2. Secret-key techniques

#chap2

2.1 Settings

The #secret-key comes from: **hand-to-hand distribution**, smartcard, a **key establishment scheme**.

We are going to see **primitives** (require peer review and cryptanalysis):

- #Keystream-generators like RC4, Trivium Salsa20
- #Block-ciphers like DES, AES, Skinny, Saturnin
- #Permutations like KECCAK-f, Ascon, Gimli, XOODOO
  But also the **mode of operations (and constructions)** to build :
- an #encryption-scheme like stream cipher, CBC, CTR, sponge
- an #authentication-scheme like CBC-MAC, sponge
- an #authentication #encryption-scheme like GCM, SIV, SpongeWrap
- a #hash-function like Davies-Meyer + Merkle-Damgard, sponge #chap3
- one primitive to another like CTR turning a block cipher into a keystream generator
  Those can have generic attacks and are analysed assuming the primitive is ideal to determine the generic attacks and to prove that no other generic attack exists.

This **allows** us to get **secure schemes up to the generic attacks** *until someone finds a flaw in the primitive*.

2.2 Keystream generators and stream ciphers

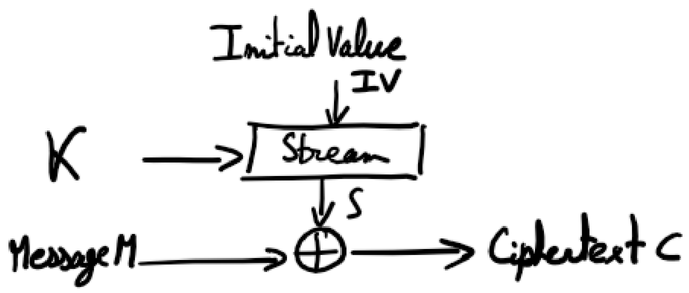
2.2.1 Stream cipher

A #stream-cipher is an #encryption-scheme that uses a #keystream-generators to output bits gradually and on demand. The output bits are called the **keystream**.

- To **encrypt**, the keystream is **XORed** bit by bit *with the plaintext*.
- To **decrypt**, the keystream is **XORed** bit by bit *with the ciphertext*.

A #Keystream-generators : $G : K \times \mathbb{N} \to \mathbb{Z}_2^\infty$

- Inputs a secret key $k \in K$ and a diversifier $d - IV \in \mathbb{N}$ that is public and **must** be a **nonce**.
- Outputs a long keystream $(s_i) \in \mathbb{Z}_2^\infty$



To encrypt plaintext $(m_0, \ldots, m_{|m|-1})$:

$$c_i = m_i + s_i \text{ (in } \mathbb{Z}_2)$$

To decrypt ciphertext $(c_0, \ldots, c_{|c|-1})$:

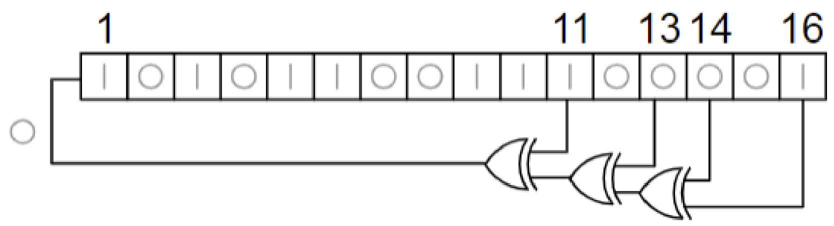
$$m_i = c_i + s_i \text{ (in } \mathbb{Z}_2)$$

If the **diversifier** is **not unique**, then the attacker will **know the difference between the plaintexts** (as in one-time-pad #OTP).
So if we have a #known-plaintext-attack , then all plaintexts will be discovered as a plaintext and the difference are known. #TP2-ex1

2.2.2 Examples

2.2.2.1 Linear feedback shift register (LFSR)

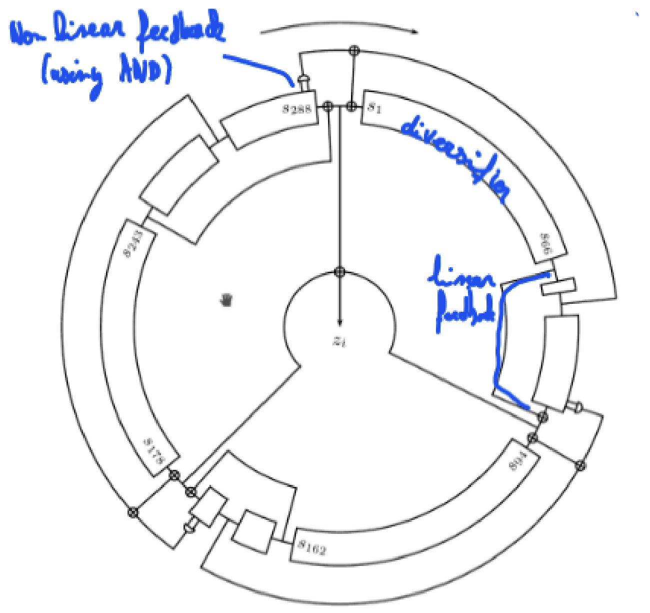
Loop of $2^{16} - 1$ values. Used in the 70's and 80's. Not used anymore. #TP2-ex2



