

# Cryptography

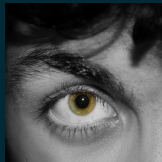
A (nearly) complete overview

Gaspare Ferraro

April 3, 2019



Visit us!  
[zenhack.it](http://zenhack.it) / [gaspa.re](http://gaspa.re)



@GaspareG  
[ferraro@gaspa.re](mailto:ferraro@gaspa.re)

# Table of Contents

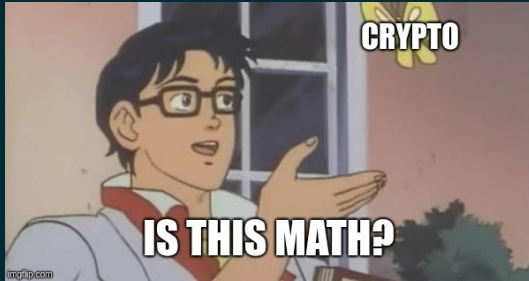
- ▶ 1. Introduction
- ▶ 2. Character encoding
- ▶ 3. Classical cryptography
- ▶ 4. Symmetric-key cryptography
- ▶ 5. Public-key cryptography
- ▶ 6. Key exchange
- ▶ 7. Hash function
- ▶ 8. Steganography

# Table of Contents

- ▶ 1. **Introduction**
- ▶ 2. Character encoding
- ▶ 3. Classical cryptography
- ▶ 4. Symmetric-key cryptography
- ▶ 5. Public-key cryptography
- ▶ 6. Key exchange
- ▶ 7. Hash function
- ▶ 8. Steganography

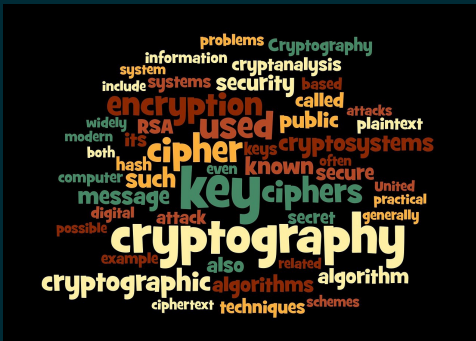
# Warning!

In this lesson we will use *math*!



It wasn't always like that though ...

# Why cryptography?



# The science of *secure communication*

# Cryptography yesterday



(a) Caesar Cipher



(b) Scytala

# Cryptography today

The needs, as well as the **available resources**, have evolved  
and today we can divide cryptography into:

**(EN|DE)CRYPTION**

**ASYMMETRIC (RSA, ECC, ...)**

**SYMMETRIC (DES, AES, ...)**

**KEY EXCHANGE**

**RSA, DH, ECDH, ...**

**AUTHENTICATION**

**RSA, DSA, ECDSA, ...**

**HASHING**

**MD5, SHA-1, SHA-256, ...**

# Table of Contents

- ▶ 1. Introduction
- ▶ 2. **Character encoding**
- ▶ 3. Classical cryptography
- ▶ 4. Symmetric-key cryptography
- ▶ 5. Public-key cryptography
- ▶ 6. Key exchange
- ▶ 7. Hash function
- ▶ 8. Steganography



# What is a message?

A **message** is a sequence of symbols used to communicate

A symbol of the message is called **character**

The set of all the possible characters is called **alphabet**

The set of all the possible (meaningful) messages is called **dictionary**

# ASCII encoding

ASCII = American Standard Code for Information Interchange  
char encoded in **7 bits** + 1 bit for check (parity bit).

