_{i+1}. The plaintext block P_i will be totally garbled, because a bit-error in the argument of $\mathcal{D}$ is diffused to the whole result. The block P_{i+1} will have a bit-error at the same position as C_i, because C_i is XORed into the decryption result. The term bit-error might not always be appropriate, as bits might be flipped intentionally by an attacker. Basically, this simple error-propagation is the cause for the attack outlined in Section 4.5. But before we investigate remedies to these problems, we have a look at other popular cipher modes.

3.1.2 CFB: Cipher Feedback Mode

CFB overcomes the block size limitations of CBC by adopting stream cipher properties. Any stream cipher encrypts and decrypts by XORing the plain or ciphertext with a key stream. Stream ciphers differ in the generation method for the key stream.

$$\begin{aligned}\mathbf{E}(P) &= S(\mathrm{IV}) \oplus P \\ \mathbf{D}(C) &= S(\mathrm{IV}) \oplus C\end{aligned}$$

As encryption joins the key material by the plaintext with a trivial XOR, decryption decomposition is equally trivial. An attack can easily modify the decrypted plaintext by flipping chosen bits from the ciphertext stream. These bit positions will be also flipped in the decrypted plaintext. This is particularly bad, if the structure of the plaintext is known, for example by some standard protocol. With some knowledge of the plaintext, an attacker can arbitrarily modify the decrypted plaintext. Thus, a stream cipher should never be used without an integrity checking mechanism.

The structure of the CFB mode is depictured in Figure 3.2. A key stream is prepared using a certain amount of preceding ciphertext material.

$$\begin{aligned}R_1 &= \mathrm{IV} \\ R_i &= (R_{i-1} \ll m) \mid\mid C_i \\ S_i &= \mathcal{E}_k(R_i) \\ S(\mathrm{IV}) &= S_1 \mid\mid S_2 \mid\mid \ \ldots \ \mid\mid S_n\end{aligned}$$

Given an n-bit block cipher, CFB can operate in m-bit mode, when $m \leq n$. After every step, m bits are shifted into the key register R, and an encryption call is made to produce the key stream for the next m plaintext bits. As for CBC, an initialisation vector is needed, which serves as initial content for the shift register.

The case $m \leq n$ is purely theoretical, as there is no gain from using a smaller shift size than the block size. In addition, there is the disadvantage that n/m encryption calls must be performed before a block with the size n is encrypted. The ratio n/m is minimised for $m \leq n$ when $n = m$.

3.1.3 OFB: Output Feedback Mode

OFB works similar to CFB, but the important difference is that it does not shift ciphertext into the encryption register, but short the key material itself.

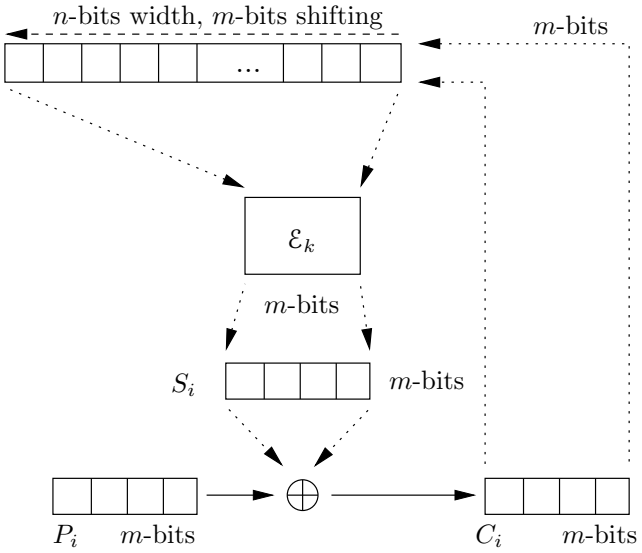


Figure 3.2: CFB Mode encryption

The key stream of OFB is generated by

$$S_0 = \mathrm{IV}$$
$$S_i = \mathcal{E}_k(S_{i-1})$$

when $m = n$. No other setting is stated here, as no other setting should be used. $m < n$ is insecure, see [DP83], and choosing $m < n$ also raises the computation burden for this mode.

A nice property of OFB is that key material can be generated in advance. An idle processor could fill a key material cache for data soon to come. This is not possible with CFB, as the key stream generation process depends on C_i, which in turn becomes available as recent as P_i.

3.1.4 CTR: Counter Mode

OFB and CFB are still defined recursively. In case of CFB the ciphertext is fed back, in case of OFB, the key material. This implies that no random access mechanism can be used on the cipher stream.

By removing this recursive reference and putting a more simple sequence in place, it is possible to seek in a cipher stream without decryption (or encryption) of the preceding blocks. Counter Mode also uses an initialisation vector, but instead of using a recursion to generate subsequent key material, it uses a simple addition.

$$S_i = \mathcal{E}_K(\mathrm{IV} + i) \tag{3.5}$$

Thus, it is possible to skip unneeded blocks in a stream, as the key stream can be easily computed for any block i. Also this mode is the first fully parallelisable one as there are no interdependencies for the generation of the individual key

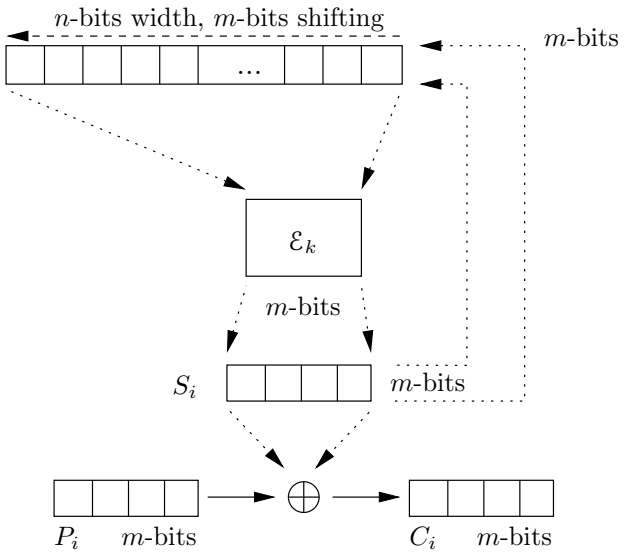


Figure 3.3: OFB Mode encryption

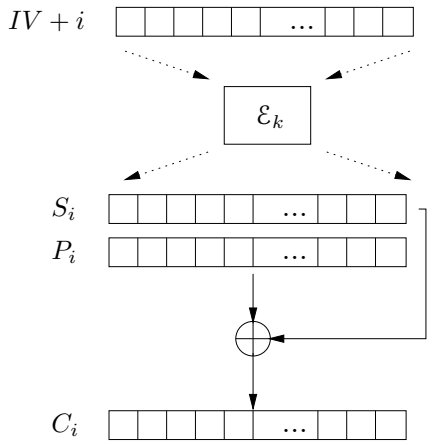


Figure 3.4: CTR Mode encryption

stream blocks S_i. However, special care must be taken that the values used in the key stream generation are unique. See [NIS03b] and Section 3.8.1.

As we will see in Section 4, the idea of using a counter for IV generation is used in a hybrid design with CBC to realise random-accessible data.

3.2 Modes for authentication

The purpose for authentication modes is to assure that the transmitted data has not been tampered with undetected. Authentication modes need a common secret, otherwise an attacker could modify the authentication data as well to

hide his data modifications. What distinguishes a regular digest from a message authentication code (MAC) is that a MAC is keyed to a secret. While regular digest algorithms are designed to be hard to invert, the existence of a common secret relaxes this requirement a bit.

3.2.1 CBC-MAC: Cipher Block Chain – Message Authentication Code

CBC-MAC is a side product of CBC encryption. The MAC of CBC-MAC is defined as the last cipher block produced by the CBC encryption of a message M. As we can see from the definition of CBC (3.2), the encryption process is recursively defined and the last cipher block C_n depends[3] not only on P_n, but also on all $P_1 \dots P_{n-1}$. The manipulation of a single bit in one of the plaintext blocks will cause C_n to change in a non predictable way. Neither can the modification of a P_i been compensated by a change in another P_j thanks to the diffusion of $\mathcal{E}$. Together with the key dependency of the process, these properties make CBC suitable as MAC.

The CBC-MAC can easily be obtained, but it is not possible to reduce the workload by using the same key for CBC-MAC and CBC encryption, when CBC-MAC is used to protect the plaintext of a CBC stream. The easiest informal proof for this assertion is the fact that when the last encryption block is reused as CBC-MAC, the information is merely duplicated, but no new information is added. The duplication of information is sufficient for integrity checks, but it is not for authentication checks, as the attacker can simply repeat his modification for every copy.

CBC-MAC is insecure for messages of varying length. An attacker can query an encryption-oracle for the single-block message a and will learn its tag CBC-MAC(a). Then he can reuse the tag as message and query the oracle for the tag of the tag, and will learn CBC-MAC(CBC-MAC(a)), but this tag is equivalent to the two block message $a \mid\mid 0^{:n}$, where n is the block length. Thus, the attacker can forge the message $a \mid\mid 0^{:n}$. The tags generated by CBC-MAC can only be used to authenticate messages with the same size.

If data is authenticated by CBC-MAC and encrypted by CBC, CBC-MAC has to run with a different key than a CBC encryption process running along side. OMAC tries to remove this requirement.

3.2.2 OMAC: One-key CBC

OMAC builds upon the XCBC structure.[4] XCBC shares many processing elements with CBC. By doing so, CBC encryption and XCBC generation taken together is very efficient. XCBC and CBC differ at the last encryption block. While CBC XORs the last plaintext block only with the preceding cipher block, XCBC also adds additional key material, K_2 or K_3. This will effectively change

[3] CBC is also said to be infinitely error-propagating for encryption, because the encryption function can be written as function of $P_1 \dots P_{n-1}$. Notice that CBC decryption has an error propagation length of 1, because decryption can be written as a function of C_{i-1} and C_i, but no other C_j. The ABC mode, which is very similar to CBC, installs an infinite error-propagation also for decryption, which enables the use of AREA (see Section 3.4.1).

[4] Do not confuse XCBC with the XCBC cipher mode. The MAC scheme and the cipher mode share nothing except the name.

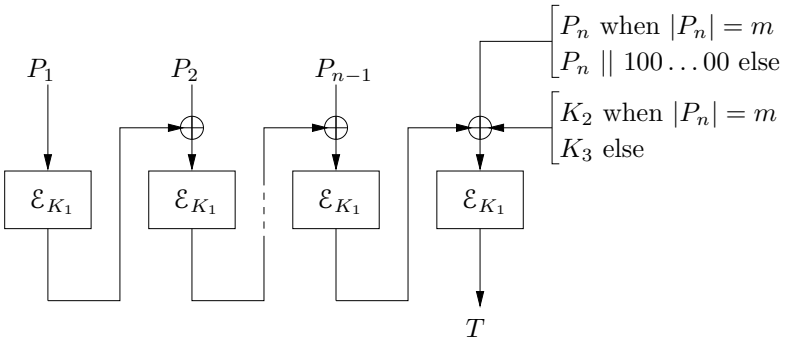


Figure 3.5: XCBC-MAC

the outcome of the final encryption step, yielding a result, which is totally distinct from the last block of the CBC encryption. Which one of the keys K_2 or K_3 is chosen depends on the message length.

XCBC does pad the message securely, which means that the padding can be removed in an unambiguous way. The padding is not done by merely adding zeros but instead '1' is inserted first, before filling the rest of the message with zeros. This is vital, otherwise there would be one MAC for two distinct short messages, M and M', with for instance, $M' = M \mid\mid 0\ldots0$.

By inserting '1' first, the position where the padding started is uniquely identifiable. To proof this, assume the last message block is inspected from back to forth. That is actually never done, but in theory the first encounter of '1' in the examination would mark the start of the padding. Such an examination would not work with the undifferentiated padding with 0.

To distinguish padded and non-padded messages, XCBC switches between the key material K_2 and K_3. Thanks to this key material selection, XCBC will produce different tags for two almost equal messages, M and M', where $M'_n = M_n \mid\mid 100\ldots00$.[5]

OMAC stands for One-Key MAC. It is a derivative[6] of XCBC. While XCBC requires three keys, OMAC generates them from a single key K by key splitting techniques.

The OMAC family in general and the specifications of OMAC1 and OMAC2 are presented in [IK02]. The algorithm for the OMAC family is given in Listing 3.6. OMAC implements XCBC with the following values for a given key K,

$$\begin{aligned} K_1 &= K \\ K_2 &= H_L(\mathrm{Cst}_1) \\ K_3 &= H_L(\mathrm{Cst}_2) \end{aligned}$$

[5] An approach taken by other designs is to always pad the message.
[6] Between OMAC and XCBC, there was also TMAC, Two-key MAC.

Listing 3.6: OMAC algorithm

```
L ← E_K(0^n)
Y_0 ← 0^n
partition P into P_1 ... P_m
for i ←1 till m − 1 do
    X_i ← P_i ⊕ Y_{i−1}
    Y_i ← E_K(X_i)
X_m ← pad_n(P_m) ⊕ Y_{m−1}
if |P_m| = n:
        X_m = X_m ⊕ H_L(Cst_1)
else
        X_m = X_m ⊕ H_L(Cst_2)
T = E_K(X_m)
return T
```

OMAC defines H_L, Cst_1, and Cst_2 as following:

$$
\begin{aligned}
H_L(x) &= L \otimes x \\
L &= \mathcal{E}_K(0^{:n}) \\
\mathrm{Cst}_1 &= x \\
\mathrm{Cst}_2 &= \begin{cases} x^2 & \text{for OMAC1} \\ x^{-1} & \text{for OMAC2} \end{cases}
\end{aligned}
$$

While OMAC1 and OMAC2 share most settings, the choice of Cst_2 is different. The mapping between polynomials and the bit representation is standard (as defined in section 2.1). The reduction polynomials for the Galois Field multiplication are defined in the OMAC paper [IK02].

3.3 Modes accepted for NIST consideration

The US-American National Institute of Standards and Technology does not stop short by only standardising a block cipher (AES), but also aims at modes of operation. The three main categories are, (1) encryption modes, (2) authentication modes, and (3) authenticated encryption modes. The last category is not merely the combination of a particular encryption mode with some authentication mode. An authenticated encryption mode tries to utilise synergy effects between encryption and authentication as best as possible while staying secure.

Before we take a brief look at the proposed scheme, we will describe common cipher mode properties. NIST asked submitters to describe their designs on the basis of a set of properties to make them easily comparable. Knowledge about these properties is of course essential for a reader to make a good choice.

Security Function A cipher mode is either for authentication (A), encryption (E), authentication and encryption (AE), or authentication-encryption with associated data (AEAD). All modes except the last operate on a single message. For a message $A \mid\mid M$, the last mode (AEAD) is able to protect the

confidentiality of a message M, and the integrity of both, A and M. The user of the mode can decide, what portion of a message is available in plaintext, without losing the authentication property.[7] AEAD modes are used for routing protocols, where you want the routing information to be available to all intermediate routers.

Performance The major criteria for the performance of a block cipher are expressed with other finer-grained properties (block cipher invocations, parallelisability and preprocessing capabilities, see below). However, the mode itself can feature other non-trivial computations like Galois Field multiplications. When there are such elements, they are stated in the performance entry.

Parallelisability Crucial for the general applicability of a mode is scalability with the amount of hardware put to its disposal. For instance, hardware used in CBC mode will quickly face its limits when used for network transmissions requiring a speed of 10GB/s or more. This speed can only be sustained by a cluster of hardware encryption engines. The distribution of the encryption load is only possible with parallelisable modes, where the encryption operations are reasonably independent of each other.[8]

Key material A mode might utilise more than one cipher primitive in its construction. Usually, these encryption/decryption primitives are all setup to use the same key. But more sophisticated modes might need additional key material. Either they require the user to supply it or they generate it on their own by key separation techniques (as for instance in OMAC).

Memory requirements A mode should have a low memory requirement for its cipher mode state. An elegant solution would be implementable at low cost even in smart cards and also in network routers, where many concurrent sessions must be handled.

Synchronisation Many cipher modes require the sender and receiver to agree on an initialisation vector. Some modes assume other values to be shared by both parties.

Error propagation The number of subsequent blocks that will be influenced by a bit-change in a block. See Section 3.4.1.

[7] The non-triviality of the addition of unencrypted but authenticated data is described in [Rog01a].

[8] A software implementation of the counter mode will usually increased the counter value by adding 1 successively, and also a definition following this idea can be given $CTR_0 = \mathrm{IV}, CTR_i = CTR_{i-1} + 1$. However, this does not qualify as non-parallelisable cipher mode, because the addition operation can be easily rewritten as non-recursive definition, and the single steps of the counter mode can be made "reasonably independent" of each other. A more interesting example is given in Section 3.4.4 and Section 3.4.5, where an apparently recursive structure is separated to be parallelisable.

Message length requirements & ciphertext expansion We encounter three common types of modes:

1. modes requiring the user to provide a message, which size is a multiple of the underlying block cipher size.

2. modes that accept messages of all sizes by padding the message. The message ciphertext is expanded to the next integral multiple of the block cipher size.

3. stream cipher modes that accept messages of all sizes and do not expand the ciphertext.

In authentication schemes, the ciphertext is also expanded by the inclusion of the MAC.

Block cipher invocations This property indicates how many block cipher calls a design has to carry out for the message M with an underlying n-byte cipher.

Pre-processing capabilities A desired property of a cipher mode is that substantial computing can be done in advance, so that when plaintext or ciphertext becomes available for processing, it can be worked off quickly. An encryption system running in Counter Mode for example, can spend idle time to pre-process the key stream for the next burst of data, and therefore reduce the latency of the system.

Provable security A methodology first explicitly formulated by Bellare and Rogaway is the proof of the security of cipher mode by the use of the random-oracle model [BR93].

For cipher primitives, the distinguishing attack class aims to cover all other attacks, like known plaintext, chosen plaintext, chosen ciphertext and so on. Cryptographers are cautious people, and they mark a cipher insecure, when there is an algorithm that can distinguish an ideal cipher[9] from the real cipher with a significant probability. A probability is significant in this context, when it is above the success rate that can be achieved by random guessing.

When such an algorithm is found, the algorithm is then said to implement a distinguishing attack.[10] Cryptographic papers use the adversary metaphor to speak of such algorithms. The distinguishing attack is so general, that it is hoped to cover all attacks yet to be found.[11] The oracle methodology used for cipher modes is similar.

To proof a cipher mode insecure, the adversary has to distinguish the cipher mode from a random permutation with a significant probability. The security of the cipher mode shall be proved independent of an underlying cipher, so the cipher is replaced by an ideal cipher represented by an encryption oracle. All

[9] An ideal cipher is a key-dependent random permutation. See [FS03], p. 46.

[10] The algorithm has to be non-trivial. An example for a trivial adversary is one that samples the real cipher, for instance with an all zero plaintext and key, and compares the sample with the candidate, which is either the real cipher or a perfect cipher.

[11] In Chapter 4, we will encounter the watermarking attack, which is basically a distinguishing attack against CBC.

parties have access to this oracle, also the adversary. This is reasonable, because in a real world implementation, the oracle is replaced by a real cipher and as security has nothing to do with obscurity the adversary is assumed to have access to the cipher specification. Hence, he can carry out any computation he likes, which is equivalent to querying the oracle.

The encryption oracle behaves like an ideal cipher. The security proof consists of a series of formal deductions that shows that there is no adversary that can distinguish the cipher mode from a random permutation. If there is such a proof, the cipher mode stands a good chance that no piece of information can be extracted from its ciphertext stream, as an attacker cannot even distinguish the ciphertext from a random permutation.

Under the assumption that the random oracle is an ideal cipher, the proof is a series of indisputable formal statements. In a real implementation, the random oracle has to be replaced with a pseudo-random permutation (PRP), when we consider ciphers, or a pseudo-random function (PRF), when we consider hashes. A security proof is likely to show, that breaking the cipher mode will require breaking the underlying PRP or PRF.[12]

Patent status Last but by no means least. Ciphers and cipher modes can be patented in a few parts of the world. Even though the number of such countries is small, a global standard or a product intended for global distribution has to consider this issue.

From a cryptographers point of view, patents are totally irrelevant. A cryptographer is interested to achieve a fixed level of security using a design with the best possible performance, or – less common – the best security for a fixed performance level. But the cryptographic community, which is subtly different from an agglomeration of scientists, care a lot about patents. The cryptographic community considers a third factor in their preferences next to security and performance: the unhampered use of intellectual property[13]. From the NIST standardisation efforts for a cipher, and a cipher mode, the latent refusal of patent-encumbered proposals from the community became highly visible. The members of the community refuse to work on and publish reviews of patented designs, because they have the feeling to work for the benefits of the patent holders without receiving any compensation for their efforts[14]. Also, it is argued that patent licensing poses a burden to small businesses and international users.

From an implementors point of view, it rarely makes sense to enter into the costly process of licensing terms negotiations. It simply does not pay off. For instance, there are many free two-pass authenticated encryption modes, but also many one-pass patented modes. Choosing a two-pass design over a

[12] [CGH04] shows that things are not as straight forward. Canetti, Goldreich and Halevi give a class of protocols that result is an insecure cipher mode, when their random oracle is replaced by a real implementation. The reader might be tempted to think that this is caused by an unlucky oracle-replacement choice. But no, the authors demonstrate that there is no oracle-replacement at all that will yield a secure implementation. The cipher modes used in the demonstrations are nothing to encounter in practise, nonetheless, a security proof should not cease further cryptanalysis, as there is no ideal cipher in the real world.

[13] In this context, the term "property" is a misnomer, as the essence of the expression "property" is the constraint of the rights of third parties.

[14] Ferguson describe this attitude briefly in the introduction to [Fer02].

patented two-pass design makes sense, because the additional cost of an AES engine in silicon is likely to be smaller than the licensing fees.

3.4 NIST: Authenticated Encryption modes

Before we have a look at concrete modes, we classify generic construction methods. In the introduction to the last section, we claimed that authenticated encryption modes use synergetic effects between the task of authentication and the task of encryption. Unfortunately, this is rather an option than a norm. Cipher modes that can produce results in a single pass are exceptions. The problem is not the incapabilities of cipher mode designers but patents. The patents on IAPM, OCB and XECB cover most of the approaches that can be used to produce one-pass modes. At the time of this writing, it is not clear if these patents are overlapping. At the second NIST workshop not even the XECB author was able to give a clear answer to the question, whether his patents also apply to IAPM or OCB [NIS01]. It is not surprising that there are solutions that stick to the safe side. They are constructed by simply gluing an authentication mode to an encryption mode to produce an authenticated encryption mode.

3.4.1 Generic construction methods

Encrypt-then-Authenticate

There are three ways to pair an authentication mode and an encryption mode together:

1. encrypt-then-mac: The data is encrypted with a cipher $\mathcal{E}$ first and then authenticated with an authentication mode $\mathcal{A}$. The result for a message M is $\mathcal{A}_K(\mathcal{E}_K(M))$.

2. mac-then-encrypt: The data is authenticated with an authentication mode $\mathcal{A}$ first and then encrypted with a cipher $\mathcal{E}$. The result for a message M is $\mathcal{E}_K(\mathcal{A}_K(M))$.

3. encrypt-and-mac: The encryption and the authentication is done independently. The result is provided separately as $\mathcal{A}_K(M) \mid\mid \mathcal{E}_K(M)$.

Most cipher modes we encounter are of the type "encrypt-then-mac". By the findings of [BN00], the encrypt-then-mac construction always achieves the desired security, while encrypt-and-mac does not. Also for the mac-then-encrypt scheme, its security is not as guaranteed as for encrypt-then-mac[15]. Because mac-then-encrypt is a simple reordering of steps compared to encrypt-then-mac, it will never save any block cipher calls, therefore mac-then-encrypt is not considered, and encrypt-then-mac is the only scheme left.

[15]This approach violates the Horton principal. Ferguson argues in [FS03] that one should not authenticate what is being said, but what is being meant. He also argues that the security disadvantages are negligible that arise from choosing mac-then-encrypt over encrypt-then-mac. However, we find no mac-then-encrypt schemes at NIST.