Feedback polynomial: $1 + D^{11} + D^{13} + D^{14} + D^{16}$

2.2.2.2 Trivium

Still used sometimes, non linear feedback linked to linear feedback and diversifier.



2.2.2.3 RC4

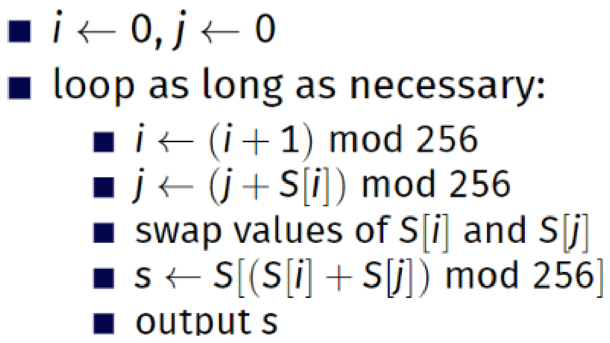
#RC4
$S \in \mathbb{Z}_{256}^{256}$, an array of 256 bytes.
Initialisation ($k \in \mathbb{Z}_{256}^{*}$: key and diversifier)

- for i from 0 to 255
  - $S[i] \leftarrow i$
- $j \leftarrow 0$
- for i from 0 to 255
  - $j \leftarrow (j + S[i] + k[\text{i mod } |\text{k}|]) \bmod 256$

- swap values of $S[i]$ and $S[j]$

The idea is to use every byte of k, the mod 256 allows to stay in the interval [0,255]. We scan the entire array and swap often the values.

Keystream generation



```
■ $i \leftarrow 0, j \leftarrow 0$
■ loop as long as necessary:
    ■ $i \leftarrow (i+1) \bmod 256$
    ■ $j \leftarrow (j+S[i]) \bmod 256$
    ■ swap values of $S[i]$ and $S[j]$
    ■ $s \leftarrow S[(S[i]+S[j]) \bmod 256]$
    ■ output s
```

But it has been broken! Indeed, an example of attack is represented in this slide. The general idea is that if we say that a bit the second element of the table is not 2, we can determine the element by swapping two times.
#TP2-ex3

2.3 Block ciphers

2.3.1 Definition

A #Block-ciphers is a **mapping** $E : \mathbb{Z}_2^n \times \mathbb{Z}_2^m \rightarrow \mathbb{Z}_2^n$

- from a secret key $k \in \mathbb{Z}_2^m$ and an *input block* $x \in \mathbb{Z}_2^n$
- to an *output block* $y = E_k(x) \in \mathbb{Z}_2^n$
  For each key k, it has an inverse: $x = E_k^{-1}(y)$

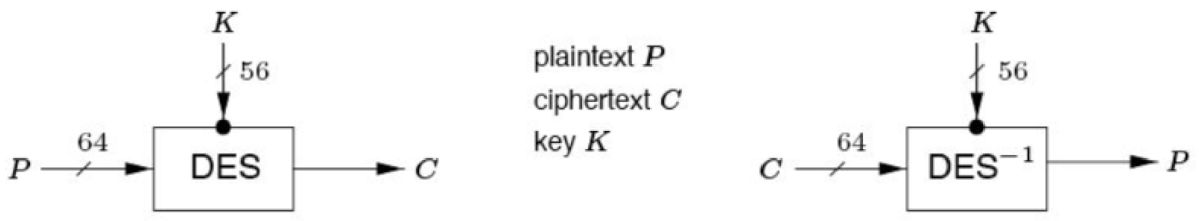
Despite its name, **a block cipher is not an encryption scheme**, except for the very restricted plaintext space $\mathbb{Z}_2^n$ but then it's not even IND-CPA.

For the security notions:

- **Pseudo-random permutation** #PRP : without knowing the secret key, it should be infeasible for an adversary having access to $E_k(.)$ to distinguish it from a permutation randomly drawn from the set of all permutations on $\mathbb{Z}_2^n$.
- **Strong PRP** #SPRP : same, but the adversary also gets access to $E_k^{-1}(.)$.