ASCII (1977/1986)																
	_0	_1	_2	_3	_4	_5	_6	_7	_8	_9	_A	_B	_C	_D	_E	_F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
8	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	000A	000B	000C	000D	000E	000F
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
16	0010	0011	0012	0013	0014	0015	0016	0017	0018	0019	001A	001B	001C	001D	001E	001F
2	SP	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
32	0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
48	0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
64	0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
80	0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	005A	005B	005C	005D	005E	005F
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
96	0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006A	006B	006C	006D	006E	006F
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL
112	0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F

Letter

Number

Punctuation

Symbol

Other

undefined

Changed from 1963 version

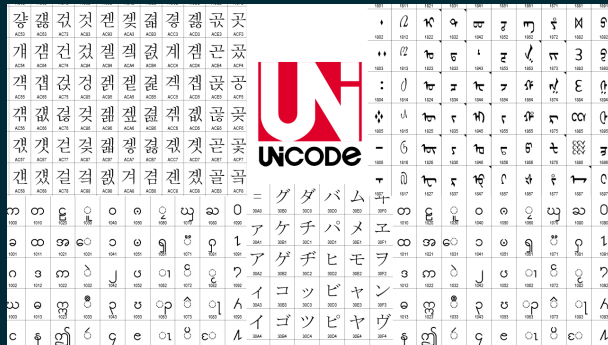
0, ..., 31 + 127 → **non-printable** chars (null, new line, tab, others)  
32, ..., 126 → **printable** chars (letters, digits, punctuation, others)

**Extended** ASCII → char encoded in 8 bits (add 128 printable chars to standard ASCII)

# Unicode encoding

Obviously 128 (or 256) characters are **not enough!**  
(Chinese, cyrillic, greek alphabets, emojis...)

Different standards: UTF-8, UTF-16, UTF-32 and others



Currently assigned "only" **137993** characters

# Morse code

(**Audio**) character encoding scheme used in (telegraph) telecommunication.

Each character is encoded using a combination of **short/long signal**.

A	• —	U	• • —
B	— • • •	V	• • • —
C	— • • — •	W	• — • —
D	— • •	X	— • • —
E	•	Y	— • — —
F	• • — •	Z	— — • •
G	— — • •		
H	• • • •		
I	• •		
J	• — — —		
K	— • — —	1	• — — — —
L	• — • •	2	• • — — —
M	— —	3	• • • — —
N	— •	4	• • • • —
O	— — —	5	• • • • •
P	• — — • •	6	— • • • •
Q	— — • • —	7	— — • • •
R	• — • •	8	— — — • •
S	• • •	9	— — — — •
T	—	0	— — — — —

# Braille code

(**Tactile**) character encoding scheme used for visually impaired people.  
Each character is encoded using a  $2 \times 3$  rectangle with "**raised dots**".

a/1	b/2	c/3	d/4	e/5	f/6	g/7	h/8	i/9	j/0
k	l	m	n	o	p	q	r	s	t
u	v	x	y	z					w

# Base64

Group message in blocks of 6 bits.

Advantage: encode all the ASCII chars in **printable characters**

source ASCII (if <128)	M						a						n											
source octets	77 (0x4d)						97 (0x61)						110 (0x6e)											
Bit pattern	0	1	0	0	1	1	0	1	1	0	0	0	0	1	0	1	1	0	1	1	1	0		
Index	19						22						5						46					
Base64-encoded	T						W						F						u					
encoded octets	84 (0x54)						87 (0x57)						70 (0x46)						117 (0x75)					

Valore	ASCII	Valore	ASCII	Valore	ASCII	Valore	ASCII
0	A	16	Q	32	g	48	w
1	B	17	R	33	h	49	x
2	C	18	S	34	i	50	y
3	D	19	T	35	j	51	z
4	E	20	U	36	k	52	0
5	F	21	V	37	l	53	1
6	G	22	M	38	m	54	2
7	H	23	X	39	n	55	3
8	I	24	Y	40	o	56	4
9	J	25	Z	41	p	57	5
10	K	26	a	42	q	58	6
11	L	27	b	43	r	59	7
12	M	28	c	44	s	60	8
13	N	29	d	45	t	61	9
14	O	30	e	46	u	62	+
15	P	31	f	47	v	63	/

Message are padded with = (e.g. *flag* → *ZmxhZwo=*)