AREA: Added Redundancy Explicit Authentication

Any cipher mode that possesses an infinite error propagation can be used in the construction of an authenticated encryption mode. Error-propagation is defined as the number of arguments that must be present to write encryption or decryption as a function of blocks P_i or C_i. CBC encryption is infinitely error propagating as C_i is written as a function of all previous plaintext blocks $P_1 \dots P_i$, while decryption can be written as a function of only two blocks. For a discussion of the advantages and disadvantages of having a cipher mode with infinite error propagation, see [Knu00].

AREA works by appending a value L to the plaintext stream. If decryption is infinitely error propagating, then a bit-change in any C_i will cause the last blocks plaintext P_n to change, as P_n is dependent on all blocks C_i thanks to error propagation. The value L is prepended unencrypted to the cipher stream, and the receiving party can check the authenticity of the ciphertext by decrypting the message and comparing the last plaintext block with L. If the message has not been tampered with, the last plaintext block decrypts correctly to L.

Manipulation Detection Code

Manipulation Detection Code is a generic term for a function that generates a checksum of a message. There are sophisticated and very simple forms of MDC. An example of a sophisticated MDC is a cryptographic hash function. Hashing the plaintext and appending the hash to the encrypted text results in a secure authenticated encryption scheme. But this approach is even slower than applying a MAC, as hash functions are usually slower[16].

Simpler forms of MDCs are cyclic redundancy codes or even plain XOR sums. In particular the last choice might not yield a secure authenticated encryption algorithm under every circumstance. For instance, an XOR sum is totally insecure when used with stream ciphers, because an attacker can manipulate a bit position twice in a message stream, so that the XOR summation yields the same result as before, or he can manipulate the MDC directly.

For XOR sums, it is trivial for the sender to generate collisions. For authentication systems, and hashes, this characteristic would make them useless. Authenticated encryption schemes are different in the amount of data they produce and the purpose they are built for. AE schemes are designed to ensure the authenticity of something that was said, while hashes and MACs protect something that had not been said. While for the former it is easy to check whether the tag is correct, the receiving party for MACs or hashes has to make up its own idea of what was meant but not said. When MACs and hashes are used to bind the sender to a statement, the receiving party is in trouble, when the sender can come up with a different meaning for a MAC value, and simply claim that the receiver had the wrong idea of what was meant. With authenticated encryption schemes this is not possible, as the tag and the ciphertext are

[16]The reason for this is that a hash function does not have a secret piece of information it can rely on. In contrast, a MAC can use key information and therefore does not have to be as collision resistant. The iterations of a hash and the rounds of a cipher are hard to compare, but in general, a hash is more computation intensive.

treated as a pair and both are bound to a single nonce. Hence, it is reasonable to use such a simple checksum mechanism.

In general, an implementor should not try to glue an arbitrary MDC to an encryption algorithm and hope it will result in a secure solution. [GD00a] gives insights about the different types of security goals that can be achieved by those synthetic construction.

3.4.2 CCM: CBC-MAC with Counter

CCM's security function is AEAD, so it cannot only authenticate encrypted data but also plaintext data sent along. CCM uses a CBC-MAC for generating authentication data, and Counter Mode for encrypting the message. First, the data is encrypted in Counter Mode, then paired with the additional data to be sent in plain, and finally a MAC is generated for the whole message.

CCM is one of a few modes that can be parameterised. For interoperability the parameter values are required to be encoded into the data stream. The parameters L and M can be tuned.

L describes the ratio between nonce size and message size. The counter mode part in CCM takes the value A_i and forms the key stream by encrypting it, $S_i = \mathcal{E}_k(A_i)$. A_i contains a nonce and the message block number i. The longer a message is, the more message block numbers are used up in the encryption, and therefore the field holding the block number must be larger. As this field becomes larger the nonce size has to decrease, as the size of A_i is fixed and imposed by the block cipher size. Because in Counter Mode special care has to be taken that the key stream base values A_i are not reused, the explicit mixing of the nonce and the message block number is a reasonable approach.

The parameter M stands for the size of the authentication tag. A larger value for M achieves better security against forged messages by trivial means. But this adds extra costs by having to transmit the larger tag. CCM is designed for scenarios, where in general, authentication is desirable, but the forgery of a single message is not fatal. An example is an encrypted video stream.

CCM is specified in [WHF]. CCM has been adopted in the IEEE 802.11 wireless standard.

Despite many alternatives, NIST decided to move forward and submit a recommendation for this mode. The cryptographic community voiced their criticism in a number of papers. The most prominent comment came from Rogaway, the designer of OCB [RW03]. The most often voiced concern is that for CCM the message size has to be known before hand. As we demonstrated in section 3.2.1, CBC-MAC cannot produce MACs for messages of varying length. The CCM designers fixed this limitation by prepending the size of the message to the message. But this makes the encryption process "offline", meaning that encryption cannot take place until the whole message is handed to encryption routine. Of course, at the start of the encryption, the caller could supply the number of message blocks, that will be handed to encryption, but then the caller himself has to know the message length in advance. This pushes the problem to an upper layer, hence algorithms that are online result in better performance under certain situation, as no message buffering is necessary.

In response, NIST made some small changes to the recommendation document, but insubstantial [NIS03a]. While writing this document, NIST made an unusual step: It decided to publish an alternative standard for authenticated

encryption. It can be argued that this is because of the shortcomings of CCM. The candidates for the second recommendation are CWC and GCM.

3.4.3 EAX: Encrypt-Authenticate-Translate

EAX is an enhanced version of CCM. CCM is using CBC-MAC for authentication. EAX is using OMAC instead of CBC-MAC, which was shown to be a close relative in the previous section. This change removes the requirement for the sender to know the message size in advance. The block cipher calls are almost identical to CCM. EAX has been presented in [BRW03].

3.4.4 CWC

The counter mode part of EAX and CCM can be processed in parallel, but OMAC and CBC-MAC are still recursive constructions that prevent any parallelisation. CWC is the attempt to replace the MAC component by the parallelisable "Carter-Wegmen" scheme for a Message Authentication Code. The theoretical foundation for this MAC is laid in [WC81].

The authentication tag T is computed as the result of a polynomial under mod $2^{127} - 1$ for a given x, and a series of coefficients $Y_1 \dots Y_n$ corresponding to the message represented as 96-bit integers. The length of the message is encoded into Y_{n+1}.

$$T = Y_1 x^n + Y_2 x^{n-1} + Y_3 x^{n-2} + \dots Y_n x + Y_{n+1} \mod 2^{127} - 1$$

The value x is derived from the key material. This integer addition can be easily rewritten to suit processing on more than one processors. For instance, the equation can be split into two polynomials, P_1 and P_2, by interleaving the coefficients and at the same time replacing x with y, which is set equal to $y = x^2$.

$$T = \underbrace{(Y_1 y^m + Y_3 y^{m-1} + \dots + Y_n)}_{P_1} x + \underbrace{(Y_2 y^m + Y_4 y^{m-1} + \dots + Y_{n+1})}_{P_2} \mod 2^{127} - 1 \quad (3.6)$$

P_1 and P_2 can be evaluated in parallel by the original algorithm. After that, the results are combined with $P_1 x + P_2$. The cipher mode is presented in more detail in [KVW03] including a security proof.

3.4.5 GCM: Galois Counter Mode

GCM tries to hone the properties of CWC even more. GCM utilises Galois Field multiplications to hash additional authentication data with a universal hash. Galois Field multiplications can be implemented in hardware at a much lower cost than the 127-bit integer arithmetic used by CWC. Furthermore, the literature on GF multiplications in hardware is fairly rich and a variety of schemes can be used, whatever is most suitable in terms of performance, gate count and required working memory. As before, encryption is done by the cipher running in Counter Mode.

The overall construction of GCM follows the "Encrypt-then-Authenticate" scheme, again putting the unencrypted but authenticated data in front of the encryption output. The additional authentication data and Counter Mode encrypted data are split into 128-bit blocks $A_1, \ldots A_m$, and $C_1, \ldots C_n$. The authentication tag is then computed by the GHASH(H,A,C) function:

$$\text{GHASH}(H, A, C) = H^{n+2} \left(\bigoplus_{i=1}^{m} A_i H^{m-i} \right) \oplus H^2 \left(\bigoplus_{i=1}^{n} C_i H^{n-i} \right) \oplus F\, H$$

where F represents the encoded length of A and C, $len(A) \mid\mid len(C)$. Because $\text{GF}(2^{128})$ is a field like $\mathbb{Z}$, the same polynomial computation reordering can take place as for CWC. Hence, this mode is parallelisable. For more in depth information, the reader is referred to the GCM paper [MV].

3.4.6 IACBC

IACBC stands for Integrity Aware Cipher Block Chaining and is introduced in [Jut01]. IACBC implements regular CBC with a post-whitening step of the ciphertext with S_i. S_i is composed of a vector, W. W contains k elements, where $k = \lceil \log_2(i) \rceil$. How S_i is computed from W is controlled by a Gray Code. If $b_k \ldots b_0$ represent the binary digits of the i^{th} Gray Code as defined in Section 2.5, then S_i is computed according to

$$S_i = \bigoplus_{j=0}^{k} b_j W_j$$

Thanks to the Gray Code oriented generation only one XOR has to be carried out to generate S_i from S_{i-1}. As IACBC builds on CBC, this mode is not parallelisable. The MAC is generated by (1) XOR summation of the plaintext, and (2) treatment of the XOR sum as regular plaintext by appending it to the plaintext stream.

The technique of adding a key dependent value before or after the encryption is called pre- or post-whitening and was seen in the design of DES-X for the first time. The term is derived following the idea the colour white is a uniform and unstructured mixture of all colours. Adding whitening values to the plaintext aims to destroy all structures that might existing in plaintext, so it does not infer with the encryption process. This technique is already included at cipher level in recent cipher designs.

3.4.7 IAPM

IAPM is a refinement of IACBC introduced in [Jut00]. Instead of recursion, IAPM uses S_i as pre- and post-whitening value. The resulting construction $\mathcal{E}(P_i \oplus S_i) \oplus S_i$ is similar to the tweakable cipher mode construction given by Liskov, Rivest and Wagner in [LRW02]. The MAC generation is the same as in IACBC.

With the removal of the recursion of the encryption process and therefore its error-propagation, it becomes possible to produce the same tag for two different plaintexts. Consider a bit change in a plaintext P_i and another bit change at

exactly the same position in plaintext block P_j. The XOR summation will yield the same results, both for the original plaintext and the modified one. However, tag collisions are not fatal for authenticated encryption modes, because the ciphertext cannot be manipulated as easily. An attacker has little hope to control single bits in the decrypted plaintext by controlling the ciphertext. The diffusion properties of the cipher primitive propagate any bit change in ciphertext in an unpredictable way, thus introducing more than a single bit change in the decrypted plaintext. The tag verification is likely to fail.

An isolated control of bits would be possible for data transmitted in plain. MACs building upon XOR summation of plaintext transmitted in plain are therefore not secure. Why the collisions that can be generated by the sender at will are no problem is explained in Section 3.4.1.

3.4.8 OCB: Offset Codebook

OCB is an all-rounder. It provides message integrity at almost optimal block cipher calls, does not expand the ciphertext, and puts only minimal requirements on the IV. The pitfall: It is patented. OCB is a refinement of IAPM and differs in that the whitening values are generated by Galois Field multiplications also controlled by a Gray Code. These whitening values are suitable for preprocessing.

$$L = \mathcal{E}_k(0)$$

$$R = \mathcal{E}_k(N \oplus L)$$

Assume γ_i to be the respective binary reflected Gray Code of i as defined in Section 2.5. The whitening values are defined as

$$Z_i = \big(\gamma_i \otimes L\big) \oplus R$$

The ciphertext is formed by the generic construction

$$C_i = \mathcal{E}_k(P_i \oplus Z_i) \oplus Z_i$$

To achieve minimal ciphertext length requirements, the last block, M_n is treated differently.

$$Y_n = \mathcal{E}_k\Big(|P_n| \oplus \big(x^{-1} \otimes L\big) \oplus Z_n\Big)$$

$$C_n = Y_n\langle 0, |P_n|\rangle \oplus M_n$$

where $|M_n|$ denotes the length of the message block n and $\langle i, k\rangle$ is the byte-slice operator defined in Section 2.1. Also, the multiplicative inverse of x is guaranteed to exist as the underlying structure is a field.

The checksum is generated by the XOR sum of all P_i, while additional material from Y_m is also used to pad the last plaintext block,

$$S = \Big(\sum_{i=1}^{n-1} P_i\Big) \oplus (P_n \;||\; Y_n\langle |P_n| + 1, n\rangle)$$

$$T = \Big(\mathcal{E}_k(S) \oplus Z_n\Big)\langle 0, \tau\rangle$$

The remarks regarding tag collision in IAPM also apply here. As you can see, there is only one block cipher call per plaintext block. The value L is assumed to be computed at key setup, which needs to be evaluated only once.

For a security proof and an in-depth discussion, the reader is referred to [RBBK01]. Fergeson presents a collision attack in OCB [Fer02]. Rogaway's remarks on the collision attack can be found in [Rog02][17].

3.4.9 CS: Cipher-State

The Cipher-State mode provides a new approach to generating authentication data. This cipher mode does not regard the cipher primitives as black box, but the cipher is dissected and examined to extract round information. This is a reasonable approach, since all recent ciphers are round-based. The main motivation to do so is the considerable performance gain.

Also in this design, we find Galois Field multiplications and a construction of the type: $\mathcal{E}_k(P_i \oplus R) \oplus R$, see [LRW02]. The whitening value R is generated from iterative GF multiplications of the form

$$R_i = R_{i-1}x \mod r$$

where r is the reduction polynomial.

An intermediate cipher state t is obtained by tapping right into the middle of encryption. This intermediate state is used for authentication only and will be incorporated into another series of GF multiplications (like for R). The authentication code is formed by

$$A_i = A_{i-1}\, x \oplus t_i \mod r$$

After tapping the encryption process, the encryption continues normally, and the ciphertext used for encryption is not altered.

The concrete algorithm also includes special cases for encryption-only, as well detects weak whitening values (all-zero). The interested reader is referred to [BDST03].

3.4.10 PCFB

Propagating CFB (PCFB) is a slightly modified version of the regular CFB mode. Instead of using the shift register entirely for ciphertext, it is partially run in output feedback mode. As the regular CFB mode, PCFB can be run with a bit size m smaller than n, the number of bits of the underlying cipher. If $m = n$, PCFB and CFB are equal. If $m < n$, than the bits $m \ldots 0$ of the feedback register are set to the ciphertext (just like in regular CFB mode), but bits $n \ldots m$ are not shifted, but set to the bits $m - n \ldots 0$ of the key stream (like in OFB).

To provide authentication, an AREA (Added Redundancy Explicit Authentication) is added to the ciphertext stream. As said in Section 3.4.1, this is a random value prepended to the ciphertext stream, that is also encrypted as closing plaintext block. If any intermediate ciphertext is manipulated, the closing plaintext decryption will be changed inevitably by the error propagating

[17] Notice that the mode WHF Rogaway refers to has been renamed to CCM.

characteristic of PCFB. More information is available in the PCFB submission paper [Hel01]. Notice that this is one of the few unpatented AE schemes that provide authentication with a single block cipher call per plaintext blocks. In contrast, all ETA constructions have to use two.

3.4.11 XCBC-XOR

XCBC-XOR is a descendant of XCBC as specified in [GD00b][18]. First, we will describe XCBC and later see how it is combined with an XOR sum to ensure authenticity. XCBC builds upon CBC, but post-processes the blocks by adding a whitening value, hence any existing CBC hardware optimisation can be reused. First, the regular chaining of CBC is obtained,

$$Z_i = \mathcal{E}_k(P_i \oplus Z_{i-1})$$

then the post-whitening is applied

$$C_i = Z_i + i \times r_0$$

by computing the integer addition under mod 2^m, when using an m-bit cipher. The post-whitening is necessary to counter cut&paste attacks as we will see in Section 4.6.

XCBC comes in three flavours: stateless, stateful, and a mixture of stateful encryption and stateless decryption (the same is true for XCBC-XOR). XCBC starts with generating a random variable r_0 and computing Z_0 from this variable. The three flavours of XCBC-XOR differ with respect to the computation of Z_0 and distribution of the value r_0.

	stateless	stateful-sender	stateful
encryption's r_0	random	$\mathcal{E}_K(ctr)$	random
first block y_0	$\mathcal{E}_K(r_0)$	ctr	$\mathcal{E}_K(r_0)$
decryption's r_0	$\mathcal{D}_K(y_0)$	$\mathcal{E}_K(ctr)$	$\mathcal{D}_K(y_0)$
Z_0	$\mathcal{E}_K(r_0+1)$	$\mathcal{E}_K(r_0+1)$	$IV + r_0$

The authors give a generic method of pairing XCBC with a Manipulation Detection Code (MDC) to obtain an authenticated encryption mode. In [GD01], the authors propose to pair XCBC with an XOR sum. The XOR sum is computed with the following equation and treated as regular plain text block:

$$P_{n+1} = \Big(\sum_{i=1}^{n} P_i\Big) \oplus Q$$

The value of Q depends on the length of P_n. If $|P_n| = m$, then Q is chosen $Q = Z_0$, otherwise $Q = \overline{Z_0}$.[19] The original plaintext and the XOR checksum, $P \mid\mid P_{n+1}$, are handed over to encryption.

[18] If you obtain this paper as PDF, do not be confused by the use of ⇔. This seems be a font encoding error, and every instance of ⇔ should be read as dash/minus sign.

[19] $\overline{x}$ denotes the inverse element under the $\oplus$ operation for x, or in other words, $\overline{x}$ is the bitwise complement of x.

The decryption process operates regularly on the whole plaintext and recovers P_i, $i = 1 \ldots n+1$. To check authenticity, two checksum values are obtained

$$S = \Big(\sum_{i=1}^{n} P_i\Big) \oplus Z_0 \qquad\qquad S' = \Big(\sum_{i=1}^{n} P_i\Big) \oplus \overline{Z_0}$$

Both values are compared with P_{n+1} obtained by decryption. If neither S nor S' is equal to P_{n+1} the message is manipulated and an error is returned. If it is equal to S, then the message is known to be authentic and unpadded. If it is equal to S', the message is authentic and padded. The padding pattern is removed and the original message is returned. XCBC-XOR is specified in [GD01].

3.4.12 XECB-XOR

The periphery of XECB is similar to XCBC, as it was designed by the same authors. The padding process and the XOR summation are equal. The core process is different though. There is no cipher block chaining, but a pre- and post-whitening construction.

$$C_i = \mathcal{E}_k(P_i + W_i) + W_i$$

Notice that in contrast to other designs, the underlying algebraic structure is the group of natural numbers mod 2^{128}. In general, an addition in this group is slower than an addition in a Galois Field, as the former has to be carried out by bitwise addition, and the latter by bitwise XOR. There is a severe performance difference between bitwise addition and bitwise XOR in favour of XOR, because for addition, there is a the carry-bit that has to be propagated through the adder gates of the ALU. Hence, $\mathcal{E}_k(P_i + W_i) + W_i$ is slower than $\mathcal{E}_k(P_i \oplus W_i) \oplus W_i$.

The whitening values are

$$W_i = \mathrm{ctr} \times R + i \times R^*$$

The distinction between padded and non-padded messages is made by means of the value Q that is used in the whitening of the XOR checksum. Q is computed just like in XCBC. XECB-XOR is presented in [GD01].

3.5 NIST: Encryption modes

Even though NIST has already standardised Counter Mode encryption as the standard mode for confidentiality-only settings, we take a look at the other approaches.

3.5.1 2DEM

2DEM is short for 2D-Encryption Mode. It treats the data as a two dimensional structure, and its encryption operates first on all rows, then an all columns. The process is depicted in Figure 3.7. First, a row encryption is carried out to obtain the intermediate results i_{kk}. Then, a column encryption produces the final

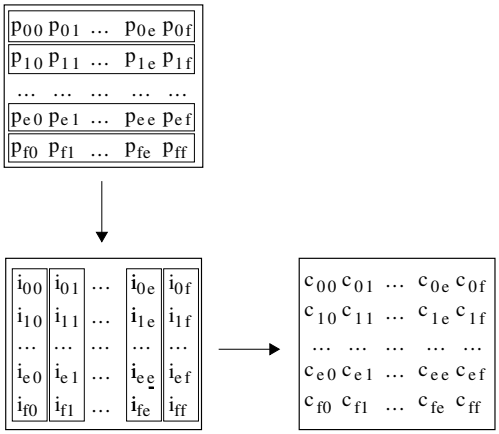


Figure 3.7: 2D-Encryption Mode

ciphertext. Assuming a 16x16 byte structure, a single bit change is first diffused to the entire row by an AES encryption operation, and afterwards diffused to the whole structure by column encryption. Operating an n-byte block cipher this way will form indivisible blocks of length n^2. 2DEM is described in more detail in [BAG01].

3.5.2 ABC

CBC encryption reuses cipher blocks in the process of obtaining further cipher blocks. ABC extends the idea to plaintext, and instead of encrypting the plaintext, it encrypts the XOR sum of the plaintext with all previous plaintexts. Optionally, the plaintext is hashed. The rationale behind this is to make the decryption process more symmetric to encryption in terms of error-propagation. This is sometimes desired, for example, to add an AREA, see Section 3.4.1.

Formally, ABC is defined as,

$$H_i = P_i \oplus h(H_{i-1})$$

$$C_i = \mathcal{E}_k(H_i \oplus C_{i-1}) \oplus H_{i-1}$$

This is invertible,

$$H_i = \mathcal{D}_k(H_{i-1} \oplus C_i) \oplus C_{i-1}$$

$$P_i = H_i \oplus h(H_{i-1})$$

Choosing $h(P) = P$ will make H_i the XOR sum of all plaintexts up to block i. Optionally $h(P) = P \otimes x$ can be chosen to include a light diffusion element. Choosing $h(P) = 0$ makes this mode equal to IGE. ABC is presented in [Knu00].

3.5.3 IGE

Infinite Grappling Extension is a special case of ABC, and thus it inherits ABC's error-propagation characteristics. In IGE, decryption and encryption is

symmetric insofar as the structure of both equations is equal. As you can see from the definitions,

$$C_i = \mathcal{E}_k(C_{i-1} \oplus P_i) \oplus P_{i-1}$$
$$P_i = \mathcal{D}_k(P_{i-1} \oplus C_i) \oplus C_{i-1}$$

decryption is merely: P_i switched with C_i, and E_k being replaced by D_k. Cipher modes featuring symmetric encryption and decryption are often chosen for embedded systems, since when encryption and decryption can share the same circuits. But this is not been the main motivation of IGE. Adding infinite error propagation paves the way for appended checksums computed by an MDC. AREA is also an option.

3.6 SISWG

In this section, we will look at cipher modes especially designed for hard disk encryption. SISWG, Security In Storage Working Group, is an IEEE task force with the designated goal to standardise security solutions for data at rest. At the moment of this writing, there are severals drafts considered as cipher mode standard. They are available from the SISWG website [SIS].

SISWG plans to standardise a narrow cipher mode and a wide cipher mode. A narrow cipher mode operates with a block size equal to the block size of the underlying cipher. A wide cipher mode involves more cipher blocks and forms an integral unit the size of a sector by letting the cipher mode diffuse the information across the whole sector.[20]

LRW is the favourite for becoming the standard narrow cipher mode, while EME is the favourite as wide cipher mode. This section also covers ABL, because it is the only wide cipher mode in the SISWG discussion that is free of any intellectual properties restriction.

3.6.1 LRW: Liskov, Rivest, Wagner

The most promising candidate for the narrow cipher block mode is LRW. Before we are going into technical details, LRW is not the official name. It is rather the nick name for LRW-AES. LRW is short for Livest, Rivest, and Wagner, and this trio has given LRW its name (although not deliberately). They authored the paper "Tweakable Block Ciphers" [LRW02] that outlines a generic way for constructing a tweakable cipher mode from a function $H : \mathbb{P} \to \mathbb{P}$ with certain characteristics[21].

$$C_n = \mathcal{E}_k(P_i \oplus H(T)) \oplus H(T)$$

LRW-AES utilises this construction method by filling in a Galois Field multiplication for H. LRW-AES, sometimes referred to as LRW-AES-32, is drafted by SISWG in [Ken04].