2.3.2 DES

#DES has been created by Horst Feistel and Don Coppersmith.
DES: $\mathbb{Z}_2^{64} \times \mathbb{Z}_2^{56} \rightarrow \mathbb{Z}_2^{64}$ (plaintext, key -> ciphertext).



DES input-output.

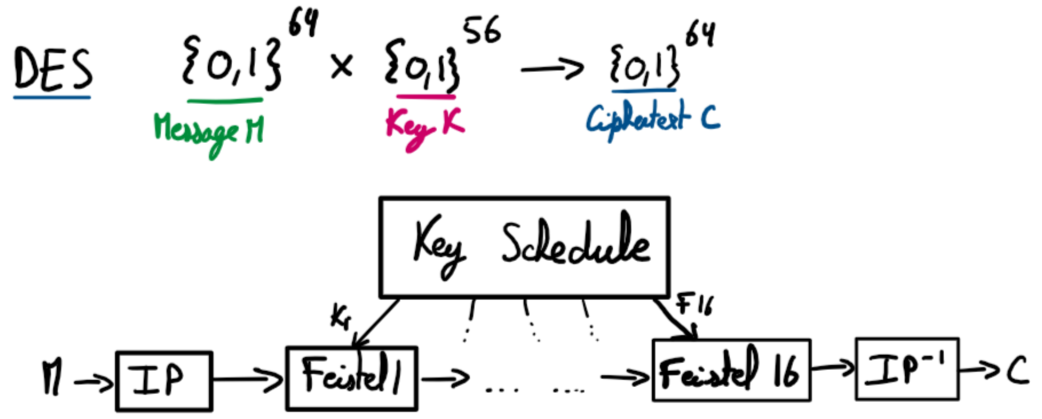
Note that the **input** and **output** are *not always plaintexts or ciphertexts.*

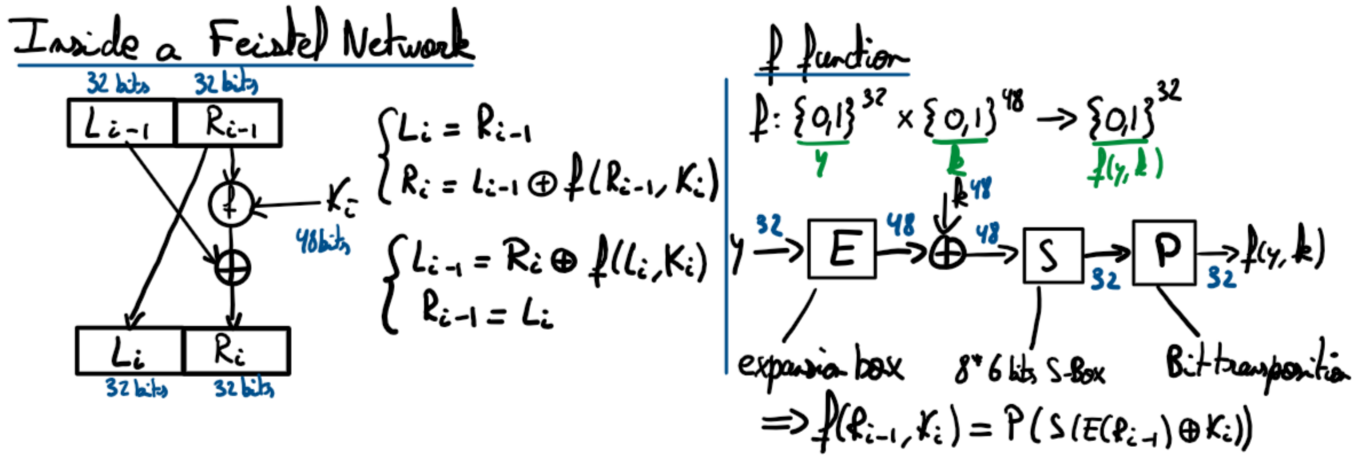
Now, let's see how the black box that is DES works:

- An **initial bit transposition** #IP is applied to the input: $L_0 \mid\mid R_0 = IP(x)$, with $L_0, R_0 \in \mathbb{Z}_2^{32}$
  A #bit-transposition is a function $f : \{0,1\}^n \to \{0,1\}^n$ that **shuffles the bit** in a vector. Note that this function is #linear $\to f(x \oplus y) = f(x) \oplus f(y)$
- **16 iterations of the round function** - #feistel-network :

$$L_i = R_{i-1} \text{ and } R_i = L_{i-1} \oplus f(R_{i-1}, k_i),$$

  where $k_1, \ldots, k_{16} \in \mathbb{Z}_2^{48}$ are the 16 **sub-keys**.
- The **inverse of IP** is applied before returning the output: $y = IP^{-1}(R_{16} \mid\mid L_{16})$.