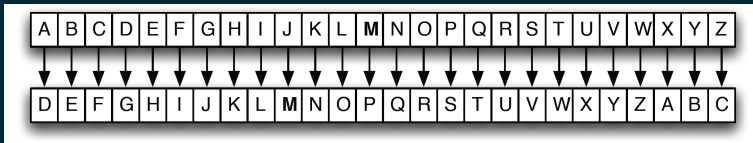
# Table of Contents

- ▶ 1. Introduction
- ▶ 2. Character encoding
- ▶ 3. **Classical cryptography**
- ▶ 4. Symmetric-key cryptography
- ▶ 5. Public-key cryptography
- ▶ 6. Key exchange
- ▶ 7. Hash function
- ▶ 8. Steganography

# Caesar cipher

Encrypt: **right shift** each letter of 3 positions

Decrypt: **left shift** each letter of 3 positions



General cipher: shift letter of  $K$  positions

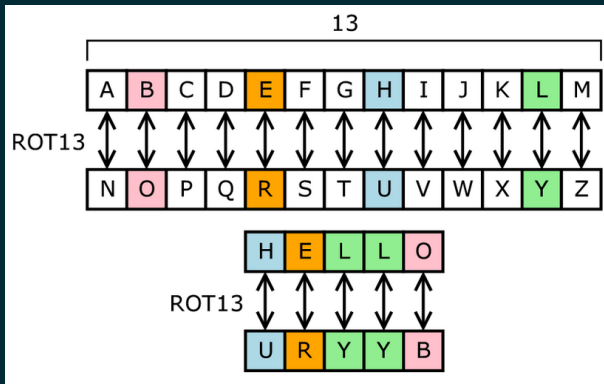
Attack: **bruteforce** all the possible  $K$  (only 25 values...)



# ROT{13, 47}

ROT13: Caesar cipher with  $K = 13$  on alphabetic dictionary

ROT47: Caesar cipher with  $K = 47$  on printable ASCII chars (33 - 126).



Why  $K = 13$  (or  $K = 47$ )? Because **Encrypt = Decrypt**

# Classical ciphers

## Substitution ciphers

- Monoalphabetic ciphers:  $C_{new} = P[C_{old}]$  (Where  $P$  is an alphabet permutation)  
(ROT-K is a monoalphabetic cipher with  $P$  is a cyclic rotation)
- Polialphabetic ciphers: **multiple substitution** alphabets  
(more than one dictionary permutation)

## Transposition ciphers

Encryption systems where the **positions** held by units of plaintext  
(characters or groups of characters) **are shifted** according to a regular system.

E.g. We want to encrypt the message *WE ARE DISCOVERED. FLEE AT ONCE* using  
the **route cipher**:

Grid:

W	R	I	O	R	F	E	O	E
E	E	S	V	E	L	A	N	J
A	D	C	E	D	E	T	C	X

Cipher text: *EJXCTEDECDAEWRIORFEONALEVSE*

# Polialphabetic substitution cipher: Vigenère

The problem with monoalphabetic ciphers is that each character of the alphabet is replaced with always the same character in the ciphertext.

What can we do to solve this weakness? **Stack more ciphers!**

The Vigenère cipher is basically a **sequence of Caesar ciphers with different shifts.**

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

This table is called **Vigenère table**.  
It contains all the Caesar ciphers.

Encryption is performed character-by-character by accessing the cell of the table within the row corresponding to the current key letter and column to the plaintext letter.

Decryption works in the same way of the encryption but we use a transposed Vigenère table.

This simple technique earned the name of **the indecipherable cipher**, and resisted attacks for over 3 centuries (1553-1863)!

# Vigenère cipher: an example

We want to encrypt the message THIS IS AN EXAMPLE with the key SECRET

First of all, we repeat the key until it has the same length of the plaintext:

$$m = \text{THISISANEXAMPLE}$$
$$k = \text{SECRETSECRETSEC}$$

In our example, the first letter of the plaintext is  $T$  and the first of the key is  $S$ , so we access the row  $S$  and column  $T$ , which yields  $L$ .