[20] A few proposals were made to construct a wide block cipher instead of a wide cipher mode, [BR99], [Luc96], but none of them is suitable for hard disk encryption, because either they are not tweakable or have certificational weaknesses according to [Cro00]. [Cro00] also introduces a wide block cipher named Mercy, that aims to close all these gaps. Unfortunately, it has other problems, and was successfully crypt-analysed by Fluhrer in [Flu01].

[21] The function H has to be ϵ-almost 2-xor-universal, short ϵ-AXU$_2$. The formal definition of this property is $\forall x, y, z : Pr_h[h(x) \oplus h(y) = z] < \epsilon$. A proof that Galois Field multiplications belong to this class is given in [BGKM03].

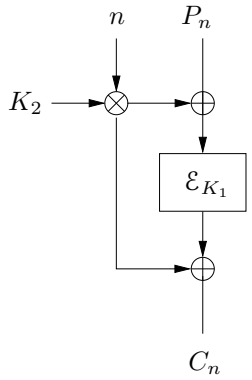


Figure 3.8: LRW-AES Mode

LRW-AES ties a ciphertext to a disk location by pre- and post-processing the cipher block with the result of Galois Field multiplication. Figure 3.8 shows the encryption process for a single plaintext block. The structure of LRW-AES is symmetric for encryption and decryption.

As you can see from the figure, LRW-AES needs an additional second key, which serves as a constant for the GF multiplication. K_2 is not derived from the key material and has to be supplied additionally. The operands and the output of the GF-multiplication have the same size as the block size. While K_1 can change its size (128, 192, 256-bit for AES), K_2 remains fixed at block size (128 bits), resulting in a variation of the total key size from 256 to 386 bits.

A security proof for the tweakable cipher mode construction in general is available in [LRW02]. In particular, LRW-AES uses Galois Field multiplications, that are considered in the security proof in [BGKM03].

3.6.2 EME: ECB-mix-ECB

EME is a parallelisable cipher mode developed by Halevi and Rogaway in [HR03a]. EME can be decomposed into 5 stages involving 3 complete traversals of the data. At key setup, L is computed as $\mathrm{L} = \mathcal{E}_k(0^{\cdot n})$.[22]

1. preprocessing by XORing $\mathrm{L} \otimes 2^n$ into the plaintext block P_n.
2. encryption of all blocks
3. computation of the mask M and XORing $\mathrm{M} \otimes 2^{n-1}$ with the blocks $2 \ldots n$. The first block is treated separately.
4. encryption of all blocks
5. postprocessing of the ciphertext by XORing $\mathrm{L} \otimes 2^n$ with the cipher blocks C_n.

[22] Notice, the original paper states $\mathrm{L} = 2 \otimes \mathcal{E}_k(0^{\cdot n})$, but we have changed the definition for presentation purposes.

Listing 3.9: EME algorithm

```
L ← E_K(0^n)
for i ∈ [1 .. m] do
    PP_i  ← 2^i ⊗ L ⊕ P_i
    PPP_i ← E_K(PP_i)
SP ← PPP_2 ⊕ ··· ⊕ PPP_m
MP ← PPP_1 ⊕ SP ⊕ T
MC ← E_K(MP)
M  ← MP ⊕ MC
for i ∈ [2 .. m] do
    CCC_i ← PPP_i ⊕ 2^(i−1) ⊗ M

SP ← CCC_2 ⊕ ··· ⊕ CCC_m
CCC_1 ← MC ⊕ SC ⊕ T

for i ∈ [1 .. m] do
    CC_i ← E_K(CCC_i)
    C_i ← 2^i ⊗ L ⊕ CC_i
return C_1 ··· C_m
```

In Figure 3.10, the cipher starts and ends with a pre-/post-processing step utilising L. L is subsequently XORed with the plaintext, and at every step L is multiplied with the constant 2 under GF(2^{128}). The result of this preprocessing, PP_i, will be encrypted to $\mathrm{PPP}_i = \mathcal{E}(\mathrm{PP}_i)$. After encryption, all PPP_i are summed up in the Galois Field. In addition, the tweak value T is added.

The summation result MP is encrypted, and forms $\mathrm{MC} = \mathcal{E}_k(\mathrm{MP})$. The addition of MP and MC forms M, which serves as base value for subsequent processing of the remaining cipher block $2 \ldots n$. M is applied to all blocks just like L in the pre- and post-processing step. $\mathrm{M} \otimes 2^n$ is XORed with block n for blocks $2 \ldots n$. From now on, the processing is a mirror-inverted version of the processing done till the application of M. First, an XOR sum is generated and applied to the first block. Then all blocks are encrypted, and finally L is applied to all blocks.

The cipher mode structure is symmetric, thus for decryption, E_k has to be replaced with D_k, and instead of plaintext, ciphertext has to be supplied as input.

Applying the same hash function over and over is in general problematic, because the target domain might disintegrate into short cycles of the hash function. But the Galois Multiplication classifies as universal hash function (see [WC81]), and therefore the iterative multiplication of L or M cannot get caught in a low-order cycle, as GF multiplications are cycle-free. The problem of codomain degeneration of hash functions is investigated closer in Section 5.3.

The algorithm for EME is given in Listing 3.9. In the Implementation Guide (Section 3.10), a different form is presented, which is more suitable for direct implementation. The SISWG draft is available as [Hal04].

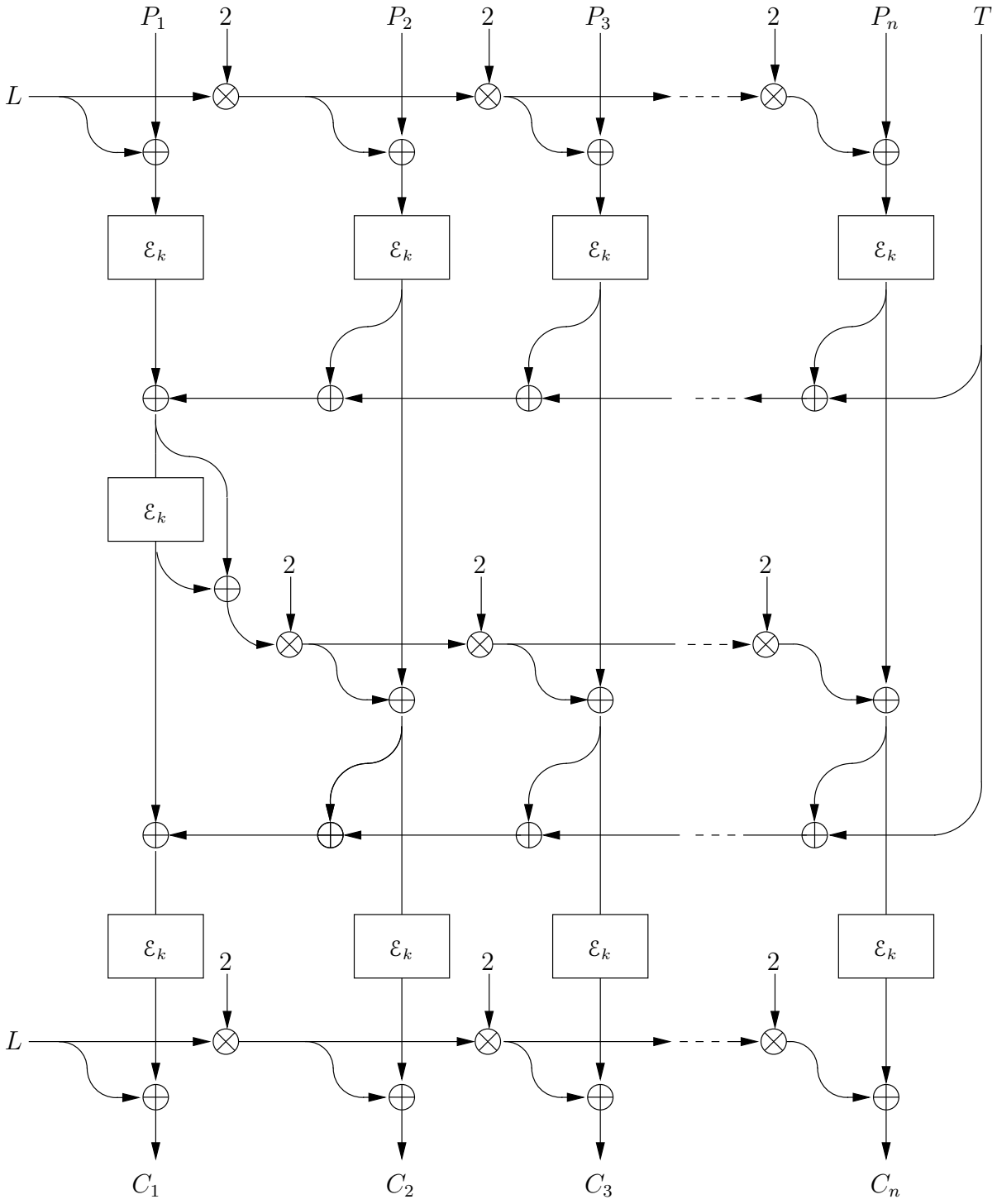


Figure 3.10: EME Mode

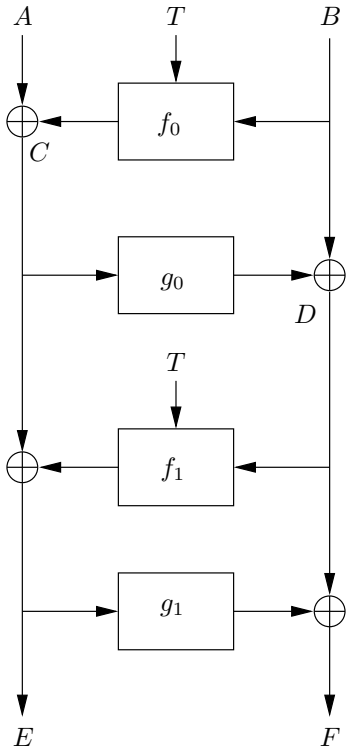


Figure 3.11: ABL4

3.6.3 ABL: Arbitrary Block Length

Nomen est omen, ABL can be configured to form cipher blocks of arbitrary length. We have seen such a flexibility only with stream ciphers that expose the ciphertext to manipulation. ABL does not inherit this property, as it does more than just XORing the plaintext with a key stream. ABL ensures non-malleable encryption with its Luby-Rackoff cipher design. Thanks to this, a bit modification will not be mirrored by a flipped bit in the corresponding plaintext position, but is diffused among the whole plaintext block. Malleability is explained in more detail in Section 4.5.

ABL is specified as ABL3 and ABL4. ABL4 adds an additional round to ABL3 to protect against nonce reuse. We will only describe ABL4, because this property is vital for hard disk encryption. We will also stick to the one-key description given in the SISWG draft [MV04b], instead of the multi-key variant given in the ABL paper [MV04a].

Figure 3.11 depicts ABL4. ABL4 splits the message into two halves, A and B, where A is the first 128-bit block of the plaintext and B is the rest. f_i, g_i are pseudo-random functions that need not be invertible. Notice that the left strand in the figure is 128-bit wide, while the left one has an arbitrary bit length l. Therefore the operands and results of f_i and g_i have different sizes. For simplicity, the following definitions do not include the key dependency.

$$f_i : \{0,1\}^l \times \{0,1\}^t \to \{0,1\}^{128} \qquad g_i : \{0,1\}^{128} \to \{0,1\}^l$$

Before we give definitions for these functions, ABL needs a set of key dependent constants at its disposal. They are generated in a counter like manner,

$$H = \mathcal{E}_K(0^{:128})$$

$$L_1 = \mathcal{E}_K(0^{:127} \,||\, 1) \qquad L_1 = \mathcal{E}_K(0^{:126} \,||\, 10)$$

$$M_0 = \mathcal{E}_K(0^{:126} \,||\, 11) \qquad M_1 = \mathcal{E}_K(0^{:125} \,||\, 100)$$

f_i is defined as

$$f_i(B,T) = \mathcal{E}_K(L_i \oplus \mathrm{GHASH}(B,T))$$

where GHASH is defined as in Section 3.4.5. f_0 and f_1 differ only by the constant L_i they use for the XOR step. As GHASH condenses the input B of arbitrary length to 128-bit, the definition fulfils the requirement, that the function's co-domain has the size 2^{128}. The second parameter T is either a nonce (for ABL3) or a tweak (for ABL4) that can have an arbitrary length t.

The opposite is desired for g_i. A 128-bit item must be expanded to an arbitrary length. This is achieved by generating a counter mode key stream as in Equation (3.5) with an IV computed from the argument and a whitening value M_i.

$$g_i(I) = S(I \oplus M_i)$$

The result of an ABL encryption is the concatenated results $E \,||\, F$ as shown in Figure 3.11.

3.6.4 Other modes

Herein we give a brief description of other cipher modes that are also discussed in SISWG. The reason why we do not elaborate them in more detail is because they are patented and therefore not as attractive for implementation.

CMC: CBC-Mask-CBC

CMC has also been drafted by the duo Halevi and Rogaway and is somewhat of a cousin to EME. It is also a wide cipher mode, but in contrast to EME is not parallelisable, because it uses two steps of CBC processing. CMC has the advantage that existing hardware implementations for CBC can be reused.

After an initial regular CBC processing, a mask is computed from the resulting ciphertext, which is applied to all intermediate cipher blocks. The CBC processing is restarted, but traversing the intermediate ciphertext from back to forth. The tweak value serves as initial vector for both CBC steps, making the cipher mode a tweakable cipher mode. CMC has a security proof in its presentation paper [HR03b].

XCB: Extended Codebook Mode

XCB is an alternative wide cipher mode developed by McGrew and Fluhrer of Cisco Systems, Inc. It also utilises a Luby-Rackoff structure, but in contrast to ABL it uses different functions and has five rounds.

The initial and last round uses an encryption and decryption call for the A half. In round two and four, GHASH is used to propagate material from the B half to the A half. The B half is encrypted at the third round of the Luby-Rackoff cipher utilising the A as IV of a counter mode key stream, just like in ABL. XCB features nearly a single cipher block invocation per block. See also [MF04] or [MF05].

3.7 NIST: Authentication modes

As authentication modes play a minor rule in hard disk encryption, we have a look at only three of them. Two of them have counterparts as authenticated encryption modes. In XECB-MAC we can see how the ideas used in the design of XECB-XOR extend to an authentication mode. The same is true for PMAC, that shares design elements with OCB.

At time of writing, NIST is about to adopt OMAC as MAC algorithm. The authors of the OMAC paper use "OMAC" to denote a MAC family, so NIST decided to use CMAC as name to avoid confusion.

3.7.1 RMAC: Randomized MAC

RMAC is using the CBC-MAC construction scheme, and additionally encrypts the CBC-MAC result with a random key.

$$\text{RMAC-tag} = \mathcal{E}_{K_2 \oplus R}(\text{CBC-MAC}(P)) \,||\, R$$

RMAC was chosen by NIST first, but then NIST backed down from this proposal, as the public comments were in favour of other modes. What is critical about RMAC is the non-constant keying of the cipher primitives. For many recent ciphers, the key setup is a non-trivial time consuming task, hence for every MAC generation, there is also a key setup that has to be done.

A feature of RMAC is that it is "salted". A salt is a binary string that is used in a computation, and is made available with the result, so a trusted party can verify the MAC by using the key material and R. The salt is randomly generated and might change across RMAC invocations, thus the same plaintext will result in different MAC values. The MAC is said to be randomised. This property is shared only by XECB-MAC among the NIST proposals. For more information, the reader is referred to [JJV].

3.7.2 XECB-MAC

Most MAC schemes we discussed so far are a descendants of CBC-MAC. With XECB we encounter a MAC scheme that comes from the class of XOR MACs. In general, an XOR MAC operates by

1. splitting the message into equal sized blocks,
2. applying a pseudo-random function to all blocks,

3. XORing all results to compose the MAC.

The second step must depend on the block number in some way, otherwise parts of the message that are equal will cancel each other out in the XOR summation. This is ok for authenticated encryption modes but not for MACs, see Section 3.4.1.

[GD01] specifies three flavours of XECB-MAC. We concentrate on the stateless sender version solely, as the differences between the three flavours are not important to demonstrate the XECB-MAC operation.

The stateless version chooses

$$\mathrm{PRF}(M, i) = \mathcal{E}_k\big(M_i + i \times \mathcal{E}_k(r_0)\big)$$

as its PRF function. r_0 is a random salt. Padding is also implemented in XECB-MAC, and to distinguish between padded and non-padded message, an artificial plaintext block is created with a value Z or $\overline{Z}$ depending on whether it is padded or not. The value Z is derived from r_0. This MAC is also randomised because of the random element r_0 used in the encryption process. XECB-MAC is specified in [GD01].

3.7.3 PMAC: Parallelizable MAC

PMAC's main advantage is of course its parallelisability. This advantage is inherited from the XOR-MAC construction it uses for most of the data. A preprocessing step adds whitening values to the plaintext blocks. These values are computed by Galois Field multiplications of a position dependent element γ_i with L. L is a constant derived from key material. The sequence γ_i forms a Gray Code. Because a Gray Code guarantees that only one bit changes at a time, the next Galois Field multiplication result can be generated with a single XOR, see Section 2.5.

After whitening, all blocks except the last are encrypted and an XOR sum is computed. The last block is directly added to the XOR sum. If padding is necessary for the last block, $L \otimes x^{-1}$ is added to the XOR sum to distinguish non-padded and padded messages. The final XOR sum is encrypted and returned as tag. PMAC was introduced by Rogaway in [Rog01b].

3.8 Evaluating the modes for Hard Disk Encryption

Until now, the presentation of the cipher modes has been rather general. Hard disk encryption happens in a setting different than communication encryption. While the latter is a transmission between distinct parties, the party does not change for the former. Hard disk encryption can be seen as an inter-time intra-party transmission, in other words, a party communicating with an alter-ego located in the future.

It is assumed that there is no way to save information in the meantime, except for the shared secret key. No counters or intermediate values can be stored. Therefore, hard disk encryption modes have to be stateless cipher modes.

Users of hard disk encryption want to utilise as much space as possible for their personal data, and do not want to devote disk space to disk space

management itself. Most hard disk encryption systems do not generate or save an initialisation vector nor does the majority use authentication tags. But as an initialisation vector is highly desirable and required for most modes, the IV is simply derived from the sector number. Hence, the IV is reused and predictable. The cipher modes employed for hard disk encryption must stay secure for this kind of IVs.

3.8.1 IVs, Nonces and Tweaks

Many papers concerning the cipher modes above use terms like initialisation vector, nonce, tweak, spice, counter, or diversification parameter. In general, these terms refer to a piece of information that is required by the encryption process – just as the key – but does not have to be kept secret.

Despite the IV terms appear to refer to the same concept, this is wrong and they must not used interchangeable. However, it is true that there are strong concepts that can be used in place of weaker ones.

There are two important characteristics for an IV,

Predictability An attacker might use distinguishing attacks against the cipher mode, when a predictable IV is used for a cipher mode that requires an unpredictable one. An example for a distinguishing attack for CBC is given in [Rog04].

Uniqueness A cipher mode might require the IV to be unique. The implications, if this requirement is violated, are different. See below.

We distinguish the following types of initialisation vectors:

Salt A unique unpredictable item. If the salt domain is large enough and no state can be maintained between encryption calls, it is reasonable to generate the salt at random, because the uniqueness requirement is unlikely to be violated.

Nonce A nonce is a **n**umber used only **once** in the context of a key. A nonce is allowed to be predictable.

Tweak A non-random predictable value that can be reused.

IV or initialisation vector general term used for all of the above.

NIST [NIS03b] recommends two methods to generate initialisation vectors for CBC and CFB, either by a FIPS-approved PRNG [CE05], or by calling $\mathcal{E}_k(\text{ctr})$, where ctr is a counter. For OFB, the IV may be predictable (contrary to the other modes above) but has also to be unique. So for OFB, an unencrypted counter is sufficient.

For CFB, OFB and CTR, the reuse of a nonce is fatal. An attacker can simply XOR together both ciphertext streams, and the key stream is cancelled out. The result is the XORed version of two plaintext streams. Any hint on the content of one of the stream will immediately reveal the other stream.

For CTR, the requirements are even more tight, as the nonce is interpreted as number and is incremented very predictably. Therefore, the user must ensure not only that n is distinct from any value used before, but also that n was not

used in the encryption of any previous message. This requirement would be violated, if a two-block message is encrypted with the initialisation vector $n-1$, and another message is encrypted with n.

3.8.2 Evaluation

All of the cipher modes presented by SISWG are suitable for hard disk encryption. This is no surprise as all of them were designed for it. In contrast, any stream cipher is unsuitable for hard disk encryption, because in hard disk encryption the IV is reused, which is forbidden for stream ciphers. Many AEAD modes presented at NIST use CTR as part of the design, hence no such cipher mode is suitable for hard disks.

The remaining NIST ciphers were designed to operate with nonces. As you can see from ABL, a cipher mode that is designed to process nonces is not necessarily secure when the nonce values are reused. For a more theoretic treatment of nonce security with respect to distinguishing attacks the reader is referred to [Rog04].

3.9 Summary of Cipher Modes

Name	Security function	Parallel-isable	IV Require-ments	Cipher calls	Comments
CBC	E	No	Salt	$\lceil \frac{\lvert M \rvert}{n} \rceil$	Many attacks. See Chapter 4.
CTR	E	Yes	Extended Require-ments	$\lceil \frac{\lvert M \rvert}{n} \rceil$	Stream cipher, unsuitable for hard disk encryption.
CFB	E	No	Salt	$\lceil \frac{\lvert M \rvert}{m} \rceil$	Stream cipher.
OFB	E	No	Nonce	$\lceil \frac{\lvert M \rvert}{m} \rceil$	Stream cipher. Secure when run in m-bit mode with $m = n$.
2DEM	E	Yes	None speci-fied	$2\lceil \frac{\lvert M \rvert}{n} \rceil$	Data treated as 2-D structure. Row and Column operations for good diffusion.
ABC	E	No	Salt	$\lceil \frac{\lvert M \rvert}{n} \rceil$	CBC styled, but with plain-text accumulation, hence error-propagation for decryption.
IGE	E	No	Salt	$\lceil \frac{\lvert M \rvert}{n} \rceil$	Error propagation for decryption.
LRW	E	Yes	Tweak	$\lceil \frac{\lvert M \rvert}{n} \rceil$	SISWG candidate. Narrow.
EME	E	Yes	Tweak	$2\lceil \frac{\lvert M \rvert}{n} \rceil + 1$	SISWG candidate. Wide. Patented.
CMC	E	No	Tweak	$2\lceil \frac{\lvert M \rvert}{n} \rceil + 1$	CBC-Mask-CBC construction. Patented. Wide.

continued on next page

Name	Security function	Parallel-isable	IV Require-ments	Cipher calls	Comments
ABL3	E	Yes	Nonce	$\lceil \frac{\lvert M \rvert}{n} \rceil + 1$	Three round Luby-Rackoff cipher. Wide.
ABL4	E	Yes	Tweak	$2\lceil \frac{\lvert M \rvert}{n} \rceil$	Four round Luby-Rackoff cipher. Wide.
XCB	E	Yes	Tweak	$\lceil \frac{\lvert M \rvert}{n} \rceil + 1$	Patented. Wide.
CCM	AEAD	No	Nonce	$2\lceil \frac{\lvert M \rvert}{n} \rceil + 2$	Encrypt-then-Authenticate construction: CTR, CBC–MAC. Message size must be known in advance, hence offline. NIST and IEEE 802.11 standard.
EAX	AEAD	No	Nonce	$2\lceil \frac{\lvert M \rvert}{n} \rceil + 2$	ETA-constr.: CTR, OMAC
CWC	AEAD	Yes	Nonce	$\lceil \frac{\lvert M \rvert}{n} \rceil + 3$	ETA-constr.: CTR, CW-MAC
GCM	AEAD	Yes	Nonce	$\lceil \frac{\lvert M \rvert}{n} \rceil + 2$	ETA-constr.: CTR, GHASH
IACBC	AE	No	Salt	$\lceil \frac{\lvert M \rvert}{n} \rceil + 1 + \lceil \log_2 \frac{\lvert M \rvert + n}{n} \rceil$	CBC core with plaintext whitening. XOR-MAC for authentication generating. Patented.
IAPM	AE	Yes	Salt	$\lceil \frac{\lvert M \rvert}{n} \rceil + 1 + \lceil \log_2 \frac{\lvert M \rvert + n}{n} \rceil$	Patented.
OCB	AE	Yes	Nonce	$\lceil \frac{\lvert M \rvert}{n} \rceil + 2$	Patented.
PCFB	AE	No	Salt	$\lceil \frac{\lvert M \rvert}{n} \rceil + 2$	Mix of CFB with OFB. Infinite error propagation, authentication via AREA.
XCBC-XOR	AE	No	Salt, Nonce for some variants	$\lceil \frac{\lvert M \rvert}{n} \rceil + 1$	CBC with post-whitening. Stateless and stateful variants. Patented. Integer arithmetic.
XECB-XOR	AE	Yes	Salt, Nonce for some variants	$\lceil \frac{\lvert M \rvert}{n} \rceil + 1$	Stateless and stateful variants. Patented. Integer arithmetic.
XCBC-MAC	A	No	None	$\lceil \frac{\lvert M \rvert}{n} \rceil + 1$	Foundation for TMAC and OMAC.
OMAC	A	No	None	$\lceil \frac{\lvert M \rvert}{n} \rceil + 2$	XCBC core. Candidate for NIST standard under the name CMAC.
PMAC	A	Yes	None	$\lceil \frac{\lvert M \rvert}{n} \rceil + 1$	XOR-MAC. Gray-code GF multiplications for whitening. Patented.
RMAC	A	No	Salt	$\lceil \frac{\lvert M \rvert}{n} \rceil + 1$	Randomised. CBC-MAC core.
XECB–MAC	A	Yes	Salt (nonce for some variants)	$\lceil \frac{\lvert M \rvert}{n} \rceil + 1$	XOR-MAC with integer whitening. Randomised. Patented.