Inside a Feistel Network
32 bits
32 bits
L_{i-1}
R_{i-1}
f
K_i
48 bits
L_i
R_i
$\begin{cases} L_i = R_{i-1} \\ R_i = L_{i-1} \oplus f(R_{i-1}, K_i) \end{cases}$
$\begin{cases} L_{i-1} = R_i \oplus f(L_i, K_i) \\ R_{i-1} = L_i \end{cases}$
f function
$f: \{0,1\}^{32} \times \{0,1\}^{48} \to \{0,1\}^{32}$
y
k
f(y, k)
y
E
S
P
k 48
f(y, k)
expansion box
8*6 bits S-Box
Bit transposition
$\Rightarrow f(R_{i-1}, K_i) = P(S(E(R_{i-1}) \oplus K_i))$

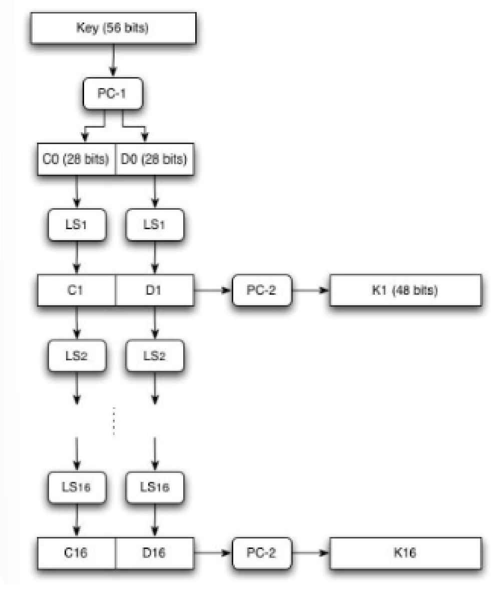
Note that f does **not need to be invertible**. The advantage of this network is that it is possible to do the reverse with the same structure, we just need to *reverse order the subkeys*.

The f function uses #expansion , 6 different #S-Box (that reduces the number of bits) and a #bit-transposition at the end.

There are different tables for the permutations, expansions and S-Boxes.

DES key schedule

From the secret key $k \in \mathbb{Z}_2^{56}$ to round keys $K_i \in \mathbb{Z}_2^{48}$



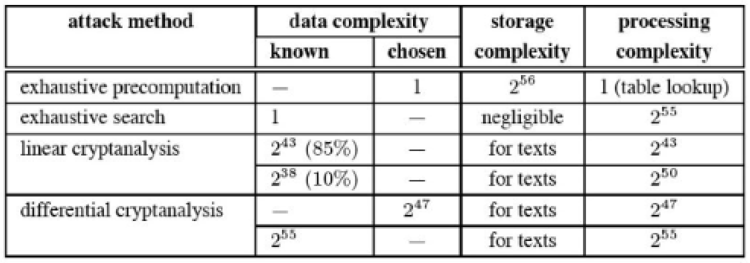
1. Bit transposition $C_0 \| D_0 = \mathrm{PC1}(k)$
   $\mathrm{PC1} : \mathbb{Z}_2^{56} \to \mathbb{Z}_2^{28} \times \mathbb{Z}_2^{28}$
2. For $i = 1$ to 16:
   - $C_i = LS_i(C_{i-1})$ and $D_i = LS_i(D_{i-1})$, where LS_i is a circular shift to the left by one position when $i = 1, 2, 9, 16$ and by two positions otherwise
   - $K_i = \mathrm{PC2}(C_i \| D_i)$, with $\mathrm{PC2} : \mathbb{Z}_2^{56} \to \mathbb{Z}_2^{48}$

DES non-ideal properties

- There are 4 weak keys k: E_k is an #involution $\to \forall x \; E_k(E_k(x)) = x$
  This means that the #encryption and #decryption are the same.
- There are **6 pairs of semi-weak keys** (k_1, k_2) such that $\forall x \; E_{k_1}(E_{k_2}(x)) = x$
  Almost involution but not because keys need to change.
- The **complementarity property**: $E_k(x) = y \Leftrightarrow E_{\overline{k}}(\overline{x}) = \overline{y}$
  $\to$ reduces the exhaustive search security by one bit.
  #TP2-ex5

DES security

The #Exhaustive-key-search works in 2^{55} because the adversary can compute with the inverted key's to gain some time. Which is feasible in 1 day with some dedicated HW or FPGA.
There are other methods that takes less time such as **differential cryptanalysis** or **linear cryptanalysis**.