And so on until the end of the message:

$$c = \text{LLKJMLSRGOEFHPG}$$

Exercise:

$$c = \text{UEGJEKTYVDSKWJWE}$$
$$k = \text{SECRET}$$
$$m = \text{???}$$

# Transposition cipher: an example

The plaintext is written in a rectangular grid and then the columns are reshuffled, the encryption key is the **column permutation**

$m = \text{THIS IS AN EXAMPLE}$

$k = \{4, 2, 1, 3\}$

T	H	I	S		S	H	T	I
I	S	A	N	Reorder columns	N	S	I	A
E	X	A	M	----->	M	X	E	A
P	L	E	=		=	L	P	E

$c = \text{SHTINSIAMXEA=LPE}$

To recover the original message, just write the ciphertext in a grid again and apply the inverse permutation of columns

Exercise:

$c = \text{AGLF3RP}\{\text{T3UM5\_1\_D}\}\text{4B}$

$m = \text{???$

<https://www.dcode.fr/tools-list>



The screenshot displays the dcode.fr website interface. On the left, a search bar contains the text 'e.g. type scrabble' and a 'GO' button. Below the search bar, the results for 'DCODE' are listed, showing various cipher tools like 'CTFCAESARCHIPER', 'PGSPNRFNEPUVCRE', 'NEQNLPLDLCNSTAPC', 'OFROMQEMDOTUBQD', 'EVHECGUCTEJKRGT', 'DUGDBFTBSDIJQFS', 'QHTQOSGOFQVWDSF', and 'RIURPTHGRWXETG'. On the right, the 'CAESAR CIPHER' section is visible, featuring a 'Caesar Cipher Decoder' tool. This tool includes a text input field with the example text 'FWI{fdhvdv\_kdshu}', a 'KNOWING THE SHIFT' section with a '-10' value, and a 'TEST ALL POSSIBLE SHIFTS (BRUTE-FORCE ATTACK)' section. Below these sections, there are buttons for 'DECRYPT CAESAR CODE' and 'DECRYPT', along with a custom alphabet input field.

Almost all possible classic ciphers (old and new), encoder/decoder, ...

# Cryptanalysis

Often the vulnerability is not in the algorithm but in **its application**...

- ▶ Bad use of the key (too short, reused, bad generated, ...)
- ▶ Messages use a poorly distributed dictionary
- ▶ We know the message format (e.g.: `FLAG{...}`)

In particular we talk about **statistical cryptanalysis** when we force the cipher not from algorithmic point of view but from statistical one

For example in english the character E has a frequency of 12.02% while Z only 0.07%

Useful tool (for substitution ciphers): <https://quipqiup.com>

Puzzle:	giuifg cei iprc tpnn du cei qprcni	
0	-0.842	defend the east wall of the castle
1	-0.859	defend the east ball of the castle
2	-0.915	defend the east mall of the castle

# Attack models

Classification of cryptographic attacks:

- ▶ **Ciphertext-only** attack: access only to the ciphertext, and has no access to the plaintext
- ▶ **Known-plaintext** attack: access to at least a limited number of pairs of plaintext and the corresponding enciphered text
- ▶ **Chosen plaintext** attack: able to choose a number of plaintexts to be enciphered and have access to the resulting ciphertext (encrypt oracle)
- ▶ **Chosen ciphertext** attack: able to choose arbitrary ciphertext and have access to plaintext decrypted from it (decrypt oracle)
- ▶ **Side-channel** attack: use of other informations to break the cipher (time, sound, power, error, ...).



# Table of Contents

- ▶ 1. Introduction
- ▶ 2. Character encoding
- ▶ 3. Classical cryptography
- ▶ 4. **Symmetric-key cryptography**
- ▶ 5. Public-key cryptography
- ▶ 6. Key exchange
- ▶ 7. Hash function
- ▶ 8. Steganography

# Symmetric-key cryptography

Symmetric ciphers are those where messages are encrypted and decrypted using the **same key**, which must be known only and exclusively to the two parts

$\mathcal{C}(m, k) = c$  (encrypt function)

$\mathcal{D}(c, k) = m$  (decrypt function)

Obviously:

$\mathcal{D}(\mathcal{C}(m, k), k) = m$

The original message is **not altered** during the communication

E.g. In the caesar cipher:

$\mathcal{C}(m, k)$  = right shift of  $k$  positions each character

$\mathcal{D}(c, k)$  = left shift of  $k$  positions each character

# Shannon principle

How to assess whether a cipher is robust enough?