Listing 3.12: LRW encryption

```
LRW-enc(P,T):
    GFMulSeq(mulTweaks, T, m)  // The lookupTable used in GFMulSeq
                                  is computed for L = E_k(0^n)

    for i in [1 .. m] do
        C_i → E_k(P_i ⊕ mulTweak_i) ⊕ mulTweak_i
    return C_1 ··· C_m
```

Listing 3.13: optimised EME algorithm, Encryption

```
L ← E_K(0^n)
MP ← T
for i ∈ [1 .. m] do
    L    ← 2 ⊗ L
    PP_i ← L ⊕ P_i
    PPP_i ← E_K(PP_i)
    MP   ← MP ⊕ PPP_i
MC ← E_K(MP)
M  ← MP ⊕ MC

CCC_1 ← MC ⊕ T
for i ∈ [2 .. m] do
    M ← 2 ⊗ M
    CCC_i ← PPP_i ⊕ M
    CCC_1 ← CCC_1 ⊕ CCC_i

L ← E_K(0^n)
for i ∈ [1 .. m] do
    L ← 2 ⊗ L
    CC_i ← E_K(CCC_i)
    C_i ← CC_i ⊕ L
return C_1 ··· C_m
```

3.10 Implementation Guide

We give two example implementations for LRW and EME. These cipher modes were chosen because they are the favourites in the standardisation process of SISWG.

With the algorithms developed in Section 2.2 it is trivial to implement a high-performance LRW implementation, see Listing 3.12. As LRW is symmetric for decryption, the decryption function can be derived by replacing $\mathcal{E}_k$ with $\mathcal{D}_k$, and switching P_i and C_i.

Concerning EME, the algorithm given in Listing 3.9 is not stated in an optimal way for implementation. Listing 3.13 gives a version more suitable for direct implementation. Notice that this algorithm is parented.

Chapter 4

CBC attacks

CBC is the most common cipher mode for hard disk encryption. Usually the disk is divided into parts that are run in CBC mode separately. This is not surprising, because if the hard disk would be run as a single large CBC stripe, the random access nature of a hard disk would be lost. The encryption process is recursive in CBC, so the encryption of the n^{th} block depends on the encryption of all preceding blocks, 0 till $n-1$. A single character change in the first sector of a disk would causes an avalanche effect for the whole following ciphertext, requiring a write operation not only to change a single block but all following disk sectors. Of course this is an undesired property. Hence, the CBC chaining is broken between every disk part and is restarted with a new IV.

These disk parts are an atomic unit in terms of write operations. Choosing the disk parts to have the size of a sector will match with the smallest unit of hard disks, where a sector is also atomic in terms of writing.

We restate the CBC encryption and decryption definition from Chapter 3. Note that decryption is not recursive, in contrast to encryption, since it is a function only of C_{n-1} and C_n.

$$\text{Base:} \qquad C_0 = IV \tag{4.1}$$

$$\mathbf{E}^{CBC} : \mathbb{P}^n \to \mathbb{C}^n \qquad C_i = \mathcal{E}_k(P_i \oplus C_{i-1}) \tag{4.2}$$

$$\mathbf{D}^{CBC} : \mathbb{C}^n \to \mathbb{P}^n \qquad P_i = C_{i-1} \oplus \mathcal{D}_k(C_i) \tag{4.3}$$

When we use a design, where the CBC chaining is cut, then we have multiple IVs. In the following, C_1 always refers to the first block in a sector, so the indices are always relative to the sector start.

To fully specify an implementable encryption scheme, the way C_0 is chosen for every sector must also be defined. As mentioned before, C_0 is a function of the sector number, but there are many ways to state such a function. The next sections will deal with how C_0 is chosen, shading light on the surprisingly grave security implications of this decision.

4.1 Initialisation Vector Modes

We define an IV mode to be *public*, if the function for generating the IV from the block number does not depend on key material. All modes depending on

key material are *private*.

4.1.1 plain-IV

The plain-IV initialises C_0 with the sector number. To ensure compatibility among different architectures, it is assumed that the IV is simply the 32-bit version of the number n encoded in little-endian padded with zeros, if necessary. This is the most simple IV mode, but at the same time the most vulnerable[1].

$$\text{IV}(\text{sector}) = \text{le32}(\text{sector})$$

This mode is a public IV mode.

4.1.2 ESSIV

The name ESSIV is derived from "**E**ncrypted **S**alt-**S**ector **IV**". It is defined as

$$\text{IV}(\text{sector}) = \mathcal{E}_{\text{salt}}(\text{sector}) \qquad \text{where salt} = H(K)$$

ESSIV derives the IV from key material via encryption of the sector number with a hashed version of the key material, the salt[2]. ESSIV does not specify a particular hash algorithm H, but the digest size of the hash must be a valid key size for the block cipher. For instance, sha256 is a valid choice for AES, as sha256 produces a 256-bit digest.

Because the IV depends on a private piece of information, namely the key, the actual IV of a sector cannot be deduced by an attacker. Thus, the IV mode is private.

This mode was developed by the author of this work for Linux 2.6.10 to counter attacks watermarking attacks.

4.1.3 Plumb-IV

Changing a single byte in an encryption block will cause all following encryption blocks to be different thanks to the recursive nature of CBC encryption. Plumb-IV[3] is designed to incorporate a kind of backward avalanche affect, so that a single byte change in the last plaintext block will effect the first cipher block.

This can be done by hashing (or MACing) the plaintext from the second block to the last and using its value as IV. If a byte changes in these plaintext blocks, the first block is influenced by the change of the IV. As the first encryption affects all subsequent encryption steps, the whole sector is changed.

$$\text{IV} = H(P_2 \,||\, P_3 \,||\, \ldots \,||\, P_n) \tag{4.4}$$

Decryption is possible, because CBC is not recursive for decryption. The prerequisites for a successful CBC decryption are two subsequent cipher blocks,

[1] This scheme was the first implemented in the Linux kernel 2.2.

[2] This is a slight misuse of the term "salt", as a salt is usually generated randomly. Here it is deduced from key material.

[3] The name of this mode comes from Colin Plumb, who proposed this mode on the Linux Kernel Mailing List. Bruce Schneier also mentions this construction in [Sch96] p. 224, but does not give a name for it, so we will refer to it as Plumb-IV.

see (3.3). First, all plaintext blocks from 2 to n are recovered. This is possible as $C_1 \dots C_n$ is all what is needed for $P_2 \dots P_n$.

$$\begin{aligned} P_2 &= \mathcal{D}(C_2) \oplus C_1 \\ P_3 &= \mathcal{D}(C_3) \oplus C_2 \\ &\dots \\ P_n &= \mathcal{D}(C_n) \oplus C_{n-1} \end{aligned}$$

Then, the IV can be recovered by recomputing the hash from the plaintext blocks $P_2 \dots P_n$, according to (4.4), exactly as it was during encryption. Having recovered the IV – that is C_0 – the first block can be decrypted regularly,

$$P_1 = \mathcal{D}(C_1) \oplus C_0$$

The decryption is therefore possible and complete.

In addition to the plaintext content, the sector number should be used for hashing. Otherwise, equal plaintext blocks will yield equal ciphertext blocks on disk. To counter attacks based on the public IV characteristic, key material should also be used in the hashing process.

A major shortcoming of this scheme is its performance. It has to process data twice, first for obtaining the IV, and then for producing the CBC encryption. Two-pass processing itself is not the problem, but with the same performance penalty other schemes are able to achieve better security properties, because Plumb-IV inherits all problems associated with CBC. Therefore, Plumb-IV seems to be no attractive candidate for implementation.

Having discussed all major IV modes, we now have a look at the attacks. Except for the watermarking attack, all attacks are mountable regardless of the choice of the IV mode. The severity of the attack might be marginally different.

4.2 Content leak attack

This attack can be mounted against any system operating in CBC Mode. It relies on the fact that in CBC decryption, the preceding cipher block's influence is very simple. The preceding block C_{i-1} is XORed into the plaintext before encryption. This block is readily available on disk (for $i > 0$) or may be deduced from the IV (for $i = 0$)[4]. If an attacker finds two blocks, i and j, with identical ciphertext, he knows that both ciphertexts have been formed according to:

$$\begin{aligned} C_i &= \mathcal{E}_k(P_i \oplus C_{i-1}) \\ C_j &= \mathcal{E}_k(P_j \oplus C_{j-1}) \end{aligned}$$

Since he found that $C_i = C_j$, it holds

$$P_i \oplus C_{i-1} = P_j \oplus C_{j-1}$$

which can be rewritten as

$$C_{i-1} \oplus C_{j-1} = P_i \oplus P_j \tag{4.5}$$

[4]Only when using a "public-IV" scheme

The left hand side is known to the attacker by reading both preceding ciphertexts from disk[5]. The attacker is now able to deduce the difference between the plaintexts by examining the difference of C_{i-1} and C_{j-1}. If one of the plaintext blocks happens to be zero the difference yields the original content of the other plaintext block.

Let's have a look at the chance of succeeding with this attack.

4.2.1 Probability

Let n be the number of possible cipher blocks out of $\mathbb{C}$, and k the number of blocks chosen randomly. When we are using an m-bit cipher, $n = 2^m$. We define $P(k, n)$ as the probability that a sample with k element of $\mathbb{C}$ with $|\mathbb{C}| = n$ results in one without duplicate elements. This probability function has an obvious feature

$$P(1, n) = 1$$

as the first pick from $\mathbb{C}$ cannot result in a collision. In order that no collision occurs, the second block must be chosen from $n - 1$ elements of $\mathbb{C}$, as the first pick occupies one element. The third block is restricted to $n - 2$ choices, the forth to $n - 3$ choices, and so on. The expression continues with a form $n - k$ when a block is added to k existing blocks.

When these blocks are chosen randomly, we can give the probabilities that adding a block to a sample results in a collision-free sample. Each addition pick must be one of $n - k$ elements from a set with n elements in total. The probability for this to occur is $\frac{n-k}{n}$. With this, we can give a recursive definition of $P(k, n)$

$$P(k + 1, n) = \frac{n - k}{n} P(k, n)$$

This gives the regular pattern

$$\frac{n-1}{n}\frac{n-2}{n}\cdots\frac{n-k+1}{n}$$

which we can also define directly with

$$P(k, n) = \frac{1}{n^k} \frac{n!}{(n-k)!} \tag{4.6}$$

As this is the collision-free probability, a collision occurs at $1 - P(k, n)$. The factorial is not easily computable for large n, therefore we use Stirling's approximation

$$n! \approx \sqrt{2\pi\, n}\Big(\frac{n}{e}\Big)^n$$

Substituting $n!$ in (4.6) gives

$$P(k, n) \approx \frac{\sqrt{2\pi\, n}\,\big(\frac{n}{e}\big)^n}{\sqrt{2\pi\,(n-k)}\,\big(\frac{n-k}{e}\big)^{n-k}}\, n^{-k}$$

[5] If one of the blocks is the first block of a sector, the IV must be examined instead, when it is available as it is with a public IV.

Applying ln and reducing the fraction gives

$$\ln P(k,n) = \ln \left[\frac{n^{n-k+\frac{1}{2}}}{(n-k)^{n-k+\frac{1}{2}}} e^k \right]$$

and pulling e^k and the exponent of the fraction out of the logarithm gives

$$\ln P(k,n) = (n-k+\frac{1}{2}) \ln \left[\frac{n}{n-k} \right] - k \tag{4.7}$$

This calculations rely on [Har05].

4.2.2 Numbers

To get a feeling for the magnitudes involved in this attack, we present a few numerical examples.

For instance, let the disk size be 200GB, and the cipher setup be CBC using AES as cipher and plain-IV as IV mode. AES is a 128-bit cipher, therefore $|\mathbb{C}| = n = 2^{128}$. In a disk sector, there are 32 cipher blocks, and when using public-IV, all cipher blocks have their preceding cipher blocks available (the preceding cipher block for the first block C_1 is the IV C_0). Thus all blocks are candidates for a collision and in such a case will disclose information. If a non-public IV mode is used, the cipher blocks in a sector useful for an attack is reduced to 31, as C_0 is not available for C_1.

The number of cipher blocks available on a 200GB disk with known C_{n-1} is 200GB $*$ 1024KB/GB $*$ 64blocks/KB[6]. This is equal to the number of 128-bit blocks in a 200GB hard disk. Hence, $k \approx 1.342 * 10^{10}$. As the extra precision is of little value, we will examine the probability for exponents 10^{10} and following. Filling this into the equation (4.7),

$$P(k,n) = e^{(\frac{1}{2}+n-k) \ln \frac{n}{n-k} - k}$$

we calculate,

k	$P(k, 2^{128})$
10^{10}	1.47×10^{-19}
10^{11}	1.47×10^{-17}
10^{12}	1.47×10^{-15}
10^{13}	1.47×10^{-13}

For comparison, the chance of guessing the correct 256-bit key with a single guess has a probability of 10^{-77}. We find that an attacker has a much better chance of finding confidential information by searching for block collisions. Of course, the difference is that a compromised key will reveal all information, while a collision reveals a random piece of information. In fact, the revealed information is meta-information, that is the XOR difference between two plaintext blocks. An attacker must have an idea of the content of block to extract

[6]For all non-public IV schemes, i.e. ESSIV/plumb IV, the ratio is 62blocks/1KB.

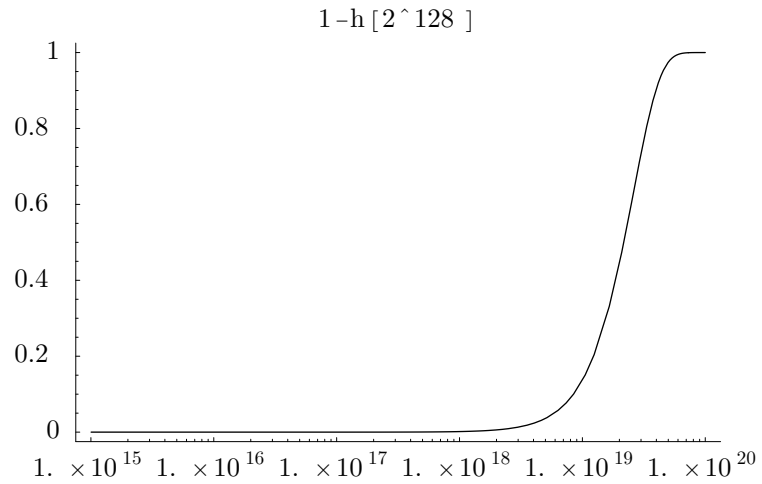


Figure 4.1: The inflexion point of the collision function

information about the other. So, despite the improved probability for this attack path, the security implication are not as grave.

After having seen a few examples, we will investigate the growth of this function.

4.2.3 Inflexion point

Figure 4.1 gives a logarithmic plot of $1 - P(k, n)$ for k. An obvious feature of this figure is the jump of the probability around 10^{19}. We will call the point with the largest growth the *inflexion point*. Translated to storage space, this inflexion point is reached for a 145 million terrabyte storage.

The point of the biggest growth can also be found numerically. It is at 1.844×10^{19}. But this is just equal to $\sqrt{2^{128}}$, which is also known as birthday boundary. Why do we encounter the birthday paradox here?

The reason is simple. Both, the birthday paradox and the content leak attack, occur because of collisions among values chosen from a single set. For the birthday attack, those values are birthdays drawn from the set of dates in year (hence its name). In our collision scenario, it is ciphertext drawn from the exhaustible set of ciphertext. So, the collision problem and the birthday attack are the elementary problem.

An attacker can relay on the mathematical fact, that is very likely to see duplicate keys – when chosen randomly – after having seen $\sqrt{2^n}$ keys (n is the key size). A more in-depth treatment of the birthday paradox is given in [BK04].

4.2.4 Extending the attack

Another information is available to the attacker. Any succeeding identical pair of ciphertext, that follows the initial identical cipher pair, hints that the corresponding plaintext is equal. The reason is simple. Assume the attacker has found $C_i = C_j$, and the attacker finds $C_{i+1} = C_{j+1}$ in addition, then he

knows from (4.5) that

$$C_i \oplus C_j = P_{i+1} \oplus P_{j+1}$$

holds for the latter pair. But the left side of the equation is zero, therefore the right side has to be zero too. Hence, $P_{n+1} = P_{m+1}$. This reasoning can be extended to cases $C_{i+2} = C_{j+2}$, and so on.

4.3 Watermarking

The probability calculated above is for an unintended block collision. This section assumes the IV mode is public. With such a mode, the IV is precomputable by an attacker, and we will see that due to this a cipher block collision can be created artificially. By means of collisions, watermarks are created that can be used to tag data. The data becomes detectable without the need for the correct cipher key.

Before we discuss this attack, we extend our notation for cipher- and plaintext block numbering. Using an m-bit cipher, there are $(512 \times 8)/m$ cipher blocks in a sector. In this section, $C_{i,k}$ shall denote the cipher block stored in sector i and which has a relative position $k-1$ to the sector start measured in m-bit cipher blocks. $C_{i,0}$ is the IV for sector i.

Assume the attacker writes to two blocks i and j, then he can deduce the IVs by computing $\mathrm{IV}(i)$, $\mathrm{IV}(j)$. The attacker is also aware that by (3.2), the plaintext is XORed with the IV prior to encryption. An artificial cipher block collision can now be generated by,

$$\begin{aligned} \mathcal{E}_k(P_{i,1} \oplus C_{i,0}) &= \mathcal{E}_k(P_{j,1} \oplus C_{j,0}) \\ \mathrm{IV}(i) \oplus P_{i,1} &= \mathrm{IV}(j) \oplus P_{j,1} \\ P_{j,1} &= \mathrm{IV}(j) \oplus \mathrm{IV}(i) \oplus P_{i,1} \end{aligned}$$

When $P_{j,1}$ is written to sector j after it has been encrypted to $C_{j,1}$, the first cipher block in sector j will be equal to the one of sector i, $C_{i,1}$.

No additional information about any plaintext can be extracted by an attacker by the means of this cipher collision as outlined in the last section. This is because the attacker created $P_{j,1}$ on his own, and therefore has to know $P_{i,1}$. What is gained by the actions in this section is that an attacker can find data tagged by him on an encrypted disk. What might be surprising is that this attack is independent of the cipher in use and independent of the key in use. This is called watermarking.

The cipher block chaining spans across the whole sector, and as we found in Section 4.2.4, the encrypted content of both sectors are equal, when all plaintext blocks following $P_{i,1}$ and $P_{j,1}$ are equal. This can be provoked by an attacker easily. But an attacker might choose a point k from which on $P_{i,k}$ and $P_{j,k}$ start to differ, i.e. $C_{i,k}$ and $C_{j,k}$ differ.

Given a 128-bit cipher and assuming 4096-bit sectors, there are 32 valid choices for k. Hence, an attacker can vary among 32 different watermarks. This is about 5 bits of external visible information, that can be access without the cipher key.[7] By using multiple watermark, an attacker might be able to encode more information.

[7] This is not exactly 5 bits of information, because the attacker needs the first block as start marker, as the absence of a watermark cannot be used for information encoding.

What kind of confidentiality is violated when this attack requires the attacker to know the plaintext in advance? This is a valid question. The data itself cannot be the cause for a confidentiality violation here, as the attacker already must know the plaintext. This attack allows an attacker to proof the existence of data. This information is meta data, information about data.

This can become relevant for the user of an encryption system that contains copyrighted material, and the user has copied the material without permission. An attacker can use the watermark evidence to proof the copyright violation, even without knowing the encryption key. The presumption of innocence will quickly fade in court, when the party asserting a copyright infringement can produce an RSA signature extracted from watermarks found on the suspects disk.

Hence, the user has a valid interest that not only the information is confidential, but also the information about the existence of previously tagged data is confidential. This is a typical case, which is covered by the class of distinguishing attacks. As we mentioned in Section 3.4, a security proof tries to show that a cipher mode is indistinguishable from a random permutation. What we have seen with the watermark attack is actually the implementation of a distinguishing attack, as it is considered a security hazard, if an attacker can distinguish the encryption of the specially crafted plaintext from a random permutation.

Having looked at the practical implications of this attack, we will demonstrate, how easily the attack can be implemented with plain-IV. A practical problem for an attacker is to obtain the sector numbers i, j. If an attacker does not place the watermarked data on disk by himself, he has little chance to control the process of copying. Hence, even if any possible IV can be deduced by knowing the sector number, it is simply not implementable in the real world, as the attacker has no control over the sectors his prepared data ends up in.

4.3.1 Attacking the plain IV scheme

What is more likely to occur in a real system is that subsequent parts of a file are written to subsequent sectors. For instance, the ext2 file system used under Linux features a 4K block size by default, and many other file system do it alike, as it is simply not economical to do disk space management on sector level. Thus, writing 4K data to a file results in at least 4 sectors put on disk unfragmented.

With plain-IV, the IV progresses with a much more foreseeable pattern. Given the sector number n, we examine the bitwise difference from n to $n+1$ and $n+2$. Assume the least significant bit of n is 0. Then $(n+1) \oplus n = 1$. If the least significant bit (LSB) n_0 of n is not zero, then the LSB of $n+1$ has to be zero, and it follows $(n+1) \oplus (n+2) = 1$. For $(n+1)$, either the backward bitwise difference or the forward bitwise difference is 1.

Given an arbitrary plaintext block P, it is possible to let P encrypt to the two identical ciphertext blocks, when we can store encrypted plaintext to the first plaintext blocks of three adjacent sectors. To achieve this, we create P' according to

$$P' = P \oplus 1$$

and write three sectors with P', P, P' as their first plaintext blocks. From the last paragraph we know, that either the backward bitwise difference for $(n+1)$ is 1 or the forward bitwise difference. If we assume $(n+1)$ to be the number of the sector in the middle, we can demonstrate this by investigating two disjoint cases.