attack method	data complexity		storage	processing
	known	chosen	complexity	complexity
exhaustive precomputation	—	1	2^{56}	1 (table lookup)
exhaustive search	1	—	negligible	2^{55}
linear cryptanalysis	2^{43} (85%)	—	for texts	2^{43}
	2^{38} (10%)	—	for texts	2^{50}
differential cryptanalysis	—	2^{47}	for texts	2^{47}
	2^{55}	—	for texts	2^{55}

Table 7.7: *DES strength against various attacks.*

Differential cryptanalysis analyses the propagation through the rounds:

$$f(x) = y$$
$$f(\delta \oplus x) = y + \Delta$$

(x varies, δ constant)

Linear cryptanalysis analyses the correlation between input and output parities:

$$\left| \Pr[v^\top x = u^\top f(x)] - \frac{1}{2} \right|$$

(*x* varies, *u* and *v* constant)

Triple-DES

In order to have a more secure #DES, we have the different #triple-DES. It consists in 3 DES linked in a chain.

Triple-key triple-DES (168-bit keys)

$$3\mathrm{DES}_{k_1\|k_2\|k_3} = \mathrm{DES}_{k_3} \circ \mathrm{DES}_{k_2}^{-1} \circ \mathrm{DES}_{k_1}$$

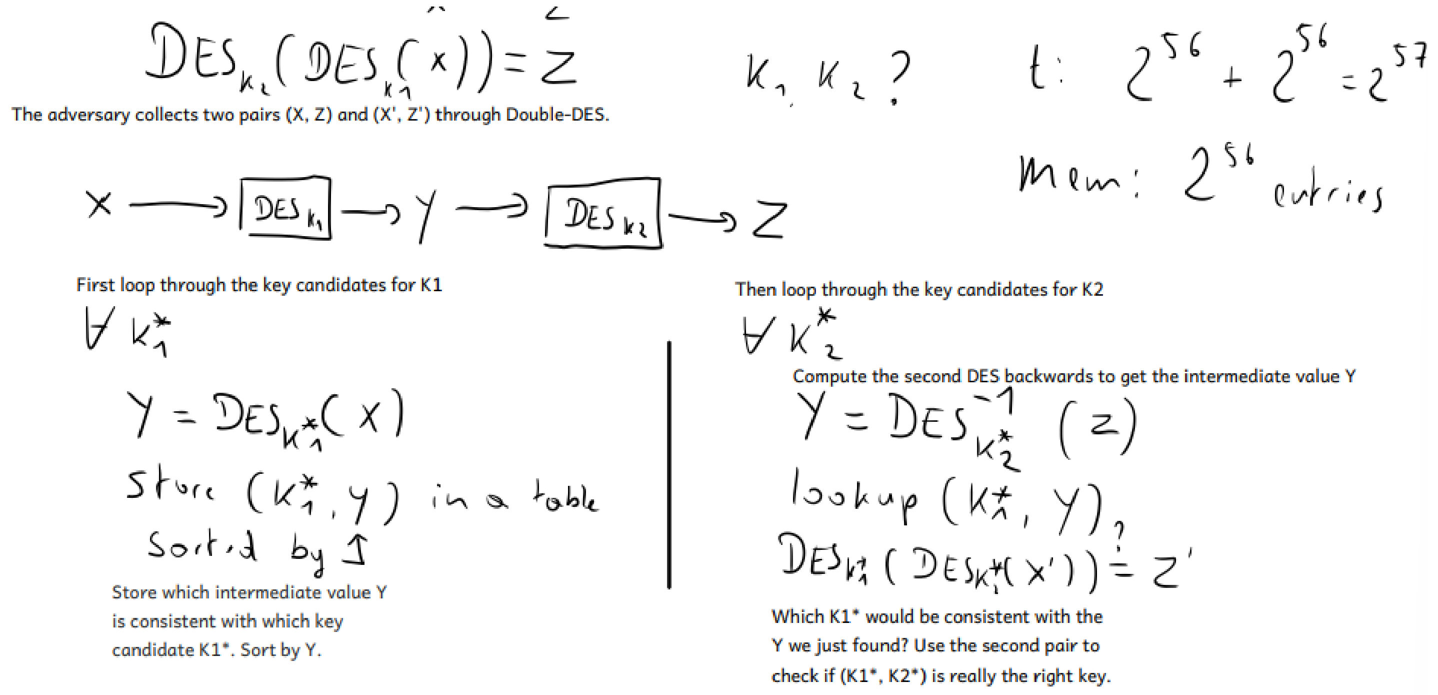
Double-key triple-DES (112-bit keys)

$$3\mathrm{DES}_{k_1\|k_2} = \mathrm{DES}_{k_1} \circ \mathrm{DES}_{k_2}^{-1} \circ \mathrm{DES}_{k_1}$$

Note that there is **retro-compability** with single DES, if a same key is used for a triple-DES, we have the same as if we only used 1 DES.