(Where robustness means its probability of being successfully attacked)

Shannon defines two key concepts:

- ▶ **Confusion**: the key must be well distributed in the encrypted message (each bit of the cipher should depend on each bit of the key with probability 50%)
- ▶ **Diffusion**: the message must be well distributed in the encrypted message (each bit of the cipher should depend on each bit of the message with probability 50%)

In the Caesar cipher we have no kind of diffusion and low confusion (why?)

# XOR cipher

Consider the **XOR** (exclusive or) operation  $\oplus$ , the following properties are valid:

- ▶  $0 \oplus 0 = 1 \oplus 1 = 0$
- ▶  $0 \oplus 1 = 1 \oplus 0 = 1$
- ▶  $x \oplus y \oplus y = x$

We define the XOR cipher as:

$$\begin{aligned}\mathcal{C}(m, k) &= m \oplus k && (m[i] \oplus k[i], 0 \leq i < ||m||) \\ \mathcal{D}(c, k) &= c \oplus k && (c[i] \oplus k[i], 0 \leq i < ||c||)\end{aligned}$$

Problem: the key  $k$  could be **shorter** than the message  $m$

Solution: reuse the key as  $k' = k \cdot k \cdot \dots \cdot k$  until  $||k'|| \geq m$

Example:

$m = 01100011 \ 01101001 \ 01100001 \ 01101111$  (ciao in ASCII).  
 $k = 01111000 \ 01111000 \ 01111000 \ 01111000$  (x in ascii 4 times)  
 $c = 00011011 \ 00010001 \ 00011001 \ 00010111$  (non printable, GxEZFw== in b64)

# One-time pad

The problem with the XOR cipher is that encrypting repeatedly reusing the same key can leak **statistical informations** of the original message

We call Vernam cipher (or one-time pad) a XOR cipher where the key has the same length of the message.

This cipher is called **perfect** because we have that:

$$P(M = m | C = c) = P(M = m)$$

The probability that  $M$  is a certain message  $m$  knowing that the cipher  $C$  is  $c$  is equal to the probability that  $M$  is a certain message not knowing the cipher (all cipher texts are equiprobable, the encrypted message does not give us any information about the real message)

Nice in theory, but:

- ▶ The key must be exchanged using a secure method (exchange them by *hand*)
- ▶ The key must be generated randomly and not reused (otherwise a many-time pad attack is possible)

# Many-time pad & XorTool

Nice article: [thecrowned.org/the-one-time-pad-and-the-many-time-pad-vulnerability](https://thecrowned.org/the-one-time-pad-and-the-many-time-pad-vulnerability)

**XorTool**: tool for statistical analysis of encrypted messages:

```
root@ddos:~/Desktop/xortool/xortool# xortool binary_xored
The most probable key lengths:
 1: 9.6%
 5: 15.0%
10: 21.7%
15: 9.3%
20: 13.6%
25: 6.0%
30: 9.1%
35: 4.2%
40: 6.6%
50: 5.0%
Key-length can be 5*n
Most possible char is needed to guess the key!
```

Knowing the initial part, we can see words in the message:

```
This is clas*****{*
Do not share*****}
{FLG:ch3ck_e*****
```

Going by trial the final flag is reconstructed:

```
This is classified*****
Do not share the s*****
{FLG:ch3ck_em@il}
```

# Block vs Stream ciphers

## Block ciphers

- ▶ Works with fixed-length groups of bits (called blocks)
- ▶ More memory/time requirements
- ▶ High diffusion and confusion
- ▶ Error propagation
- ▶ Need to handle messages length (padding)

Famous ciphers:

DES

AES

BlowFish

## Stream ciphers

- ▶ Works by encrypt digits one at the time
- ▶ Faster encryption/decryption
- ▶ Low diffusion
- ▶ Low propagation error
- ▶ Need a key stream (usually a shift register)

Famous ciphers:

ChaCha20

Salsa20

LFSR-based

# Padding a message (PKCS#5 & PKCS#7)

How to handle messages of length not multiple of the block size?