Assuming $\mathrm{LSB}(n) = 0$, then we have

$$\begin{aligned} \mathcal{E}_k\big(\mathrm{IV}(n) \oplus P'\big) &= \mathcal{E}_k\big(\mathrm{IV}(n+1) \oplus P\big) \\ \mathcal{E}_k\big(n \oplus P \oplus 1\big) &= \mathcal{E}_k\big(n \oplus 1 \oplus P\big) \end{aligned}$$

Assuming $\mathrm{LSB}(n+1) = 0$, we conclude

$$\begin{aligned} \mathcal{E}_k\big(IV(n+1) \oplus P\big) &= \mathcal{E}_k\big(IV(n+2) \oplus P'\big) \\ \mathcal{E}_k\big((n+1) \oplus P\big) &= \mathcal{E}_k\big((n+1) \oplus 1 \oplus P \oplus 1\big) \\ \mathcal{E}_k\big((n+1) \oplus P\big) &= \mathcal{E}_k\big((n+1) \oplus P\big) \end{aligned}$$

This shows that either the first cipher blocks of sector n and $n+1$, or $n+1$ and $n+2$ are equal. The information encoding technique by choosing a break point in the sequence of collision can be applied here to encode sector-size/cipher-block-size different watermarks. By using several watermarks, probably more information is encodeable.[8]

To find the watermark, an attacker can search the disk sector-wise, comparing the first block of the sector with the first block of the preceding sector. If they match, the watermark is found. As shown in Section 4.2, the probability for a random match is so small, that it is reasonable to assume that this is not a coincidence.

The disk search can be done in a single pass, and is much more feasible than finding two identical blocks, that are scattered on the disk as assumed in Section 4.2. A complete description of watermarking can be found in [Saa04]. This attack has been demonstrated to work in a real implementation. The attack can be defeated by using ESSIV or by totally diverting from CBC.

4.4 Data modification leak

CBC encryption is recursive, so the n^{th} block depends on all previous blocks. A change in a preceding block causes the corresponding and all subsequent cipher blocks to change unpredictably. This property is called error-propagation. But the other way round – changes to n^{th} causing changes to $n-1 \ldots 1$ – would also be nice. Why?

The weakness becomes visible, if storage on a remote computer is used, or more likely, the hard disk exhibits good forensic properties. The point is, when the attacker has access to preceding (in time) ciphertext of a sector, either by recording it from the network, or by using forensic methods, he can guess data modification patterns by examining the historic data. If a sector is overwritten with a partially changed plaintext, there is an amount of bytes at the beginning, which are unchanged. This point of change[9] is directly reflected

[8] Assuming an ideal environment, where no other watermarks are present on the disk, no incidental cipher block collision occurs, and no data is duplicated by the file system by defragmentation or other disk management activities

[9] aligned to the cipher block size boundaries

in the ciphertext. Hence, an attacker can deduce the point of the change in plaintext by finding the point, where the ciphertext starts to differ.

This weakness can only be cured when the encryption is randomised. A general approach to probabilistic encryption is given in [GM84]. In brief, a random salt is added to the encryption result, making the ciphertext larger than the plaintext. With this expansion, it is possible to map the same plaintext to multiple ciphertexts. So, an attacker can gain no information about the modifications made. He cannot even distinguish a plaintext change from a simple plaintext rewrite.

Without storing additional salt information, the encryption function becomes bijective. Pairs of identical plaintext will always encrypt to identical ciphertext. Omitting a salt is more convenient for hard disk encryption systems, because access of the sector i of the virtual encryption can be mapped to the access of sector i on the backing device. Also, no disk space has to be reserved to store a salt.

Without a salt, there are several approaches to fix this information leak at least partially. Plumb-IV is an option for CBC because of the backward propagating nature of encryption Plumb-IV installs with its IV calculation. Other approaches are wide-block cipher modes as introduced by SISWG. An attacker will still be able to distinguish a rewrite and a plaintext modification, but he will only be able to do that at sector level. So, the sector is not a transparent structure anymore, as in regular CBC or in any narrow SISWG cipher mode. ABL can be used to create larger structures than block size that are opaque, but the price for this is that all sectors of these larger structures have to be accessed when a single sector is read or written.

4.5 Malleability

Non-malleable cryptography is introduced in [DDN98]. It is an extension of the already strong notation of semantically secure cryptography as defined by [GM84].

Informally, malleability is defined, that given a ciphertext $\mathbf{E}(\alpha)$, the chance of generating a different ciphertext $\mathbf{E}(\beta)$ does not change, where the plaintext β is related to α a preliminarily chosen way, $R(\alpha, \beta)$.

The formal definition given in the original paper is too broad to show the malleability of CBC. Therefore, we will narrow it down to a more specific non-stochastic variant.

Definition 4 *A cipher mode is called malleable, if there exists a function f on ciphertexts, a function g on plaintexts and a range $\langle k, i\rangle$ such that for all ciphertexts C,*

$$D(f(C))\langle k, i\rangle = g(D(C))\langle k, i\rangle$$

The identity functions for ciphertext and plaintext are too trivial to proof malleability.

It can be easily verified that the formal definition above fulfils the informal probabilistic definition of the previous paragraph, because if there is a function set (f, g), the probability of generating a related plaintext defined by g becomes 1, if f is applied to the ciphertext.

The essence of our definition is when a modifications is made by f, the changes can be tracked in the decryption process. A cipher tries to prevent such tracking deliberately, so it should also be prevented by the cipher mode design.[10] The function g is used to reflect the knowledge, that can be gained by tracking the differences. Proofing that there is a single pair (f, g) is sufficient to mark a cipher mode as malleable. But as our definition is more narrow than the original, there are other malleable cipher modes that are not matched by our definition.

4.5.1 The malleability of CBC

CBC is a malleable encryption scheme. The decryption structure of CBC is the source of this weakness. As we can see in its definition, CBC decryption depends on C_{k-1}. An attacker can flip arbitrary bits in the plaintext P_k by flipping bits in C_{k-1}. More formally[11], if

$$C = \mathbf{E}^{CBC}(P)$$

then a pair of functions (f, g) can be given as

$$f(C_1 \mid\mid ... \mid\mid C_n) = C_1 \mid\mid ... \mid\mid C_{k-1} \oplus M \mid\mid ... \mid\mid C_n$$

and

$$g(P_1 \mid\mid ... \mid\mid P_n) = P_1 \mid\mid ... \mid\mid P_{k-1} \mid\mid P_k \oplus M \mid\mid ... \mid\mid P_n$$

that predicts the changes correctly at position k (for a single block $i = 1$). To proof this assertion in more detail, we investigate how P_k is decrypted by (3.3),

$$P_k = C_{k-1} \oplus \mathcal{D}(C_k)$$

If C_{k-1} is modified by f to be $C'_{k-1} = C_{k-1} \oplus M$, then the decryption for P_k will be changed to:

$$P'_k = C'_{k-1} \oplus \mathcal{D}(C_k) = \underbrace{C_{k-1} \oplus \mathcal{D}(C_k)}_{P_k} \oplus M = P_k \oplus M$$

The plaintexts in the intervals $[1, k-2]$ and $[k+1, n]$ are unchanged. Hence, g can predict them trivially from P. What is not predicable is P_{k-1}, because it is partially deduced from decrypting the modified ciphertext C'_{k-1}. Every underlying cipher with a reasonable diffusion will propagate a single bit change among the whole plaintext block in the decryption process. Thus, P_{k-1} is not predicable and will in practise produce a pseudo-random output.

The first block of the CBC ciphertext stream is also malleable, when the IV can be modified by an attacker. This may be the case for communication data streams. For hard disks this cannot happen, since the IV is usually derived from the sector number. The attacker can only influence the IV by moving the data.

[10] Cipher attacks based on plaintext modifications are used in differential cryptanalysis.
[11] The IV parameter for $\mathbf{E}^{CBC}$ has been intentionally omitted.

4.6 Movable cipher blocks

An attacker can move, swap, and copy plaintext as he likes in a CBC ciphertext vector. CBC decryption depends on two variables, C_{i-1} and C_i. Both can be manipulated by an attacker.

To make meaningful manipulations, an attacker has to replace the pair C_{i-1} and C_i with another ciphertext pair, C_{j-1} and C_j. Three plaintext blocks will be affected, P_{i-1}, P_i and P_{i+1}, as C_{i-1} and C_i are used in the decryption of these three blocks. Originally, the plaintext block are decrypted according to:

$$\begin{aligned} P_{i-1} &= \mathcal{D}(C_{i-1}) \oplus C_{i-2} & \qquad P_{j-1} &= \mathcal{D}(C_{j-1}) \oplus C_{j-2} \\ P_i &= \mathcal{D}(C_i) \oplus C_{i-1} & \qquad P_j &= \mathcal{D}(C_j) \oplus C_{j-1} \\ P_{i+1} &= \mathcal{D}(C_{i+1}) \oplus C_i & \qquad P_{j+1} &= \mathcal{D}(C_{j+1}) \oplus C_j \end{aligned}$$

Suppose C_{i-1} and C_i are substituted by C_{j-1} and C_j. Before we have a look at the consequences of this modification for the plaintexts P_{i-1}, P_i, P_{i+1}, we assert that from the equations above the following relations can be derived:

$$\mathcal{D}(C_{i+1}) = P_{i+1} \oplus C_i \qquad \mathcal{D}(C_{j-1}) = P_{j-1} \oplus C_{j-2}$$

After replacing C_{i-1} with C_{j-1} and C_i with C_j, the affected plaintext blocks decrypt according the following equations,

$$\begin{aligned} P'_{i-1} &= \mathcal{D}(C_{j-1}) \oplus C_{i-2} &&= P_{j-1} \oplus C_{j-2} \oplus C_{i-2} \\ P'_i &= \mathcal{D}(C_j) \oplus C_{j-1} &&= P_j \\ P'_{i+1} &= \mathcal{D}(C_{i+1}) \oplus C_j &&= P_{i+1} \oplus C_i \oplus C_j \end{aligned}$$

As you can see the plaintext block i is replaced by the plaintext block j. The other modifications cannot be suppressed by an attacker.

This attack is also known as copy&paste attack. It can also be used to read ciphertext the attacker is not authorised to do. Assume the attacker is a legitimate user of a multi-user system, and that the attacker has obtained access to the physical storage. He can copy ciphertext from a protected file to sectors allocated to a file the attacker has access to. By accessing his file, he will be able to read ciphertext he is not authorised to read. This attack does not work with other cipher modes – i.e. LRW, EME, ABL, just to name a few.

4.7 Threat models

Common goals of data and communication security are:

Confidentiality, the plaintext is not disclosed to anyone, except to the party possessing the correct secret.

Integrity, the plaintext has not been modified, or the modification is detectable.

Authenticity, the source of the plaintext can be doubtlessly identified. This goal requires an unaltered message, hence this goal depends on integrity.

Confidentiality deserves a bit of clarification. As we have seen in the watermarking attack, the existence of specially crafted plaintext is detectable by an attacker without the encryption key. Certainly, some kind of confidentiality requirement is violated, but in contrast to a regular confidentiality violation, no content is revealed to the attacker, as he must know the crafted plaintext to detect it. What is revealed here is not data – as it is already known – but meta data, that is the existence of data.

To discuss threats arising from this special confidentiality violation, we define

Tag Confidentiality, the plaintext remains undisclosed to everyone, except to the party possessing the common secret. The plaintext is not recognisable even if it has been tagged by an attacker.

Further situations in which an attacker might gain information about the encryption system are:

Cold read: The data storage is exposed to an attacker, when it is not in use. For instance, this might be physical theft of the hard disk, or prosecution of law enforcement agencies. The security properties relevant for this threat model are confidentiality and tag confidentiality. Applicable attacks are the content leak attack, and the watermarking attack.

Hot read: An attacker has read access to the data storage, while it is in use. Real world examples for this situation are an unreliable backup tape operator, or network attached storage devices. In addition to previous attacks, data modification patterns are revealed to an attacker as well.

Hot write: An attacker has read/write access to the data storage while it is being in use. There are 3 different severity levels for data modification threats:

1. undetected meaningful modification,
2. undetected meaningless modification,
3. detected modification.

Undetected meaningful modification can lead to unauthorised system access, when for instance a security map is modified. Many UNIX file systems store permissions in a bitmap[12]. In conjunction with the CBC data modification attack, this threat becomes serious, and even more grave, if the location of the security map is hinted through the watermarking attack as well. Moving ciphertext around on disk might result in the break down of the user privilege system. For instance, an attacker might modify `/etc/passwd` or `/etc/shadow` of a UNIX system[13], or overwrite

[12] ext2fs stores the access permission in the i_flags field in the inode table.

[13] This is trivial to implement in Linux when using CBC plain-IV. Any user can use the fibmap system call to query the kernel for the logical block number for any file. Using this call for `/etc/passwd` and a private file will reveal the IVs of both files for any public IV scheme. The attacker can then craft a modified passwd file taking into account the IV difference. The modified passwd can include his user name with user ID zero, which is equivalent to superuser under Linux. To compromise the system, the attacker has to copy his modifications over to the location of `/etc/passwd`.

the physical content of a file the attacker has access to with the content of a protected file.

Undetected meaningless modification can cause uncontrolled system disruption. Corrupting the database of firewalling rules might cause an improper configured boot system to skip the firewalling setup and proceed regularly leaving insecure services open to an attacker.

Detectable modification can help to stop threat like the one outlined above. To ensure that modifications are detectable, we need integrity and authenticity. The encryption block layer can immediately cease all disk access to a media, when the authenticity information signals tampering.

Chapter 5

Password management

Querying Google for "Security Hardware" results in about 72 million hits. Many vendors advertise key tokens, tamper-resistant smart cards, and biometric access devices. However, the majority of these devices only work with the software they came with, and are not compatible with a broad range of security applications including hard disk encryption systems. Because of this and probably also because of the additional costs, the majority of users do not employ such devices. The majority of consumer hard disk encryption systems is based on regular off-the-shelf hardware. In this chapter, we will see which requirements have to be met that hard disk encryption systems are reasonably secure with regular hardware.

5.1 Key hierarchies for passwords

Key hierarchies are well-known constructs in cryptography. Key material is treated as data, encrypted and handled like data. PGP uses key hierarchies to counter the performance penalties of public key encryption systems. The same is true for the Secure Shell Protocol (SSH). A session key for a regular block cipher is generated on the fly and sent via public key encryption to another party. Upon successful decryption of the session key by the other party, this session key can be used by both parties for further communication.

Another example for key hierarchies is SSL. SSL supports public key infrastructures (PKI), which can be depicted as tree of signatures originating from a set of trusted root signatures. Regular web browsers come packaged with a set of well known public keys, for instance keys of Verisign, Thawthe or RSA Data Security Inc. These companies offer key creation and signing services to their clients. We can use the idea of key hierarchies in hard disk encryption as well.

5.1.1 A simple design for multiple passwords

Encryption data by a single key means that this key can only be changed by re-encryption the whole data with the new key. For large amounts of data, such a transition is not easily done on-the-fly and also, re-encryption is a lengthy process, which would cause unacceptable interruption to the whole system if done offline. Furthermore considering the storage properties of magnetic media

used nowadays, it is far from sure that re-encryption is feasible with respect to the destruction (overwriting) of the data encrypted with the obsolete key.

A key hierarchy solves the re-encryption problem and makes the overwriting problem easier to solve. The key utilised for encryption is not entered directly (or deduced by hashing or other means), but stays fixed and its material is in turn stored encrypted with another key. To avoid confusion, we will call the key that encrypts the bulk data *master key*, and the key that encrypts the master key's material *user key*. The master key's encrypted key material must be stored, and this requires only 16 to 32 bytes. In the next chapter, we will see that this parsimony is not always a good property.

Having a second layer of keys, changing the user key becomes easy: The master key is recovered from the decryption via the old user key, it is encrypted with the new user key, and the encryption result is stored in place of the old encrypted copy. The bulk of data does not have to be re-encrypted as the master key does not change.

A second key layer does also add another possibility, namely multiple keys. The master key can be encrypted multiple times by different user keys, thus enabling the system to have more than one access path to its data.

5.1.2 Secret splitting

By storing multiple copies, the user can build up access conditions with as many disjunctions as he likes, possession of `<user key 1> or <user key 2> or <user key 3>`. What is missing is conjunctions.

Secret splitting is a technique that allows a secret to be reconstructed only if all sub parts are present. In general, the secret can be any arbitrary raw data. This technique enables the user to not only use disjunction in the access condition, but also conjunctions in any combination or enclosure, for instance, `(<user key 1> and <user key 2>) or <user key 3>`.

A simple secret splitting method for a message M is the generation of $S_1 \dots S_{n-1}$ random messages of the same size as the original message M, and the computation of

$$S_n = M \oplus \Big(\bigoplus_{i=1}^{n-1} S_i \Big) \tag{5.1}$$

The contents of M will be split across n parts $S_1 \dots S_n$. We can transform this equation into

$$M = S_n \oplus \Big(\bigoplus_{i=1}^{n-1} S_i \Big) = \bigoplus_{i=1}^{n} S_i \tag{5.2}$$

This also gives the equation to reconstruct the original message from all parts S_i. As we can see, every sub part is treated equally. Each sub part affects every output bit, and every possible message can be created be modifying a single S_i. The field addition could be replaced by a Galois Field multiplication to produce another secret splitting scheme. Then Equation (5.1) has to be changed to calculate the multiplicative inverse. See [Sch96], pp. 70.

5.1.3 Threshold schemes

Even though any combination can be built by disjunction and conjunction, it has limitations. Consider the problem posed by Liu in [Liu68],

> Eleven scientists are working on a secret project. They wish to lock up the documents in a cabinet so that the cabinet can be opened if and only if six or more scientists are present. What is the smallest number of locks needed? What is the smallest number of keys to the locks each scientist must carry?

The minimal solution is "462 locks and 252 keys per scientist". This number grows exponentially with the number of scientists involved. Threshold schemes have been invented to have more modest requirements. A (k, n)-threshold scheme allows access to a secret, if at minimum k secret fragments out of n are available.

A threshold scheme can be constructed by the means of an algebraic system of polynomials. An example of such a scheme is developed in [Sha79]. A polynomial of order $k-1$

$$f(x) = a_{k-1}\, x^{k-1} + a_{k-2}\, x^{k-2} + \cdots + a_1\, x + a_0$$

with $a_1 \ldots a_{k-1}$ random constants and a_0 equal to the secret can be used to implement the threshold scheme. This polynomial is evaluated n times and handed out to the distinct parties as points in a 2-dimensional plane $(x_i, f(x_i))$. A joint effort of at least k parties can reproduce all coefficients of the $k-1$ order polynomial by solving the following equation:

$$\begin{bmatrix} f(x_1) \\ f(x_2) \\ \cdots \\ f(x_{k-1}) \\ f(x_k) \end{bmatrix} = \begin{bmatrix} a_0 & a_1 x_1 & \ldots & a_{k-1}(x_1)^{k-1} \\ a_0 & a_1 x_2 & \ldots & a_{k-1}(x_2)^{k-1} \\ \ldots & \ldots & \ldots & \ldots \\ a_0 & a_1 x_{k-2} & \ldots & a_{k-1}(x_{k-2})^{k-1} \\ a_0 & a_1 x_{k-1} & \ldots & a_{k-1}(x_{k-1})^{k-1} \end{bmatrix}$$

Therefore, the secret contained in a_0 is recoverable. Fewer than k sub parts do not reveal any information for a brute-force attack, as the algebraic system is under-determined and has additional degrees of freedom that span across the field's entire value domain. To make this scheme better implementable, the coefficients should be elements of a Galois Field.

5.2 Anti-forensic data storage

5.2.1 The problem with magnetic storage

According to [Gut96], hard disks have a very long memory. Even if data appears to be gone, even if the disk has been reinitialised with zeros, even if you invoked the security-erase ATA command of your IDE hard disk, data can be easily recovered. Special care must be taken to destroy data properly. Two level key hierarchies must store the encrypted master key on disk. Extra precaution must be taken for this data, since data is not guaranteed to be ever erased. Bad block remapping of modern firmwares supports data safety but weakens the opposite, data destruction.

Unintended remapping of sector data is especially bad for key material created via a key hierarchy, as the encrypted key material is typically very short and can be accommodated by a single sector in a reserved remapping zone.

5.2.2 Design

If the probability to destroy a certain block of data is $0 \leq p \leq 1$, then the probability that the block survives is $1 - p$. Given a set of data consisting of n data items, the probability to erase the whole data set becomes worse since p^n becomes smaller as n becomes larger. But the probability that the whole set survives, $(1 - p)^n$, becomes smaller as well with increasing n. If $(1 - p)^n$ becomes smaller, than $1 - (1 - p)^n$ must become larger, which is exactly the chance that the whole block does not survive. The reader should notice the subtle difference between "whole block is erased" and "whole block does not survive". The former describes that all items are destroyed. The latter means that one data item or more is destroyed.

Usually, you cannot control p, as this depends on the hard disk and the firmware. But the size of the data set n can be chosen by the user. By making n larger, one can make the chance of destroying at least one data item arbitrary large. For instance, assume the backing device to be a redundant RAID array such that it is extremely hard to erase data on it, say $p = 0.01$. We can still destroy one of 1000 data blocks almost for sure with a probability of 0.99995.

After this insight, the remaining task is to construct a data vector from an original data item and to make the destruction of a single vector part crucial for the survival of the whole information contained in the data vector. This goal can be achieved if the information of a single data item D is distributed uniformly to the larger data vector. "Uniformly" in this context refers to the property that any data item is equally important to extract the original information D. D can be an encrypted master key or some other piece of data, that we want to be revocable from a hard disk.

The mapping from D to the data items of the data vector S must fulfil a few requirements. When $\mathbb{D}$ is the domain of D and $\mathbb{S}$ is the domain of the items of the vector S, then there must be a function $\mathfrak{S}$ to generate a mapping $\mathbb{D} \rightarrow \mathbb{S}^n$. Also, there must be a function $\mathfrak{D}$ to reconstruct D, so $\mathbb{S}^n \rightarrow \mathbb{D}$. We require this reconstruction function to fail completely if one of the items of S is missing. This can be expressed as

$$\mathrm{P}[\mathfrak{D}(s_1, \ldots, s_{i-1}, x, s_{i+1}, \ldots, s_n) = D] = \frac{1}{|\,\mathbb{D}\,|}$$

for all $s_1, \ldots, s_{i-1}, s_{i+1}, \cdots \in \mathbb{D}$ and for all $i = 1, \ldots, n$. Any D is equally likely to be generated when a single s_i is missing, or in other words, D has a strong dependency on all arguments of $\mathfrak{D}$.

An ad-hoc method to create this dependency can be stated: For the derived data set $S = s_1, s_2, \ldots s_n$, generate $s_1 \ldots s_{n-1}$ random data items and compute s_n as

$$s_n = s_1 \oplus \cdots \oplus s_{n-1} \oplus D$$

The reconstruction can be done by computing

$$D = \bigoplus_{i=1}^{n} s_i$$

that is XORing all data items together. If one item s_i is missing, D cannot be reconstructed, since every single s_i affects the entire result. This simple scheme is also a form of secret splitting as seen in Section 5.1.2, but instead of handing out the parts to different parties, they are all stored on disk.

As we can destroy a single unspecific data item easily as shown in the previous few paragraphs, and as a single missing data item makes the base information unrecoverable, the data item D can be made erasable. When the user desires to revoke D, all items s_i are overwritten and D, which is contained in these items, is erased with a significantly improved probability.

We can improve this scheme a bit. A bit-change in s_i will only cause a bit-change at the corresponding position of D. Assume s_i to be partially destroyed in a certain range, then D is unrecoverable at exactly the same range. Only a full destruction of a single s_i causes the entire information of D to be unrecoverable. This property can be enhanced by making the recovery scheme sensitive to bit-errors.

The first approach is to insert a diffusion element into the chain of XORs. A cryptographic hash function can be used as such an element. Assume $s_1 \dots s_{n-1}$ to be random items again, then we compute q_i as hash chain of these items,

$$\begin{aligned} q_1 &= s_1 \\ q_i &= H(q_{i-1}) \oplus s_i \end{aligned} \tag{5.3}$$

The last item s_n is chosen as

$$s_n = q_{n-1} \oplus D$$

The diffusion of s_n has to be omitted, as the hash function is not invertible. This degrades security marginally. The reconstruction can be carried out by computing q_{n-1} as in (5.3), and evaluating:

$$D = q_{n-1} \oplus s_n$$

This yields the original D.