Not double DES

We do not use the double DES technique because it is not a lot more secure. Indeed it takes 2^{57} in attack time and 2^{56} memory. That is because the **meet in the middle attack**. It consists in attacking the first and the last DES (in reverse for the last) and compare the results in the middle.



$$\mathrm{DES}_{k_2}(\mathrm{DES}_{k_1}(x)) = z$$

The adversary collects two pairs (X, Z) and (X', Z') through Double-DES.

k_1, k_2 ?

$t: 2^{56} + 2^{56} = 2^{57}$

Mem: 2^{56} entries

$$\begin{tikzcd} x \arrow[r] & \boxed{\mathrm{DES}_{k_1}} \arrow[r] & y \arrow[r] & \boxed{\mathrm{DES}_{k_2}} \arrow[r] & z \end{tikzcd}$$

First loop through the key candidates for K1

$\forall k_1^*$

$y = \mathrm{DES}_{k_1^*}(x)$

store (k_1^*, y) in a table

sorted by y

Store which intermediate value Y is consistent with which key candidate K1*. Sort by Y.

Then loop through the key candidates for K2

$\forall k_2^*$

Compute the second DES backwards to get the intermediate value Y

$y = \mathrm{DES}_{k_2^*}^{-1}(z)$

lookup (k_1^*, y),

$\mathrm{DES}_{k_2^*}(\mathrm{DES}_{k_1^*}(x')) \overset{?}{=} z'$

Which K1* would be consistent with the Y we just found? Use the second pair to check if (K1*, K2*) is really the right key.

2.3.3 Rijndael and AES

Finite Field

#Finite-field are **algebraic structures** in which one can **apply the basic operations** of *addition, subtraction, multiplication and division* with their **usual properties** (such as distribution).
Example: $GF(5)$ also noted $\mathbb{Z}/5\mathbb{Z}$ or $\mathbb{F}_5$.
- Addition: $4+3=2 \bmod 5$
- Subtraction: $2-3=4 \bmod 5$
- Multiply: $3\cdot 2=1 \bmod 5$
- Divide: $4/2=4\cdot 2^{-1}=4\cdot 3=2 \bmod 5$

For any $n\in\mathbb{Z}$, the three first operations exists in $\mathbb{Z}/n\mathbb{Z}$ and those structures are called #rings .

Division is **fully defined** when *all nonzero elements have an inverse*, which is only the case when n is a #prime (chap 4).

All the finite fiels have p^i elements for a prime p and the particular case of **prime order fields** ($i=0$) are the $\mathbb{Z}/p\mathbb{Z}$ are written $GF(p)$ or $\mathbb{F}_p$.

To **construct final fields of non prime orders** p^i for $i\neq 0$, one has to work with #polynomials over $GF(p)$ with operations #modulo an irreducible polynomials of degree i.

GF(2)[x] ring

Addition in GF(2)	Multiplication in GF(2)
$0+0=0$	$0\cdot 0=0$
$0+1=1$	$0\cdot 1=0$
$1+0=1$	$1\cdot 0=0$
$1+1=0$	$1\cdot 1=1$

Encoding $(0x7B)_{16}=(0111\ 1011)_2$

$= 0x^7+1x^6+1x^5+1x^4+1x^3+0x^2+1x+1$

$= x^6+x^5+x^4+x^3+x+1 \in \mathbb{F}_2[x]$

Addition $(x^3+x^2+1)+(x^2+x+1)$

$\quad 1101 + 0111$

$=\overbrace{(1+0)}^{1}x^3+\underbrace{(1+1)}_{0}x^2+\underbrace{(0+1)}_{1}x+\underbrace{(1+1)}_{0}\cdot 1$

$= x^3+x$

$\Rightarrow 0xD + 0x7 = 0xA \in GF(2)[x]$

Multiplication $(x^2+x)\cdot(x^6+1)$

$= x^2\cdot x^6+x^2\cdot 1+x\cdot x^6+x\cdot 1$

$= x^8+x^2+x^7+x$

$= x^8+x^7+x^2+x$

$\Rightarrow 0x6 \cdot 0x41 = 0x186 \in GF(2)[x]$

Modular reduction of polynomials

For a given polynomial $m(x)$. A polynomial $a(x)$ can always be written

$a(x)=b(x)\cdot m(x)+r(x)$ ~ division + rest

$\rightarrow$ degree of $r(x)$ < degree of $m(x)$

$\Rightarrow$ $a(x)$ is congruent to $r(x)$ modulo $m(x)$

$\Rightarrow$ $r(x)\equiv a(x) \bmod m(x)$

To reduce $a(x)$ modulo $m(x)$

$\rightarrow$ repeatedly add/subtract a multiple of $m(x)$ to make the highest degree term disappear until a polynomial of degree less tan the degree of $m(x)$ is obtained