Idea: append "some chars" to the message (**padding string**)

## **PKCS#5:**

*The padding string PS shall consist of  $8 - (||M|| \bmod 8)$  octets all having value  $8 - (||M|| \bmod 8)$*

## **PKCS#7:**

*For such algorithms, the method shall be to pad the input at the trailing end with  $k - (l \bmod k)$  octets all having value  $k - (l \bmod k)$ , where  $l$  is the length of the input*

Why  $8 - (||M|| \bmod 8)$  and not  $(||M|| \bmod 8)$ ?

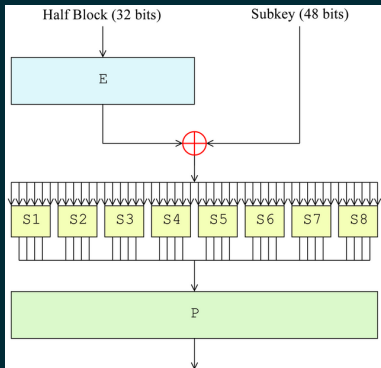


# DES

## Data Encryption Standard

Developed in 1975 by Feistel, encrypt blocks of 64 bits with a 56 bits key

Implements the confusion and diffusion principle by 16 rounds of the **Feistel function**



The Feistel function consists in 4 stages:

- ▶ 1. **Expand** the half block from 32 to 48 bits (E-Box)
- ▶ 2. **Mix** result and subkey using a XOR operation
- ▶ 3. **Substitution** of the 6-bits input with a 4-bits output according to a lookup table (S-Box)
- ▶ 4. **Permutation** of the result (P-Box)

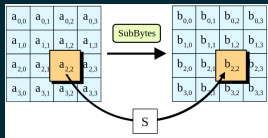
Problem: DES is vulnerable to a (*smart*) brute force attack (only 56 bits for the key...)

# AES

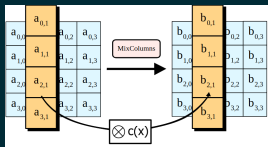
## Advanced Encryption Standard

AES replaced DES starting from 2001 and is currently the standard in secure communications (TLS1.3 supports only AES and ChaCha20)

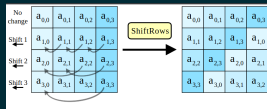
Based on a **substitution-permutation network** (the Feistel network equivalent of DES)



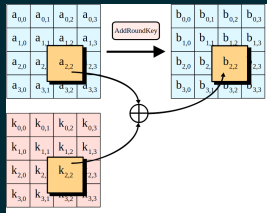
Step 1:  
Each byte is replaced with another according to a lookup table (S-Box)



Step 3:  
Linear mixing operation where each column is mapped into a new one



Step 2:  
Transposition of each rows by 0, 1, 2 or 3 positions



Step 4:  
Each byte is XORed with the corresponding value of the subkey

# Block cipher mode of operation

How to cipher two or more blocks? Different modes, different features:

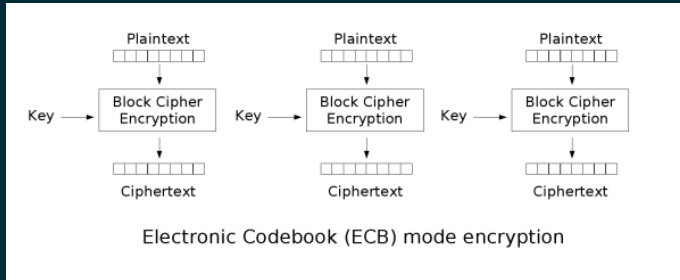
- ▶ **Parallel encryption**: encrypt different blocks at the same time
- ▶ **Parallel decryption**: decrypt different blocks at the same time
- ▶ **Random read**: decrypt any single block without decrypting the previous ones