As illustration, you can find the overall composition in Figure 5.1. H denotes the diffusion element[1]. In the splitting phase, s_1 to s_{n-1} are randomly generated and the intermediate result I is computed. Then s_n computed as $s_n = D \oplus I$. When recovering the base information, the whole chain is computed as shown resulting in D, the original data item.

A sample implementation of the scheme as given in Figure 5.1 is available at [Fru04]. It is called AFsplitter, short for anti-forensic splitter. In the following chapter, we use terms like AFsplit or AFmerge to refer to information decomposition $\mathfrak{S}$ or reconstruction $\mathfrak{D}$.

[1]The diffusion element H with an arbitrary input and output size can be constructed from a cryptographic hash h with fixed output size. Simply calculate $H(q)$ as $h(0 \mid\mid q) \mid\mid h(1 \mid\mid q) \mid\mid h(2 \mid\mid q) \mid\mid \cdots$ until enough data is obtained.

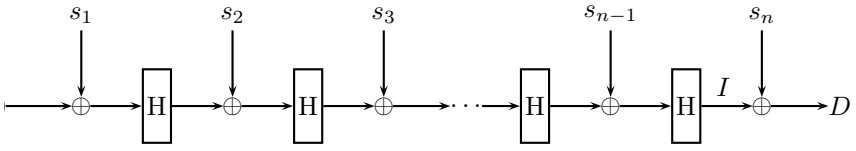


Figure 5.1: AFsplitter

It is also possible to use a cipher instead of a hash function. The design of Figure 5.1 is similar to CFB encryption but without storing the ciphertext. CFB has an infinite error propagation, and this property is crucial for the applicability as information splitting technique. It is possible to use any other mode with this property, for instance ABC, IGE, or PCFB.

CFB is easy to implement, but if for some reason the implementor wants to stick to a hash function, the scheme can be made perfect for the diffusion of the last block s_n even with a hash function. As you can see from Figure 5.1, a single bit of s_n does not affect the whole value of D. This shortcoming can be removed by using obfuscating s_1 with the help of s_n. Computing $s_1 \dots s_n$ as in the original scheme, then calculate

$$s_{n+1} = H(s_n) \oplus s_1$$

and store the result in place of s_1. This is invertible,

$$s_1 = H(s_n) \oplus s_{n+1}$$

and must be computed before the original reconstruction process can start. The difference is, if s_n is partially unavailable, s_1 will be totally different, because s_n is diffused by H. As you can see from Figure 5.1, any error in s_1 will be propagated until the end of the chain, causing D to be different. Thus, diffusion even for the last element s_n is possible even with non-invertible diffusion functions.

5.2.3 Parameter determination

As the diffusion element in the information splitting design causes a bit error in the data storage to have devastating effect on the output, we focus on the probability of destroying bits, or in other words producing bit errors. Given the probability p that a bit is destroyed, $1 - p$ denotes the probability that a bit survives. The probability, that an entire data set with n items, each item k bits long, survives, is $(1-p)^{n\,k}$. We would like to protect an encrypted version of the master key, which is k bits long. Our aim is to make the survival probability equal to the probability of guessing the master key with a single guess, which is $\frac{1}{2^k}$.

$$(1-p)^{n\,k} = \left(\frac{1}{2}\right)^k$$

By applying ln to both sides and cancelling k, we obtain

$$n = \frac{\ln 2}{\ln(1-p)} \tag{5.4}$$

n is also known as the inflation factor, because the underlying data item is inflated to take n-times the size of the original. Note that n does not depend on the length of the key k. To give the reader a feeling for the magnitudes, selected values of p with the corresponding inflation factor are presented in the following table.

p	n
0.05	13.5
0.01	67.0
0.001	692.8
0.0001	6931.1

5.3 Passwords from entropy weak sources

5.3.1 The problem

User supplied passwords have usually two unwanted properties. They are short, and often are based on dictionary words. Both emerge due to the user's preference for easy memorable short passphrases. From a cryptographer's point of view, the problem with short strings as well as English words is that they lack entropy.

A password generated by the standard Unix command `mkpasswd` will give you a 13 character random string, where the term "character" refers to a set of 64 symbols[2]. For better illustration, these are four passwords generated by `mkpasswd`: `DWy1XBje4dPDk FwgoS2m.kZisI gfr2sist.GqIA ftZZgfLdJR236`. If the reader succeeded to remember a single one, his task to remember a modern 128-bit key would be only half way done, as a single string accounts only for 78 bit entropy[3]. All of these strings have to be remembered to provide enough entropy for a 256-bit key.

The usual passphrase length is nowhere near that, nor is it is randomly generated. Choosing a regular English word with 10 characters yields 12 bits entropy[4]. The entropy gap to a 128 bit key or a 256 bit key is highly visible. A potential attacker could easily traverse a dictionary instead of the whole password domain. Even if these 10 characters are randomly generated, the passphrase domain will only be 60 bits when using 2^6 symbols/character. In 1999 distributed.net compromised a 56 bit DES key within 22 hours, so we need more than that.

Despite new techniques to make passphrases more memorable for people[5], the technological close in on the brute force resistance of low entropy passphrases will push the requirements for their size. As technology advances, larger key sizes become attackable. The logical consequence is to increase the key size, but this implies that the underlying passphrase has to be lengthened as well. As technology advances much faster than the humans ability to remember arbitrary passwords, the gap to the feasibility of dictionary or low entropy attacks becomes smaller everyday.

[2] A-Z, a-z, 0-9, '/', '.'
[3] $\prod_1^{13} 64 = 2^{78}$
[4] The entropy per character is less than 1.2 bits according to Shanon's experiment [Mah00]
[5] See the NIST report on picture passwords [NIS03c]

5.3.2 PBKDF2

There is no deterministic algorithm that can magically sprinkle entropy on top of a short passphrase. Determinism of algorithms and the idea of entropy are diametrical concepts. The following scheme does not improve insufficient entropy, but it artificially hampers the computation of a single brute force operation. A single brute force trial consists of (1) setting up the block cipher with a key derived from the attacked passphrase, (2) decryption of a cipher block likely to contain known information[6], and (3) checking the result against plausible outcomes. Making regular decryption slower is not desirable, because it would also hamper the legitimate use of the system, hence the subject of interest is the key setup phase.

By inserting a CPU intensive transformation into the key setup path, the single brute force operation can be made much more expensive to carry out without impairing regular use. Mathematically, the CPU intensive transformation is a map of the given password to the key domain. This function does not have to be invertible. Thus, instead of computing $D_k(c)$, the attacker is forced to compute $D_{f(k)}(c)$, where computing f accounts for the majority of processing time.

The trick is that the intensity of $f(k)$ can be adjusted by the user at free will. The user will choose an intensity level that makes $f(k)$ computable on his encryption device in a reasonable amount of time (one to five seconds). By inserting the computation of $f(k)$ into the key setup phase, we artificially hamper the key setup. Cipher designers have a different aim. They try to keep key setup time as low as possible. This is ok, as a cipher designer assumes the key to have enough entropy. But a key derived from a short user password does not share this property.

What is the gain by artificially hampering the key setup? Compare a random key k_1 of size 2^x with a second key of size 2^y that is obtained with $k_2 = f(p)$ from a password p with an effective entropy of 2^x. There is only one attack path for k_1, a regular brute force attack on the x-bit key domain. This implies 2^x steps of work. For k_2 derived from $f(p)$, there a two attack paths: a brute force attack on the y-bit key domain of k_2 or attacking the x-bit password domain of p. By various extension schemes[7], the number of bits y derived from a password can be made as large as necessary to provide security over the lifespan of the data. The size x of the password domain cannot be adjusted, as the human's capability to remember arbitrary strings does not scale at all. While the x-bit key k_1 might become attackable over time, the y-bit key k_2 stays secure. The only chance for an attacker to attack k_2 is by attacking it via the x-bit password domain. But this path is blocked by the CPU intensive computation of $f(p)$. This blocking does not come from an increased complexity, as in both cases an attacker has to do 2^x steps of work. The difference is that these steps are much harder for the attack path via $f(p)$ as it requires computing $f(p)$ for every attacked password.

Under two assumptions, this hardening is very effective. Challenging a 60-bit password requires computing 2^{60} $f(p)$ results. This would take 10^{10} years, when the attacker can compute one result every second. However, this calculation is too naive. It assumes (1) the attacker has the same processing

[6] such as English text or a file system header

[7] For instance, the hash extension trick we have seen in the AFsplitter section

capabilities as the user, and (2) there is no technological growth. Both assumptions are untenable, but before we relax them in the next section, we want to know if there is a way for an attacker to circumvent the computation of $f(p)$.

An attacker needs access to a precomputed dictionary containing the result $f(p)$ for every password, so that instead of computing $f(p)$, he can lookup the result for $f(p)$ in his dictionary. A complete enumeration of the password domain generated by `mkpasswd` (of the size 2^{78}), would require 8 yottabytes[8] storage, which might be achievable one day[9]. Therefore, instead of utilising a generic function $f(p)$, it is advisable to make the function not only password dependent, but also dependent on a random value s. With this measure, an attacker would have to have a dictionary available, which contains $f(p, s)$ for all possible passwords p and possible values s.

By adding a bit to s, the storage requirements for all precomputed dictionaries doubles. s does not have to be secret and can be stored on disk, so it is relieved of any size constraints imposed by the limitations of humans to remember random content. Hence, the solution is quite simple, make s that large that storing a dictionary for all passwords and values s is unlike to become feasible for the time span one would like to keep the data secret. s is a typical salt, as it serves to randomise the process and does not have to be secret.

Inside PBKDF2

Hash functions qualify best for the purpose of constructing a password-based key derivation function, as they are well defined for variable sized inputs such as passphrases. To make them CPU intensive, one has just to iterate them a number of times. PBKDF2 (presented in RFC 2898 [Kal97]) is based on this idea. PBKDF2 stands for "password-based key derivation function, revision 2".

When iterating the hash function h, we have to assure that by repeatedly applying it the co-domain of h does not degenerate into a smaller set of values. This smaller set is called a cycle. If a function with a domain of size n provides n different values for n successive applications of h to an arbitrary start value, the function is said to be cycle free.

Formally, a function $h : \mathbb{H} \to \mathbb{H}$ is a cycle free function if

$$\{h^i(a) \mid 0 \leq i < |\mathbb{H}|\} = \mathbb{H}$$

for all $a \in \mathbb{H}$.

This property cannot be verified for hash functions such as SHA1 or ripemd160 simply because $\mathbb{H}$ is too large.[10] Assume n is very large and the function contains a single short cycle, the attacker can shortcut the computation by directly trying the relatively few values of that cycle. This is more likely to succeed than brute force, because the probability is above-average that after n iterations of h, the result gets caught in the set of cycle values.

To prevent such shortcut attacks, PBKDF2 does not simply iterate a hash function h, but also compute a result based on its intermediate values. XORing

[8] 2^{30} Petabytes

[9] Cryptographers tend to be satisfied only when the storage complexity exceeds 10^{78}, the approximate numbers of atoms in the universe.

[10] Universal hash functions are designed to be cycle-free.

the intermediate values together should provide enough safety against this possible degeneration. For a discussion of the entropy degeneration of hash functions see [Use01].

In addition, PBKDF2 can generate an output of arbitrary length k. The following function f builds on the iteration of a pseudo-random function, usually a hash function in an HMAC setup [BCK97]. It yields the i^{th} block[11] derived from the password p, the salt s, and the iteration depth c.

$$f(p, c, s, i) = h^1(s \mid\mid i) \oplus h^2(s \mid\mid i) \oplus \cdots \oplus h^c(s \mid\mid i)$$

where $h^j(a) = \text{PRF}(P, h^{j-1}(a))$ and $h^0(a) = a$.[12]

As you can see, the intermediate results of the iteration $h^c(\dots)$ are XORed together to form the result of the i^{th} block. The final result, which is k blocks long, is produced by the concatenation of all results for different i,

$$f(p, c, s, 1) \mid\mid f(p, c, s, 2) \mid\mid \cdots \mid\mid f(p, c, s, k)$$

5.3.3 Numbers

The naive estimation of the PBKDF2 security in the last section built on the assumptions that technology does not evolve and that an attacker has at most the computational resources of the user at his disposal. We will relax these assumptions and show that the pressure on PBKDF2 comes from two sides: (1) the relative advantage of the attacker over the user measured in PBKDF2 calculations/second, and (2) the growth of the attackers computing resources that will occur over a longer period.

In contrast to similar models for cipher security, PBKDF2 security does not depend on the current level of technology. As the intensity of the PBKDF2 function can be adjusted easily, the user will always choose an iteration count that an equally equipped attacker will have to spend one second for a single brute force trial. An attacker possessing k-times more processing resources than the user has to spend $\frac{1}{k}$ seconds on a single trial. However, this term only depends on the resource ratio between the attacker and the user, but not on the absolute processing capabilities of the attacker. So, the speed with which the attacker can compute the underlying hash function is irrelevant.

Number Crunchers and Specialisation

It is reasonable to assume that there is a gap between the processing capabilities of a user and those of a well-funded and equipped attacker. There are two factors that determine processing power: scale and specialisation.

"Economy of Scale" might exist in other fields, but for massive multiprocessing it does not apply. Adding 100% of processing capabilities to a multiprocessor system does not make it twice as fast. Additional scheduling work has to be done, additional I/O, and additional task distribution. The homogeneity factor is certainly below 1.

[11] measured in output blocks of the hash function

[12] $h^j(\dots)$ also depends on p, but we have omitted the variable from the presentation, as it is merely handed through from f to PRF.

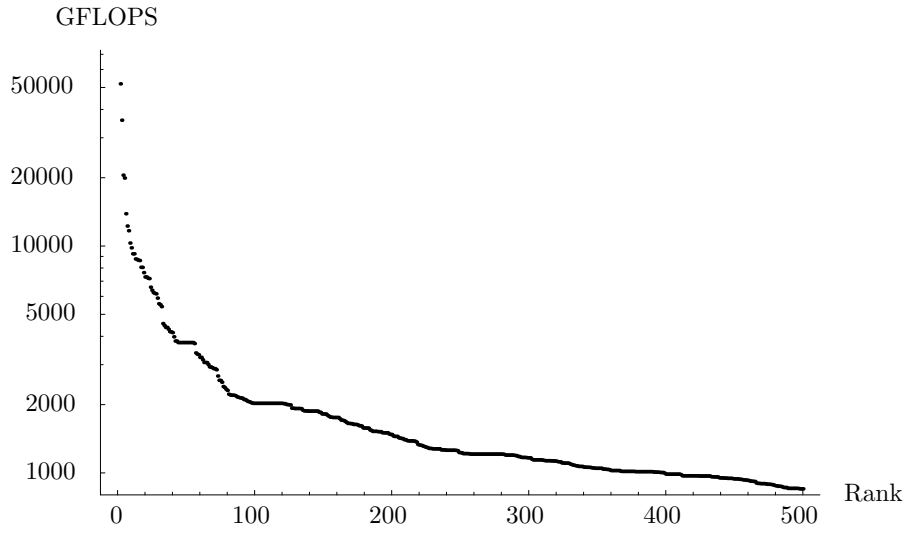


Figure 5.2: The worlds Top 500 supercomputers' processing power

What is the upper bound one can assume for "scale"? The top rank in the list of supercomputers[13] is claimed by IBM's BlueGene, with 72 TFLOPS[14]. The second rank among supercomputers is held by NASA with 51 TFLOPS, followed by the Earth Simulator deployed in Japan with 35 TFLOPS. More data is plotted in Figure 5.2. In contrast, a Pentium 4 achieves 0.8 GFLOPS. This is only the peak value of the P4 2.2GHz Northwood core, the average is much lower, about 0.4 GFLOPS[15]. By conservative estimates, the ratio between the user's processing power and that of a well-funded attacker might be $1 : 10^6$.

Most users of cryptography operate with a general purpose CPU. The Intel architecture is the most widely used platform. What are the gains one can expect when using custom hardware? We give three comparisons between Intel and specialised hardware for SHA1, MD5 and AES.

SHA1 as well as MD5 belongs to the MD4-family of hash functions. An MD4-family hash has 3 to 5 rounds, each consisting of 16 steps – except SHA1, which has 20 steps. The members of the MD4-family are different in their step function. The authors of [BGV96] have done a good job on optimising every cycle out of their implementations for the MD4-family on the Pentium architecture. The results[16] are a 837 cycles/block SHA1 implementation and a 337 cycles/block MD5 implementation.

Helion provides commercial IP cores for many cryptographic applications

[13] according to `http://top500.org`

[14] FLOPS are not suitable as base for comparisons in every aspect. For a more in-depth treatment about the art of benchmarking, we refer the reader to [PH97]. We still use FLOPS, as the aspects of hardware specialisation are addressed separately.

[15] `http://www.tech-report.com/reviews/2002q1/northwood-vs-2000/index.x?pg=3`, `http://www.tech-report.com/reviews/2001q2/pentium4-1.7/index.x?pg=4`, `http://www.tech-report.com/reviews/2001q1/p4-vs-athlon/index3.x`, `http://www.hardwareanalysis.com/content/reviews/article/1475.5/`

[16] after the refinements in [Bos97]

on FPGAs [Hel]. They claim that their cores achieve a speed of 65 cycles/block for MD5 and 82 cycles/block for SHA1. A similar speed is provided by a SHA1 core on opencores.org[17], which is designed to take 81 cycles/block. Another SHA1 core is offered by HDL Design House with speed of a 80 cycles/block along a MD5 core with a performance of 64 cycles/block.

For the Pentium architecture, Lipmaa offers a hand-optimised AES library [Lip]. It is the fastest implementation on Intel at the time of this writing. A block is encrypted at 254 cycles/block. The fastest FPGA implementation presented in [EYCP00] is able to deliver an encrypted block in only 2.1 cycles.

To summarise: We found a specialisation advantage of magnitude 1:10 for SHA1, 1:5 for MD5 and 1:100 for AES. There are real world examples of scalability with magnitudes about $1 : 10^6$. In total, an attacker might own equipment that allows him to calculate PBKDF2 results 10^8 times faster than the user. We call this factor, *SS-factor*, short for scale and specialisation.

A factor we have not mentioned yet but that is sometimes brought forward as an argument: The *they-had-a-larger-basement-then-we-thought* factor, or NSA factor. We leave estimates of the true processing capabilities of the National Security Agency or other institutions to conspiracy theorists, as this work is not the place to adopt doubtful assumptions. It is up to the user to decide over additional security margins.

Technological Growth Model

> "The future isn't what it used to be!"
>
> – Believed to be a comment made by an IBM executive concerning predicted future trends in personal computing around 1992.

The quotation above is one of the many forecasting quotes[18]. The reason for their diversity might be, that forecasting is such a hard thing to get right. Nonetheless, the reader should get a feeling for the strength that is associated with passwords secured by PBKDF2.

In our model, we assume that the user sets his password at period 0 and stores data on disk protected with this password. An attacker gains access to this data by some means and brute-force attacks the password. Here, technological improvement works in favour of an attacker, which can upgrade his decryption hardware any time. In contrast, the user's data is a cold piece of information and the only hope of its owner is that the attacker does not find a plausible interpretation for it. The user must anticipate this disadvantage as well anticipate the technological growth that will happen over the longer time frame. According to this findings, he must chose an appropriate password length – that is password entropy.

The following growth model assumes that technology grows exponentially with a yearly growth rate r. Let $f(t)$ be a function that denotes how many passwords an attacker can try in period t. For period 0, $f(0)$ depends on his resource advantage over the user k and the confirmation time the user has chosen c.

$$f(0) = \frac{k}{c}$$

[17] http://www.opencores.org/projects.cgi/web/sha_core/overview
[18] http://www.met.rdg.ac.uk/cag/forecasting/quotes.html

With a password confirmation time of one second, an attacker can try $k \times 365 \times 24 \times 3600$ passwords in the first year[19]. When technology grows with a yearly growth rate r, we can model the processing power by the function

$$f(t) = f(0)\, e^{rt}$$

The integral over $f(t)$ gives the total advantage gained over a period t with respect to period 0.

$$F(t) = f(0) \int_0^t e^{rx}\, \mathrm{d}x$$

If $r > 0$,

$$F(t) = \frac{e^{rt} - 1}{r}\ f(0) \tag{5.5}$$

when $r = 0$,

$$F(t) = t\ f(0) \tag{5.6}$$

To compute the time required for the total traversal of a password domain with the size p, we require t to hold for $p = F(t)$. By expressing t as a function of p, we obtain

$$t = \frac{1}{r} \ln\left(1 + \frac{p\, r}{f(0)}\right) \qquad \text{for } r > 0$$

and

$$t = \frac{p}{f(0)} \qquad \text{for } r = 0$$

This equation gives the number of years that are required to check a password domain with size p.

We examine five different password sets:

1. An English sentence about 40 characters long. Such a sentence is also known as passphrase. The advantage is that it is easier to remember and can be typed more easily. We expect 1.2 bits entropy/character, hence $|P| = 2^{48}$.
2. A picture password containing 10 elements from a set of 60 pictures, $|P| = 2^{60}$.
3. A random 13 character string, as generated by `mkpasswd`, yielding 6 bits entropy per character, $|P| = 2^{78}$.
4. A 11/60 picture password and a random 6 character string, $|P| = 2^{100}$.
5. An entropy strong 128-bit key probably stored on a smart card.

Along, we examine five different growth scenarios:

1. Technology doubles every 2 years, $r = 0.347$
2. Technology doubles every 5 years, $r = 0.139$
3. Technology doubles every 10 years, $r = 0.069$

	password domain cardinality				
technology doubles every	2^{48}	2^{60}	2^{78}	2^{100}	2^{128}
2 years	10^{-2}	15	50	95	150
5 years	10^{-2}	30	120	230	370
10 years	10^{-2}	45	225	445	725
20 years	10^{-2}	75	430	875	1435
no growth	10^{-2}	365	10^{7}	10^{14}	10^{23}

Table 5.3: Traversal times for an entire password domain

(in years, rounded to 5 years)

4. Technology doubles every 20 year, $r = 0.035$
5. No growth, $r = 0$

Table 5.3 has the results of all combinations of the scenarios above, assuming a one second password verification and an SS-factor of 10^8. The growth parameter has no influence on the security of the 2^{48} password domain, because there is no security at all. An equipped attacker can traverse the password domain in 4 days. $|P| = 2^{60}$ is more promising without technological growth, but here we can clearly see the security degradation by the technological progress. The distributed.net effort accomplished to do a key search in a 56-bit domain in 24 hours. Would the key have been protected by PBKDF2, this would not have been possible in such a short time frame and probably impossible without substantial technological progress.

For the 2 years scenario, $|P| = 2^{100}$ appears reasonable secure, while for the 5 years scenario, $|P| = 2^{78}$ is the first with more than 100 years password security. The results for the non-zero growth scenarios differ widely from the no-growth scenario even compared with the most conservative 20 year doubling scenario.

It is unlikely that an attacker finds the key by a lucky pick much earlier than given in Table 5.3, as the majority of the computing effort happens in the last few years. For instance, only the first half of a 2^{100} password space have been searched after 93 years in the two-year doubling rate scenario.

A question the reader might want to ask: What's a reasonable r? This is hard to tell. There are estimations that brute forcing a 128-bit AES key is impossible with regular silicon based hardware simply because there is not enough energy on this planet to power the chips. Most likely the answer depends on the feasibility of technologies like cold fusion and quantum computers. The best a cryptographic paper can do is to cover the most extreme and the most conservative scenario.

Calculation $|P|$ from t

We combine (5.5) and (5.6),

$$p = \begin{cases} \frac{e^{rt}-1}{r} f(0) & \text{if } r \neq 0 \\ \frac{t}{f(0)} & \text{if } r = 0 \end{cases}$$

[19] neglecting the technological growth that happens in the first year

	secure for		
technology doubles every..	50 years	100 years	250 years
2 years	78	103	178
5 years	64	74	104
10 years	60	65	80
20 years	59	61	69
no growth	57	58	59

Table 5.4: Minimum password bits for different security requirements

(calculated with SS-factor = 10^8)

and evaluate it for different t and different r values. Table 5.4 shows the results as $\log_2$. We can use this table to read the minimum numbers of password bits, also for different SS-factors. For every raise in the decimal power of the SS-factor, add 3.3 bits ($\frac{\log_{10}}{\log_2}$) to the number of bits.