$m(x)=x^8+x^4+x^3+x+1 \qquad a(x)=x^8+x^7+x^3+1$

$\Rightarrow r(x)=a(x) \bmod m(x)=(\cancel{x^8}+x^7+\cancel{x^3}+1)+(\cancel{x^8}+x^4+\cancel{x^3}+x+1)$

$= x^7+x^4+x$

$a(x)=x^{10}+x^9+x^8$

$\Rightarrow r(x)=(x^{10}+x^9+x^8) \bmod (x^8+x^4+x^3+x+1)$

add $x^2\cdot m(x)$ $= x^2\cdot(x^8+x^4+x^3+x+1)+x^{10}+x^9+x^8$

$= \cancel{x^{10}}+x^6+x^5+x^3+\cancel{x^{10}}+x^9+x^8+x^2$

add $(x+1)\,m(x)$ $= \cancel{x^9}+\cancel{x^9}+x^6+\cancel{x^5}+\cancel{x^3}+\cancel{x^2}+\cancel{x(x^8}+\cancel{x^4}+\cancel{x^3}+\cancel{x}+\cancel{1})+\cancel{x^8}+\cancel{x^4}+\cancel{x^3}+\cancel{x}+1$

$= x^6+1$

GF(2^8) and calculations

Let $m(x)=x^8+x^4+x^3+x+1$ be the irreducible polynomial used in Rijndael

$\Rightarrow x^8=x^4+x^3+x+1$ is used a lot to simplify big polynomials

Multiplication

- Muliply in $GF(2)[x]$
- Reduce the result modulo $m(x)$

$(x^2+x+1)x^7 = \underline{x^9}+\underline{x^8}+x^7$

$= (x(x^4+x^3+x+1))+(x^4+x^3+x+1)+x^7$

$= x^7+x^5+x^3+x^2+1$

Multiplicative inverse

- Every non zero element has a multiplicative inverse

$\rightarrow a\cdot a^{-1}=1 \quad \forall a\in R\setminus\{0\}$

$\rightarrow$ There always exists another polynomial $a^{-1}\rightarrow a\times a^{-1}=1$

$x^{-1}=x^7+x^3+x^2+1 \rightsquigarrow x\cdot x^{-1}=x^8+x^4+x^3+x$

$= x^4+x^3+x+1+x=1$

$x^{-2}=x^7+x^6+x^3+x+1 \rightsquigarrow x^2\cdot x^{-2}=x^9+x^8+x^5+x^3+x^2$

$= \cancel{x^6}+\cancel{x^4}+\cancel{x^2}+\cancel{x}+\cancel{x^4}+\cancel{x^3}+\cancel{x}+1+\cancel{x^5}+\cancel{x^3}+\cancel{x^2}$

$= 1$

AES

In order to have a more performant and secure block cipher, a competition has been organised. It was to get a new #Block-ciphers with a block-size of $n=128$ bits and a key size of $m=128, 192$ and 256.

So there are 3 instances of Rijndael with different block sizes:

AES-128 : $\mathbb{Z}_2^{128} \times \mathbb{Z}_2^{128} \to \mathbb{Z}_2^{128}$	10 rounds
AES-192 : $\mathbb{Z}_2^{128} \times \mathbb{Z}_2^{192} \to \mathbb{Z}_2^{128}$	12 rounds
AES-256 : $\mathbb{Z}_2^{128} \times \mathbb{Z}_2^{256} \to \mathbb{Z}_2^{128}$	14 rounds

Rijndael - $\mathbb{Z}_2^{128}$

The **input** $x \in \mathbb{Z}_2^{128}$ is mapped to an array of 4×4 bytes where x_i is the i^{th} byte of x. And vice-versa for the output. Each byte $s_{i,j}$ represents an element of the **finite field** GF(2^8).

$$\begin{pmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{pmatrix} \leftarrow \begin{pmatrix} x_0 & x_4 & x_8 & x_{12} \\ x_1 & x_5 & x_9 & x_{13} \\ x_2 & x_6 & x_{10} & x_{14} \\ x_3 & x_7 & x_{11} & x_{15} \end{pmatrix}$$

This #finite-field (finite -> linearity, distributivity, etc...) is of **size 256**.