Mode	Parallel encryption	Parallel decryption	Random read
<b>Electronic Code Book (ECB)</b>	Yes	Yes	Yes
<b>Cipher Block Chaining (CBC)</b>	No	Yes	Yes
Propagating CBC (PCBC)	No	No	No
Cipher Feedback (CFB)	No	Yes	Yes
Output Feedback (OFB)	No	No	No
Counter (CTR)	Yes	Yes	Yes

# ECB (Electronic Code Book)

The message is divided into blocks, and each block is encrypted/decrypted **separately**:

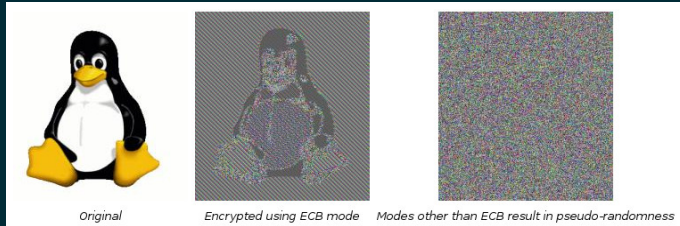
$$C_i = f(M_i, \text{Key})$$
$$M_i = f(C_i, \text{Key})$$



Problem: **no diffusion**, ECB encrypt same plaintext in same ciphertext

# How to break ECB (Chosen-prefix attack)

Problem: no diffusion, ECB encrypt **same plaintext** in **same ciphertext**



How to reverse: `ae69a8467c46bd2f8d30db166ebfc135`  $\rightarrow$  `XXXXXXXX` ?

Idea: `AAAAAAAY` `AAAAAAAX` `XXXXXXXXP`  $\rightarrow c_0 \ c_1 \ c_2$

To find X try all the possible value of Y until  $c_0 = c_1$

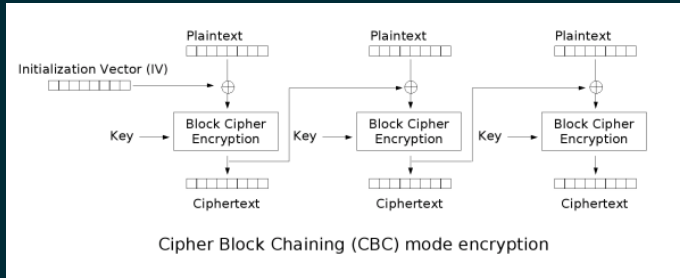
Idea<sup>2</sup>: `AAAAAAXY` `AAAAAAXX` `XXXXXXPP`  $\rightarrow c_0 \ c_1 \ c_2$

To find X try all the possible value of Y until  $c_0 = c_1$

# CBC (Cipher Block Chaining)

In CBC mode each plaintext block is XORed with the **previous ciphertext** block before being encrypted

An initialization vector IV is needed for the first block (usually random generated)



$$C_0 = IV$$

$$C_{i+1} = \mathcal{C}(M_i \oplus C_i, \text{Key})$$

$$M_{i+1} = \mathcal{D}(C_{i+1}, \text{Key}) \oplus C_i$$

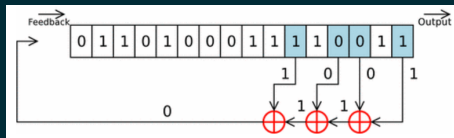
# Stream cipher: LFSR

## Linear-Feedback Shift Registers

Shift register whose input bit is a linear function of its previous state

```
class LFSR:
    def __init__(self, register, branches):
        self.register = register
        self.branches = branches
        self.n = len(register)

    def next_bit(self):
        ret = self.register[self.n - 1]
        new = 0
        for i in self.branches:
            new ^= self.register[i - 1]
        self.register = [new] + self.register[:-1]
        return ret
```



```
register = [0,1,1,0,1,0,0,0,1,1,1,1,0,0,1,1]
branches = [10,12,13,15]
gen = LFSR(register, branches)
for c in stream:
    print(c ^ gen.next_bit())
```

Vulnerability: we can retrieve register and branches if we know a portion of the key stream (via the Berlekamp-Massey algorithm)