5.4 TKS1: Template Key Setup 1

This section intends to summarise all the findings from the previous sections and distill a concrete design template from them. We have found that (1) password hierarchies are useful for multiple passwords, (2) anti-forensic splitting is required for the storing revocable data, (3) PBKDF2 is an enhancement to password security.

Figure 5.5 assembles all techniques in a single structure. As you can see, the key is coming from a key storage because of the key hierarchy. It is stored in an anti-forensic split way, therefore it has to be merged before it can be used. When the password is entered correctly, the encrypted master key is decrypted and handed to the disk cipher.

The salt as well as the AF-split master key is coming from a key storage. The salt can be saved without splitting, since it is not sensitive to the security of the system. The passphrase comes from an entropy-weak source like the user's keyboard input.

In the following two step-by-step listings, we presents the recovery process and initialisation process of TKS1. In Chapter 6, you will find a more concrete formalisation of these algorithms.

To recover the master key, we have to

1. read salt and iteration rate from key storage,
2. read the AF-split encrypted master key from storage and AF-merge in memory to get the encrypted master key,
3. let the user enter the passphrase,
4. process the passphrase with PBKDF2 (salt and iteration as parameter) to derive the key for the encrypted master key,
5. decrypt the encrypted master key with the derived key,
6. set up the real time cipher with the master key,

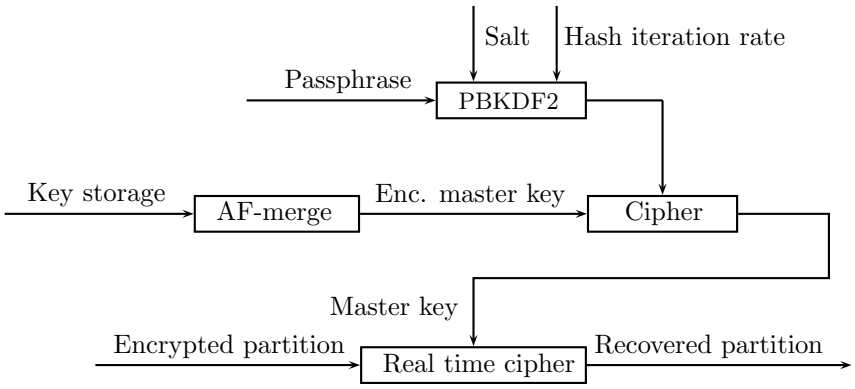


Figure 5.5: TKS1 scheme

7. (destroy master key copy in memory.)

We have focused on the simplest case here. To support multiple passwords, the process can be repeated for different key material in the key storage.

The initialisation of the system is straight forward.

1. generate a master key,
2. generate a salt for PBKDF2,
3. choose an appropriate hash iteration rate by benchmarking the system,
4. let the user enter the passphrase,
5. process the passphrase with PBKDF2 to obtaining the key for the master key cipher,
6. encrypt the master key with the master key cipher,
7. AF-split the encrypted master key,
8. save the AF-split encrypted master key, the iteration rate and the password salt to storage,
9. (setup the real time cipher with the master key,)
10. (destroy master key copy in memory.)

After carrying out these steps, everything is in place for a successful master key recovery later. The benchmarking step is not stringent, but no implementor should lock his implementation to a fixed value. The reason is the implementor does not know on which systems his software will end up or how long his software will be in use. The PBKDF2 iteration count must grow along with technology. By benchmarking the user's system and assuming that the user

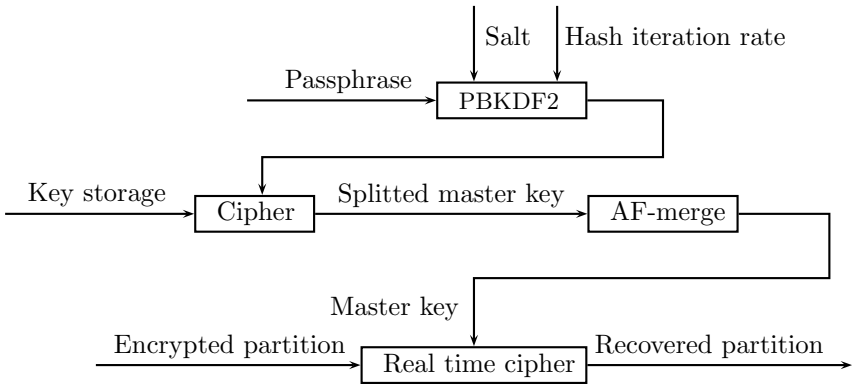


Figure 5.6: TKS2 scheme

has a reasonable modern hardware[20], we can ensure that the iteration count will be set appropriately.

When the user wants to change the password, the master key can be recovered as shown above, but instead of using it for the real time cipher, it can be re-encrypted using a new password derived with PBKDF2. The master key encrypted with the old password can be easily destroyed as proposed in Gutmann's paper [Gut96].

5.4.1 TKS2: Variant

A variant can be obtained from TKS1 by switching AF-splitting and encryption. This scheme is easier to implement when the implementor aims to reuse a transparent hard disk encryption subsystem. This subsystem can be setup up to encrypt and decrypt disks in a transparent way. By pairing the key encryption with a disk write, encryption and writing can be done in one step namely by writing to a virtual partition. The transparent encryption system will then transform the data with the encryption algorithm and write it to the backing partition. Pairing the key writing and the key encryption requires that the AF operation is moved to the middle of the process.

The scheme has marginally other cryptographic features than TKS1. An attacker has more ciphertext at his disposal as compared to TKS1, because the encryption is done after data inflation. However, as the AFsplitter generates random data with high entropy, there is little chance for known plaintext attacks here.

[20]Reasonable modern means that you do not use your 25 MHz processor, when there are 3 GHz machines around. Next to them, using a 300MHz processor might be still an exception, but as we have seen in the SS-factor analysis, a speed difference within magnitudes of one power of 10 are negligible.

Chapter 6

A tour of LUKS: Linux Unified Key Setup

The first traces of cryptography can be spotted in the Linux kernel in 2000 with the release of the patch-int kernel series. As in the year 2000 the US export restrictions for cryptography were still believed to be a drag shoe for US developers, this patch series was never included into the main kernel. The first patch-int series was for the Linux 2.2 kernel, but got out of sync with the main kernel development with the advent of the 2.4 kernel. Long after the first version of the 2.4 kernel, two other projects stepped up: kerneli, more consent oriented and driven by a small community, and loop-AES, a one man show. Both projects based their hard disk encryption modules on the loopback device driver, which was already present in the kernel. Traditionally, the losetup tool was used to manage loopback devices. But the maintainer of util-linux, to which losetup belonged, refused to include third party modification to losetup, and both projects, kerneli and loop-AES, started to distribute their own version of util-linux. Both projects matured and took separate ways in terms of features, and so their util-linux modifications became incompatible with each other.

Both of them coexisted for a while, and unfortunately some Linux distributions started to include one of the modified versions, while other distributions included the other one. So, an unpleasant situation arose for end users. Having encrypted a partition with one Linux distribution, the user was not sure if he could access the partition with another distribution.

The incompatibility originated from the key processing that is done inside losetup. The user supplies a passphrase, from which the actual key is derived via hashing and further handed to the kernel. As new features were added to kerneli and loop-AES projects, losetup needed to understand these features in order to activate them in the kernel. The situation became particularly bad, when cryptoloop of kerneli was included in the 2.6 release of the Linux kernel, but no official version was around to use this kernel facility. With util-linux 2.12, its maintainer wrote his own implementation to handle encryption. It was neither compatible with kerneli nor loop-AES and added a third incompatible implementation into the bargain. The result were frustrated user postings to the development mailing lists.

The situation was partially remedied by cryptsetup. cryptsetup was an

equivalent to losetup but instead for cryptoloop it was used for dm-crypt, another kernel facility for hard disk encryption. With cryptsetup, a unified tool was created especially for the purpose of processing user keys. But again, distributions decided to take their own approaches and started to manage dm-crypt volumes with the low-level tool dmsetup. Also cryptsetup took a number of parameters to set key size, hash algorithm and other options. This options setting would have to be remembered, because other settings would result in differently key and an unaccessible partition.

To remedy this problem, the author of this work developed a meta data format to save all information required for the key setup. But while writing the standard documents, he noticed that the set of information that needed to be saved could only be surveyed, if the key setup process was specified. This lead to the question, what is a secure key setup?

The answer to this question is given in Section 5.4. TKS1 is a byproduct of the standardisation efforts for LUKS. TKS1 is a template design for the key setup process, and includes all ingredients necessary to provide a high level of safety for managing keys on regular user hardware.

After all theoretic prerequisites have been arranged, a proof-of-concept implementation was released in summer 2004. Thanks to much user feedback, LUKS hit 1.0 in March 2005. The final 1.0 version switched to the equally secure TKS2 variant for implementation reasons.

Along with the 1.0 version of the reference implementation, the LUKS on-disk format specification 1.0 [Fru05] was released. With this document, LUKS not only standardises meta data syntax, but also its semantics to ensure compatibility among implementations. Defining the semantics of descriptor tokens effectively standardises the whole disk encryption. Thus, LUKS is a complete hard disk encryption standard.

The advantages of LUKS are

- compatibility via standardisation,
- secure against low entropy attacks thanks to PBKDF2,
- support for multiple keys by using a key hierarchy,
- effective passphrase revocation thanks to AFsplitter,
- free reference implementation, GNU Public License.

6.1 Disk layout

A LUKS disk has the following layout:

LUKS phdr	KM1	KM2	···	KM8	bulk data

A LUKS partition starts with the LUKS partition header (phdr). TKS2 is built as two level key hierarchy that demands for a key storage area. Encrypted and anti-forensic processed copies of the master key are stored in the key material sections labelled KM1, KM2, ..., KM8 in the figure. After the key material, the bulk data is located, which is encrypted by the master key. The phdr together with an active key material section contains all necessary information needed by the encryption subsystem to access the bulk data.

field name	description
magic	magic for LUKS partition header
version	LUKS version
cipher-name	cipher name specification
cipher-mode	cipher mode specification
hash-spec	hash to use in HMAC mode for all PBKDF2 operations
payload-offset	start offset of the bulk data (in sectors)
key-bytes	number of key bytes
mk-digest	master key checksum from PBKDF2
mk-digest-salt	salt parameter for master key PBKDF2
mk-digest-iter	iterations parameter for master key PBKDF2
uuid	UUID of the partition
key-slot-1	key slot 1
key-slot-2	key slot 2
...	...
key-slot-8	key slot 8

Table 6.1: PHDR layout

6.1.1 The partition header

The first data items of the phdr are designated for the bulk data encryption subsystem: the used cipher, the cipher mode, the bulk data offset, the master key size. The second part is used for the verification of master key candidates: a master key digest and the PBKDF2 parameters for digest generation. A UUID is also stored, so partitions on mobile disks such as USB sticks can be recognised and paired with their passphrases automatically. Table 6.1 summaries the content of the phdr.

The cipher and cipher mode settings are those to use for the bulk data associated with a LUKS partition. The underlying encryption system must support these settings, of course. To ensure compatibility, the specification document [Fru05] maintains a registry of tokens to use for cipher names, cipher mode names and hash names. The registry also includes normative information or references to other normative documents for such tokens. Because of this, every aspect of a LUKS partition becomes well-defined, i.e. LUKS is a complete standard.

Also included in the phdr is a digest of the master key. This is used for password verification. To generate the digest, the PBKDF2 primitive is reused and the PBKDF2 parameters such as salt and iteration count are stored in the phdr. Usually, the iteration count is much lower than for password protection, since the master key is generated from an entropy-strong source and does not make a good target for brute force attacks. Attacking the bulk data directly is more promising.

The key-slot is an aggregate data item containing the fields depicted in Table 6.2. A key slot refers to a copy of the master key keyed with a password or key file. A key slot contains information about the PBKDF2 processing parameters (salt, iteration count) and the anti-forensic processing (stripe numbers). The key material offset points to the key material section associated with this key slot. The key material contains the raw binary data representing an in-

field name	description
active	state of key slot, enabled/disabled
iterations	iteration parameter for PBKDF2
salt	salt parameter for PBKDF2
key-material-offset	start sector of key material
stripes	number of anti-forensic stripes

Table 6.2: Key slot layout

flated encrypted master key. The phdr contains 8 key slots, hence there are also 8 key material sections on a partition.

Every active key slot is locked by an individual password. The user may choose as many passwords as there are key slots. To access a partition, the user has to supply only one of these passwords. The unlocking process is described in the next section.

6.2 Semantics

Having discussed all data structures, we would like to see them in action. In the following section, the terms password, passphrase or key file can be used interchangeable, as LUKS does not make a difference between them. LUKS features 4 essential high-level commands,

create partition: initialise an empty partition with a new master key, and set an initial passphrase.

open partition: recover the master key with the help of a user supplied passphrase, and install a new virtual mapping for the backing device.

add key: add a new passphrase to a key slot. A valid passphrase has to be supplied for this command.

revoke key: disable an active passphrase.

These high level actions are suitable to be made available to the end user. For presentation and implementation reasons, it is better to think of some of these high level actions as an aggregation of low level actions:

pre-create partition: generate a new master key and a new valid phdr, and write the phdr with no key slot active.

set key: use a given copy of the master key to activate an empty key slot with a new user supplied passphrase.

recover master key: given a user passphrase, try to recover the master key from an active key slot.

We separate the high level action of *create partition* into two parts: *pre-create partition* and *set key*. The pre-creation step is responsible for initialising the phdr, generating the master key and the disk layout. After this step, the key slots are all empty, and the partition would be unusable if the process

would stop here. Hence, a *set key* step is carried out with the master key still in memory to activate a key slot.

Equally, the *add key* step is separated into *recover master key* and *set key* step. The master key recovery has to be done with an existing valid passphrase to bring a copy of the master key into memory. Then, an empty key slot is used in the *set key* low level action to be filled with another copy of the master key keyed with the new passphrase.

The *open partition* consists only of the *recover master key* low level action followed by the installation of a new virtual partition into the operating system device layer[1].

Other high level commands can be synthesised. Password changing is the aggregation of *add key* with the new passphrase followed by *revoke key* for the old one. The following presentation is not as detailed as the LUKS specification, and shortened for the sake of better readability. Implementors should consult [Fru05].

6.2.1 Partition pre-creation

This action is responsible for generating a valid phdr. The procedure depicted in Listing 6.3 starts with the master key generation. The master key's size is chosen as requested by the user (Lines 1-2). The phdr is initialised (Lines 5-10) with the variables supplied by the user, and a digest for the new master key is computed and stored in the phdr (Lines 12-18). The master key digest will be used later to check a master key candidate for validity.

Listing 6.3: Partition initialisation

```
masterKeyLength ← defined by user
masterKey ← read from random source,
            length masterKeyLength

phdr.magic ← LUKS_MAGIC
phdr.version ← 1
phdr.cipher-name ← as supplied by user
phdr.cipher-mode ← as supplied by user
phdr.key-bytes ← masterKey
phdr.uuid ← generate uuid

phdr.mk-digest-salt ← read from random source,
                      length: LUKS_SALTSIZE
phdr.mk-digest-iteration-count ← LUKS_MKD_ITER or user input
phdr.mk-digest ← PBKDF2(masterKey,
                        phdr.mk-digest-salt,
                        phdr.mk-digest-iteration-count,
                        LUKS_DIGESTSIZE)
```

[1] not depicted here, because beyond the scope of this document

```
20  stripes ← LUKS_STRIPES or user defined
    keyMaterialSectors ← ⌈(stripes*masterKeyLength)/SECTOR_SIZE⌉
22  currentOffset ← ⌈(size of phdr)/SECTOR_SIZE⌉

24  for each keyslot in phdr do as ks {
        ks.active ← LUKS_KEY_DISABLED
26      ks.stripes ← stripes
        ks.key-material-offset ← currentOffset
28      currentOffset ← currentOffset + keyMaterialSectors
    }
30
    phdr.payload-offset ← currentOffset
32
    write phdr to disk
```

The rest of the procedure takes care of generating the disk layout. The AF inflation factor (`stripes`) is determined, and used to calculate the size of a single key material section. The key material sections have to be aligned to sector boundaries, as the `key-material-offset` field's unit is sectors. Then, a loop assigns a sector region to every key slot (Lines 24-29). The bulk data starts after the last key material section.

6.2.2 Key setting

The more interesting part is how keys are activated. In Listing 6.4 it is assumed that a copy of the master key is in memory. How a master key can be brought to memory from an active key slot is described in the next section.

After the user passphrase is read, the parameters for PBKDF2 are determined. Good security requires an appropriate number of PBKDF2 iterations. The iteration count is chosen, so that the password verification takes the number of intended seconds. The longer a password verification takes, the harder brute-force attacks will be, see Section 5.3. The salt is generated randomly before the PBKDF2 processing takes place (Lines 8-14). We follow the design guidelines of TKS2, and inflate the master key by using the AF splitting technique. The result is of size $\texttt{masterKeyLength} \times \texttt{stripes}$ (Lines 16-19).

The split master key is encrypted with the PBKDF2 processed password (Line 21). Notice that the key material is encrypted with exactly the same encryption settings as the bulk data, and the same cipher, cipher mode and key size is used. This is because the master key has to be protected at an equal strength as the bulk data. This seems reasonable, as the master key in turn protects the bulk data, hence its encryption must not be the weakest link in the security construction.

Finally, the encrypted and split master key is written to disk, the key slot is flagged active, and the phdr is updated.

Listing 6.4: Key setting

```
masterKey ← must be available, either because it is still in memory from the
            pre-creation action or because it has been recovered by a correct
            password.
masterKeyLength ← phdr.key-bytes

emptyKeySlotIndex ← find inactive key slot index in phdr by scanning the
                    keyslot.active field for LUKS_KEY_DISABLED
keyslot ks ← phdr.keyslots[emptyKeySlotIndex]

password ← read password from user input
ks.salt ← read from random source, length LUKS_SALTSIZE
PBKDF2-IPS ← benchmark system          // iterations per seconds
ks.iteration-count ← PBKDF2-IPS ×
                     intentedPwdCheckingTime   // in seconds
pwd-PBKDF2ed ← PBKDF2(password,
                      ks.salt,
                      ks.iteration-count,
                      masterKeyLength) // key size is the same as
                                          for the bulk data

splitKey ← AFsplit(masterKey,          // source
                   masterKeyLength,    // source length
                   ks.stripes)         // multiplication factor
splitKeyLength ← masterKeyLength × ks.stripes

encryptedKey ← encrypt(phdr.cipher-name,   // cipher name
                       phdr.cipher-mode,   // cipher mode
                       pwd-PBKDF2ed,       // key
                       splitKey)           // content

write to partition(encryptedKey,           // source
                   ks.key-material-offset, // sector number
                   splitKeyLength)         // length in bytes

ks.active ← LUKS_KEY_ACTIVE          // mark key as active in phdr

update keyslot ks in phdr
```

6.2.3 Master key recovery

The initial requirements for a key recovery action are a user password and a valid LUKS phdr (Lines 1-5). As we have seen in the previous action, the master key is AF-split, and stored encrypted by a PBKDF2 processed password. To recover it, the process has to be reversed. First, we need a PBKDF2 processed version of the user passphrase. So, PBKDF2 is called with the parameters found in the key slot (Line 8). The result is used to decrypt the split master key (Line 16). After an AF-merge operation, a possible master key has been recovered (Line 21). The master key candidate is checked against the master key digest already stored in the phdr (Lines 25-29). If the candidate's digest matches, the process stops and a valid master key is handed to the caller. If not, the checking is repeated for all other active key slots.

Listing 6.5: Master key recovery

```
read phdr from disk
check LUKS_MAGIC and version number in phdr

pwd ← read password from user input
masterKeyLength ← phdr.key-bytes

for each active keyslot in phdr do as ks {
    pwd-PBKDF2ed ← PBKDF2(pwd,
                          ks.salt,
                          ks.iteration-count
                          masterKeyLength)
    encryptedKey ← read from partition at
                   ks.key-material-offset and length
                   masterKeyLength × ks.stripes

    splitKey ← decrypt(phdr.cipher-name,
                       phdr.cipher-mode,
                       pwd-PBKDF2ed,       // key
                       encryptedKey)       // content

    masterKeyCandidate ← AFmerge(splitKey,
                                 masterkeyLength,
                                 ks.stripes)

    candidateDigest ← PBKDF2(masterKeyCandidate,
                             phdr.mk-digest-salt,
                             phdr.mk-digest-iter,
                             LUKS_DIGEST_SIZE)
    if candidateDigest = phdr.mk-digest:
            break loop and return masterKeyCandidate as
              correct master key
}
return error, password does not match with any keyslot
```

6.2.4 Key revocation

This actions is quite simple. The key material section holding a copy of the master key is clear according to Gutmann, and the corresponding key slot is flagged disabled.

Peter Gutmann has shown in [Gut96], how data destruction shall be done to maximise the chance that no traces are left on the disk. All LUKS implementations are required to follow Gutmann's advice.

6.3 LUKS for dm-crypt

LUKS is not just a paper tiger. It is an existing and usable hard disk solution. The pseudo code presented in the previous section is implemented in cryptsetup. Unfortunately, the maintainer Christophe Saout has not found time yet to include the LUKS extensions for cryptsetup in the main branch of cryptsetup. So, the author of this work draw the conclusion and forked cryptsetup. The LUKS enabled branch is available from `http://luks.endorphin.org`.

At the time of this writing, cryptsetup-luks is packaged for the Linux distributions Gentoo, Debian and Redhat. Gentoo has the best LUKS support, and manages to be installed and booted from a root file system residing on a LUKS partition.

cryptsetup-luks implements all actions described in this section. To facilitate shell scripting, trivial actions like LUKS partition detection or UUID extraction are also implemented.

Appendix A

Mathematical supplements

This *Mathematica* Notebook is intended as addition to "New Methods of Hard Disk Encryption".

CBC collision attack

This section presents the mathematics behind Chapter 4. The question whetever a sample out of a finite domain contains duplicate items is an elementary problem. As argued in Chapter 4, the probability that a sample with k elements out of a set with n elements is collision-free is:

```
In[1]:= g[n_, k_] := n! / ((n-k)! n^k)
```

This representation is ok for small values of n and k. But as we are investigating large n $\left(\text{around } 2^{128}\right)$ and large k $\left(\text{around } 10^{12} - 10^{15}\right)$, *Mathematica* cannot carry out the factorial computation. For large n and k, we replace it by Stirling's estimate,

```
In[2]:= Stirling[n_] := Sqrt[2 π n] (n/E)^n

In[3]:= g[n, k] /. {n! → Stirling[n], (n - k)! → Stirling[n - k]}
Out[3]= E^-k n^(1/2 - k + n) (-k + n)^(-1/2 + k - n)
```

This can be simplified, when rewritten as logarithm. *Mathematica*'s FullSimplify function comes to hand using the assumption, that n and k are positive integers.

```
In[4]:= collAssumpt = {{k, n} ∈ Integers, k > 0, n > 0};

In[5]:= FullSimplify[Log[E^-k n^(1/2 - k + n) (-k + n)^(-1/2 + k - n)], Assumptions → collAssumpt]
Out[5]= -k + Log[(1 - k/n)^(-1/2 + k - n)]
```

We have to express one more constraint for k. When we have $k > n$ there must be a collision, as the set N contains n elements and after $n = k$ the set is exhausted for unique picks. The collision probability is 1 for $k > n$. We concentrate on $0 < k < n$, as this is more interesting. Adding this assumption, we simplify the expression a bit further.

```
In[6]:= collAssumpt = Append[collAssumpt, n > k > 0];
```

```
In[7]:= FullSimplify[-k + Log[(1 - k/n)^(-1/2+k-n)], Assumptions → collAssumpt]
Out[7]= -k + (-1/2 + k - n) Log[1 - k/n]
```

The logarithm's argument $1 - \frac{k}{n}$ is a term very close to 1. A floating point representation will quickly fail here. Therefore, we replace the logarithmic expression by its corresponding Taylor series.

```
In[8]:= Series[Log[1 - x], {x, 0, 5}]
Out[8]= -x - x^2/2 - x^3/3 - x^4/4 - x^5/5 + O[x]^6
```

The expression becomes (omitting the rest term $0[x]^6$),

```
In[9]:= -k + (-1/2 + k - n) Normal[%] /. x → k/n
Out[9]= -k + (-k^5/(5 n^5) - k^4/(4 n^4) - k^3/(3 n^3) - k^2/(2 n^2) - k/n) (-1/2 + k - n)
```

We can now given an estimate of `g[n, k]` that is easily computable for large n and k,

```
In[10]:= h[n_, k_] := e^(-k+(-k^5/(5 n^5) - k^4/(4 n^4) - k^3/(3 n^3) - k^2/(2 n^2) - k/n) (-1/2+k-n))
```

Let us compare the two variants,

```
In[11]:= {g[10000, 100], h[10000, 100]} //N
Out[11]= {0.608566, 0.608566}
```

With `h` we obtained a function that can be evaluated for bigger n and k without the dangling sword of Damocles impersonated by over- and underflows. Let us investigate $n = 2^{128}$and $k = 10^{12}$, which is roughly equivalent to a 200 GB hard disk running with AES and CBC in plain-IV mode.

```
In[12]:= N[h[2^128, 10^12], 25]
Out[12]= 0.9999999999999853063206
```

This is the collision free probability. On first glance, it appears to be very unlikely that a collision occurs. To verify this, we rewrite this as collision probability, which states the magnitudes a bit better,

```
In[13]:= 1 - h[2^128, 10^12] //N
Out[13]= 1.44329 × 10^-15
```

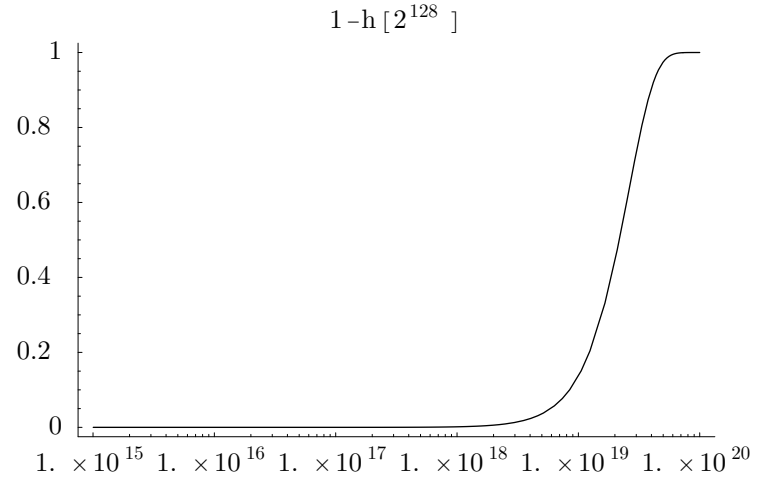
Let us investigate this function for a set of values for k.

```
In[14]:= 1 - h[2^128, {10^13, 10^14, 10^16, 10^18, 10^19, 10^20}] //N
Out[14]= {1.46883×10^-13, 1.46937×10^-11, 1.46937×10^-7, 0.00146829, 0.136651, 1.}
```

As you can see, the collision probability sharply increases at 10^{19}. That is far below the size of set N, which is approximately 10^{38}. A plot reveals more features of the function,

```
In[15]:= <<Graphics`Graphics`

In[16]:= LogLinearPlot[1 - h[2^128, Floor[x]], {x, 10^15, 10^20},
           PlotLabel → 1 - h[2^128]]
```

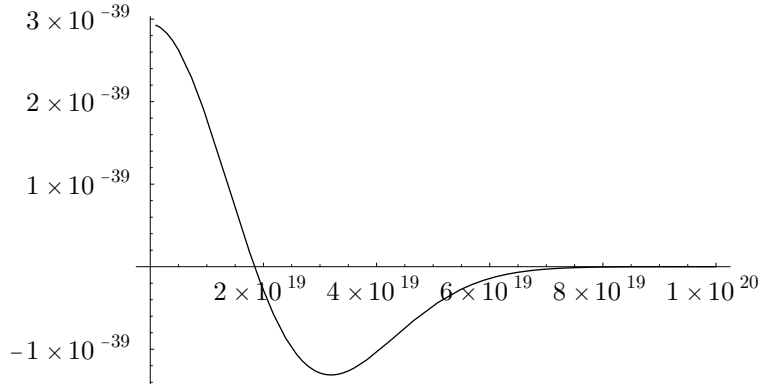


Out[16]= -Graphics-

The point with the biggest rise of the collision probability is called inflexion point. From the plot above, we can deduce it to be near 2×10^{19}. But we can deduce this point algebraically by considering the second derivative.

```
In[17]:= h''[n_, k_] = D[1 - h[n, k], {k, 2}] ;

In[18]:= Plot[h''[2^128, Floor[x]], {x, 10^18, 10^20}]
```



Out[18]= -Graphics-

We use `FindRoot` to find nulls in this function. We will see why it is reasonable to start with $\sqrt{2^{128}}$ later.

```
In[19]:= FindRoot[h''[2^128, k], {k, Sqrt[2^128]}, WorkingPrecision → 20]
Out[19]= {k → 1.844674407370955162 × 10^19}
```

But in fact, our hint for $\sqrt{2^{128}}$ as start value was perfect. Notice that

```
In[20]:= Sqrt[2^128]//N
Out[20]= 1.84467 × 10^19
```

This is equal to our result for k. The explanation for this is given in Chapter 4. The probability at $k = 2^{64}$ is

```
In[21]:= h[2^128, 2^64] //N
Out[21]= 0.606531

In[22]:= Clear[f]; Clear[p]; Clear[g] ; Clear[h]
```

Tech-Growth Model

This model is used in Chapter 5 to estimate the security of PBKDF2 protected passwords. In this section, P denotes the set of passwords, and its cardinality, $|P|$, is denoted by p. We assume that c describes the processing power at the beginning of the first period. Assuming that the user chooses 1 second password confirmation time, 365 24 3600 passwords can be checked in the first period (without technological growth). This makes $c = 1/(365\ 24\ 3600)$. k is the SS-factor, the ratio between the user's processing power and the attackers processing power.

Let $f(t)$ be a function that describes the number of passwords that can be computed in the time frame t until $t + 1$.

```
In[23]:= f[t_, r_, c_] := k/c e^(r t)
```

The value computed by $f(t)$ is only a projection for the period t. It neglects the technological growth that will happen in period t. The real number of passwords computed is between $f(t)$ and $f(t+1)$, when the technological growth rate is larger than 0. Integrating this function obtains the overall number of passwords computed in a given time frame.

```
In[24]:= F[t_, r_, f0_] = f0 ∫_0^t e^(r x) dx
Out[24]= (-1 + e^(r t)) f0
         ---------------
                r
```

We define the special case $r = 0$,

```
In[25]:= F[t_, 0, f0_] = f0 t;
```

Given the number of passwords p, we can solve this equation by demanding $p == F(t)$ for t. This gives an explicit equation for t, that means after t years, the whole password domain will have been traversed and hence, the correct password must have been found.

```
In[26]:= Solve[p == (-1 + e^(r t))/r f_0, t][[1]] //Simplify
Out[26]= {t → Log[1 + p r/f_0]/r}
```

For $r > 0$, we use this solution

```
In[27]:= t[r_, p_, f0_] = Log[1 + p r/f0]/r;
```

But for $r = 0$, the solution of the integral $F[t]$ becomes $f_0\, t$. Hence, $t = \frac{p}{f_0}$.

```
In[28]:= t[0, p_, f0_] = p/f0;
```

We define the function `r[y]` that gives the respective growth rate, when the technology level doubles every y years.

```
In[29]:= r[y_] := Log[2]/y
```

Reserving one second for password confirmation and assuming an SS-factor of 10^8, $365\,24\,3600\,10^8$passwords can be tried in the first period. We will use this value for f_0.

```
In[30]:= f0' = 365 24 3600 10^8;
```

Using the `Outer` function, we generate all combinations of the scenarios we have chosen,

```
In[31]:= resT = Outer[t, {r[2], r[5], r[10], r[20], 0},
                        {2^37, 2^48, 2^60, 2^78, 2^100, 2^128},
                        {f0'}] //N;
```

and split the result into two tables (for formatting reasons),

```
In[32]:= TableForm[Rows[resT, {1, 2, 3}],
           TableHeadings →
             {{"2 years", "5 years", "10 years", "20 years", "no growth"},
               {"2^37", "2^48", "2^60"}}]
```

Out[32]=

	2^37	2^48	2^60
2 years	0.0000435813	0.0879025	13.9933
5 years	0.0000435815	0.0887074	28.4579
10 years	0.0000435815	0.0889802	47.1922
20 years	0.0000435816	0.0891174	75.4596
no growth	0.0000435816	0.0892551	365.589

```
In[33]:= TableForm[Rows[resT, {4, 5, 6}],
           TableHeadings →
             {{"2 years", "5 years", "10 years", "20 years", "no growth"},
               {"2^78", "2^100", "2^128"}}]
```

Out[33]=

	2^78	2^100	2^128
2 years	49.9706	93.9706	149.971
5 years	118.317	228.317	368.317
10 years	226.634	446.634	726.634
20 years	433.268	873.268	1433.27
no growth	9.5837×10^7	4.01969×10^{14}	1.07903×10^{23}

We can use the function `F[t, r, f0]` to analyse how big the password domain has to be, if we requried passwords to be secure for 50, 100 or 250 years.

```
In[34]:= resP = Log[2, Outer[F, {50, 100, 250}, {r[2], r[5], r[10], r[20], 0},
                      {f0'}]]//N;
```

```
In[35]:= TableForm[Transpose[resP], TableHeadings →
           {{"2 years", "5 years", "10 years", "20 years", "no growth"},
             {"50 years", "100 years", "250 years"}}]
```

Out[35]=

	50 years	100 years	250 years
2 years	78.0147	103.015	178.015
5 years	64.3352	74.3366	104.337
10 years	60.2908	65.3352	80.3366
20 years	58.556	61.2908	68.8364
no growth	57.1298	58.1298	59.4517

Bibliography

[Ano04] Anonymous, *Private Conversation*, 2004.

[AT&T] AT&T Research, *The On-Line Encyclopedia of Integer Sequences*, http://www.research.att.com/~njas/sequences/.

[BAG01] A. A. Belal and M. A. Abdel-Gawad, *2-D Encryption Mode*, 2001, http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/2dem/2dem-spec.pdf.

[BCK97] M. Bellare, R. Canetti, and H. Krawczyk, *The HMAC papers*, 1996-1997, http://www.cs.ucsd.edu/users/mihir/papers/hmac.html.

[BDST03] C. Beaver, T. Draelos, R. Schroeppel, and M. Torgerson, *ManTiCore: Encryption with Joint Cipher-State Authentication*, Cryptology ePrint Archive, Report 2003/154, 2003, http://eprint.iacr.org/2003/154.

[BGKM03] I. F. Blake, C. Guyot, C. Kent, and V. K. Murty, *Encryption Of Stored Data In Networks: Analysis of a tweaked block cipher*, 2003, http://grouper.ieee.org/groups/1619/email/msg00256.html.

[BGV96] A. Bosselaers, R. Govaerts, and J. Vandewalle, *Fast Hashing on the Pentium*, 1996, http://www.esat.kuleuven.ac.be/~bosselae/publications.html.

[BK04] M. Bellare and T. Kohno, *Hash Function Balance and its Impact on Birthday Attacks*, 2004, http://citeseer.ist.psu.edu/bellare02hash.html.

[BN00] M. Bellare and C. Namprempre, *Authenticated Encryption: Relations among notions and analysis of the generic composition paradigm*, *Lecture Notes in Computer Science*, 1976, 2000, http://www-cse.ucsd.edu/users/mihir/papers/oem.html.

[Bos97] A. Bosselaers, *Even Faster Hashing on the Pentium*, 1997, http://www.esat.kuleuven.ac.be/~bosselae/publications.html.

[BR93] M. Bellare and P. Rogaway, *Random Oracles are Practical: A Paradigm for Designing Efficient Protocols*, in *ACM Conference on Computer and Communications Security*, pages 62–73, 1993, http://citeseer.ist.psu.edu/bellare95random.html.

[BR99] M. Bellare and P. Rogaway, *On the Construction of Variable-Input-Length Ciphers*, *Lecture Notes in Computer Science*, 1636:231–244, 1999, http://citeseer.ist.psu.edu/bellare99construction.html.

[BRW03] M. Bellare, P. Rogaway, and D. Wagner, *EAX: A Conventional Authenticated-Encryption Mode*, Cryptology ePrint Archive, Report 2003/069, 2003, http://eprint.iacr.org/2003/069.

[CE05] J. Campbell and R. J. Easter, *Annex C: Approved Random Number Generators for FIPS PUB 140-2*, 2005, http://csrc.nist.gov/publications/fips/fips140-2/fips1402annexc.pdf.

[CGH04] R. Canetti, O. Goldreich, and S. Halevi, *The Random Oracle Methodology, Revisited*, *Journal of the ACM*, 51:557–594, 2004.

[Cro00] P. Crowley, *Mercy: A Fast Large Block Cipher for Disk Sector Encryption*, in B. Schneier, editor, *Fast Software Encryption: 7th International Workshop*, volume 1978, pages 49–63, Springer-Verlag, New York, USA, 2000, http://www.ciphergoth.org/crypto/mercy/mercy-paper.pdf.

[DDN98] D. Dolev, C. Dwork, and M. Naor, *Non-malleable Cryptography*, 1998.

[DP83] D. W. Davies and G. I. P. Parkin, *The Average Cycle Size of the Key Stream in Output Feedback Encipherment*, *Lecture Notes in Computer Science*, 149, 1983, http://www.springerlink.com/index/BVUT4EBQP5TMT5MQ.

[EYCP00] A. Elbirt, W. Yip, B. Chetwynd, and C. Paar, *An FPGA Implementation and Performance Evaluation of the AES Block Cipher Candidate Algorithm Finalists*, 2000, http://csrc.nist.gov/encryption/aes/round2/conf3/papers/08-aelbirt.pdf.

[Fer02] N. Ferguson, *Collision attacks on OCB*, 2002, http://www.cs.ucdavis.edu/~rogaway/ocb/fe02.pdf.

[Flu01] S. Fluhrer, *Cryptanalysis of the Mercy Block Cipher*, 2001, http://www.ciphergoth.org/crypto/mercy/fluhrer-dc.html.

[Fru04] C. Fruhwirth, *AF splitter reference implementation*, 2004, http://clemens.endorphin.org/AFsplitter.

[Fru05] C. Fruhwirth, *LUKS on-disk-format specification 1.0*, 2005, http://luks.endorphin.org.

[FS03] N. Ferguson and B. Schneier, *Practical Cryptography*, Wiley Publishing Inc., 2003, ISBN 0-471-22357-3.

[GD00a] V. Gligor and P. Donescu, *On Message Integrity in Symmetric Encryption - Draft*, 2000.

[GD00b] V. D. Gligor and P. Donescu, *Fast encryption and authentication: XCBC encryption and XECB authentication Modes*, 2000, http://citeseer.ist.psu.edu/gligor00fast.html.

[GD01] V. D. Gligor and P. Donescu, *ADDENDUM XCBC Encryption with Authentication and XECB Authentication Modes*, 2001, http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/xecb-mac/xecb-mac-doc.pdf.

[GM84] S. Goldwasser and S. Micali, *Probabilistic encryption*, *Journal of Computer and System Sciences*, 28:270–299, 1984.

[Gut96] P. Gutmann, *Secure Deletion of Data from Magnetic and Solid-State Memory*, 1996, http://www.cs.auckland.ac.nz/~pgut001/pubs/secure_del.html.

[Hal04] S. Halevi, *Draft Proposal for Tweakable Wide-block Encryption*, 2004, http://siswg.org/docs/EME-AES-03-22-2004.pdf.

[Har05] L. Hars, *Private communication*, 2005.

[Hel] HelionTech, *Helion IP Core Products*, http://www.heliontech.com/core.htm.

[Hel01] H. Hellström, *Propagating Cipher Feedback*, 2001, http://www.streamsec.com/pcfb1.pdf.

[HR03a] S. Halevi and P. Rogaway, *A Parallelizable Enciphering Mode*, 2003, http://eprint.iacr.org/2003/147.

[HR03b] S. Halevi and P. Rogaway, *A Tweakable Enciphering Mode*, 2003, http://eprint.iacr.org/2003/148.

[IK02] T. Iwata and K. Kurosawa, *OMAC: One-Key CBC MAC*, 2002, http://citeseer.ist.psu.edu/iwata02omac.html.

[JJV] E. Jaulmes, A. Joux, and F. Valette, *RMAC - A randomized MAC beyond the birthday paradox limit.*

[Jut00] C. Jutla, *Parallelizable Encryption Mode with Almost Free Message Integrity*, 2000, http://citeseer.ist.psu.edu/jutla00parallelizable.html.

[Jut01] C. S. Jutla, *Encryption Modes with Almost Free Message Integrity*, *Lecture Notes in Computer Science*, 2045:529–543, 2001, http://citeseer.ist.psu.edu/jutla00encryption.html.

[Kal97] B. Kaliski, *RFC 2898; PKCS #5: Password-Based Cryptography Specification Version 2.0*, 1996-1997, http://www.faqs.org/rfcs/rfc2898.html.

[Ken04] C. Kent, *Draft Proposal for Tweakable Narrow-block Encryption*, 2004, http://siswg.org/docs/LRW-AES-10-19-2004.pdf.

[KI02] K. Kurosawa and T. Iwata, *TMAC: Two-Key CBC MAC*, Cryptology ePrint Archive, Report 2002/092, 2002, http://eprint.iacr.org/2002/092.

[Knu00] L. R. Knudsen, *Block chaining modes of operation*, *Reports in Informatics*, 2000, ISSN 0333–3590, http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/abc/abc-spec.pdf.

[KSV99] M. Kassianidou, V. Srinivasan, and B. Villalobos, *Claude Shannon & Information Theory*, 1999, http://www-cse.stanford.edu/classes/sophomore-college/projects-99/information-theory/.

[KVW03] T. Kohno, J. Viega, and D. Whiting, *CWC: A high-performance conventional authenticated encryption mode*, Cryptology ePrint Archive, Report 2003/106, 2003, http://eprint.iacr.org/2003/106.

[Lip] H. Lipmaa, *Complete AES (Rijndael) Library*, http://home.cyber.ee/helger/implementations/.

[Liu68] C. L. Liu, *Introduction to Combinatorial Mathematics*, McGraw-Hill, New York, 1968.

[LN97] R. Lidl and H. Niederreiter, *Finite Fields*, Cambridge University Press, 1997.

[LRW02] M. Liskov, R. Rivest, and D. Wagner, *Tweakable Block Ciphers*, 2002, http://citeseer.ist.psu.edu/liskov02tweakable.html.

[Luc96] S. Lucks, *BEAST: A Fast Block Cipher for Arbitrary Blocksizes*, in *Communications and Multimedia Security*, pages 144–153, 1996, http://citeseer.ist.psu.edu/lucks96beast.html.

[Mah00] M. Mahoney, *Refining the Estimated Entropy of English by Shannon Game Simulation*, 2000, http://www.cs.fit.edu/~mmahoney/dissertation/entropy1.html.

[Mes82] B. E. Meserve, *Fundamental Concepts of Algebra, Reprint*, Dover Publications, 1982.

[MF04] D. A. McGrew and S. R. Fluhrer, *The Extended Codebook (XCB) Mode of Operation*, Cryptology ePrint Archive, Report 2004/278, 2004, http://eprint.iacr.org/2004/278.

[MF05] D. A. McGrew and S. R. Fluhrer, *The Extended Codebook (XCB) Mode of Operation - Version 3*, 2005, http://grouper.ieee.org/groups/1619/email/pdf00019.pdf.

[MV] D. A. McGrew and J. Viega, *The Galois/Counter Mode of Operation (GCM)*, http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/gcm/gcm-spec.pdf.

[MV04a] D. McGrew and J. Viega, *Arbitrary Block Length (ABL) Mode: Security without Data Expansion*, 2004, `http://grouper.ieee.org/groups/1619/email/pdf00005.pdf`.

[MV04b] D. McGrew and J. Viega, *Draft Proposal for Tweakable Wide-block Encryption*, 2004, `http://grouper.ieee.org/groups/1619/email/pdf00005.pdf`.

[NIS01] NIST, *Report on the Second Modes of Operation Workshop*, 2001, `http://csrc.nist.gov/CryptoToolkit/modes/workshop2/Workshop2Page.html`.

[NIS03a] NIST, *NIST Responses to Public Comments on Draft SP 800-38C*, 2003, `http://csrc.nist.gov/CryptoToolkit/modes/comments/NIST_responses.pdf`.

[NIS03b] NIST, *NIST Special Publication 800-38A: Recommendation for Block Cipher Modes of Operation*, 2003, `http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf`.

[NIS03c] NIST, *Picture Password: A Visual Login Technique for Mobile Devices*, 2003, `http://csrc.nist.gov/publications/nistir/nistir-7030.pdf`.

[NIS05] NIST, *Modes of operation for Symmetric Key Block Ciphers*, 2005, `http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/`.

[PH97] D. Patterson and J. Hennesy, *Computer Organization Design*, Morgan Kaufmann Publishers, Inc., 1997, ISBN 1-55860-428-6.

[RBBK01] P. Rogaway, M. Bellare, J. Black, and T. Krovetz, *OCB Mode*, Cryptology ePrint Archive, Report 2001/026, 2001, `http://eprint.iacr.org/2001/026`.

[Rog01a] P. Rogaway, *The Associated-Data Problem - Draft*, 2001, `http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/ocb/ocb-ad1.pdf`.

[Rog01b] P. Rogaway, *PMAC, Proposal to NIST for a parallelizable message authentication code*, 2001, `http://www.cs.ucdavis.edu/~rogaway`.

[Rog02] P. Rogaway, *Some Comments on WHF mode*, 2002, `http://www.cs.ucdavis.edu/~rogaway/ocb/11-02-156r0-I-Some-Comments-on-WHF-Mode.pdf`.

[Rog04] P. Rogaway, *Nonce-Based Symmetric Encryption*, 2004, `http://www.cs.ucdavis.edu/~rogaway/papers/nonce.pdf`.

[RW03] P. Rogaway and D. Wagner, *A Critique of CCM*, 2003, `http://www.cs.berkeley.edu/~daw/papers/ccm.html`.

[Saa04] M.-J. O. Saarinen, *Encrypted Watermarks and Linux Laptop Security*, 2004, `http://mareichelt.de/pub/notmine/wisa2004.pdf`.

[Sch96] B. Schneier, *Applied Cryptography: protocols, algorithms, and source code in C*, John Wiley & Sons, Inc., 2 edition, 1996, ISBN 0471–12845–7.

[Sha50] C. E. Shannon, *Prediction and Entropy of Printed English*, *Bell Sys. Tech. J*, 3:50–64, 1950.

[Sha79] A. Shamir, *How to share a secret*, *Commun. ACM*, 22(11):612–613, 1979, ISSN 0001-0782, http://doi.acm.org/10.1145/359168.359176.

[SIS] SISWG, *Security In Storage Working Group website*, http://www.siswg.org.

[Use01] Usenet, *Thread: Entropy reduction resulting from hashing*, 2001, http://groups-beta.google.com/group/sci.crypt/browse_frm/thread/3d8fc53381e832ec/5fbccf49538ec1ee.

[WC81] M. N. Wegman and J. L. Carter, *New hash functions and their use in authentication and set equality*, *Journal of Computer and System Sciences*, 22:265–279, 1981, http://cr.yp.to/bib/entries.html#1981/wegman.

[WHF] Whiting, Housley, and Ferguson, *Counter with CBC-MAC (CCM)*, http://csrc.nist.gov/CryptoToolkit/modes/proposedmodes/ccm/ccm.pdf.