Rijndael uses the representation $GF(2)\ [x]/[x^8 + x^4 + x^3 + x + 1]$.
A **byte with arithmetic value** $s = \Sigma_{n=0}^{7} s_i 2^i$ represents the polynomial $s = \Sigma_{n=0}^{7} s_i x^i$ in $GF(2)[x]$.
(See exercice session).
The #operations are the following:

- Addition is like a bitwise XOR operations
  E.g., $(x^7 + x + 1) + (x^6 + x) = (x^7 + x^6 + 1) \Leftrightarrow$ `0x83` $\oplus$ `0x42` $=$ `0xC1`
- Multiplication is done modulo $x^8 + x^4 + x^3 + x + 1$
  E.g., $x(x^7 + x + 1) = x^8 + x^2 + x = (x^8 + x^2 + x) + (x^8 + x^4 + x^3 + x + 1) = x^4 + x^3 + x^2 + 1$

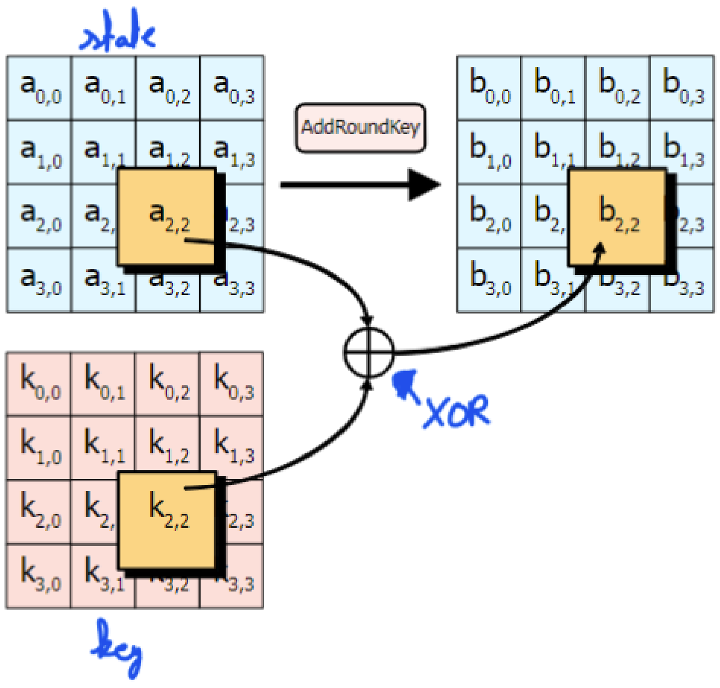
AES

The #AES #data-path :
Input $y \in \mathbb{Z}_2^{128}$ and key $k \in \mathbb{Z}_2^{128 \text{(or 192 or 256)}}$

- **Key schedule**: $(K_i) \leftarrow KeyExpansion(k)$
- **Write the input as a state in** $GF(2^8)^{4\times4}$
- **AddRoundKey** : $State$ XOR K_0
- For each round $i = 1$ to 9 (or 11 or 13)
  - **SubBytes** (state)
  - **ShiftRows** (state)
  - **MixColumns** (state)
  - **AddRoundKey** (state, K_i)
- Last round:
  - **SubBytes** (state)
  - **ShiftRows** (state)
  - **AddRoundKey** (state, K_{10}) (or 12 or 14)
- Output $y \leftarrow$ state back in $\mathbb{Z}_2^{128}$.

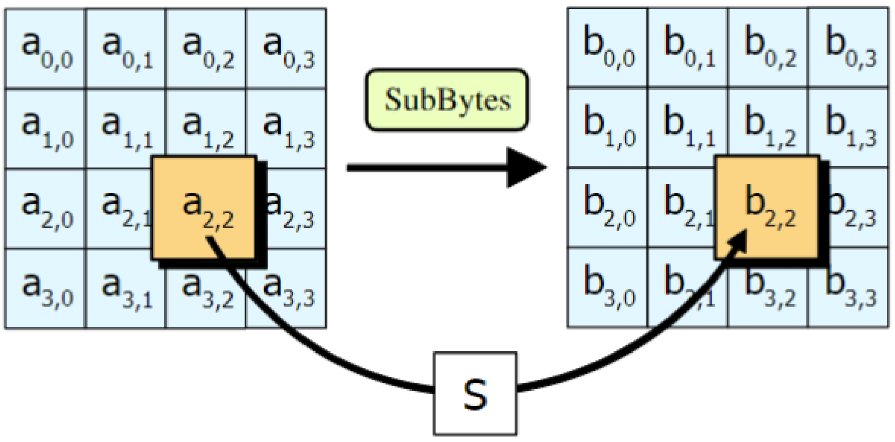
AddRoundKeys

This step consists in XORing the state to a key received from the key schedule.

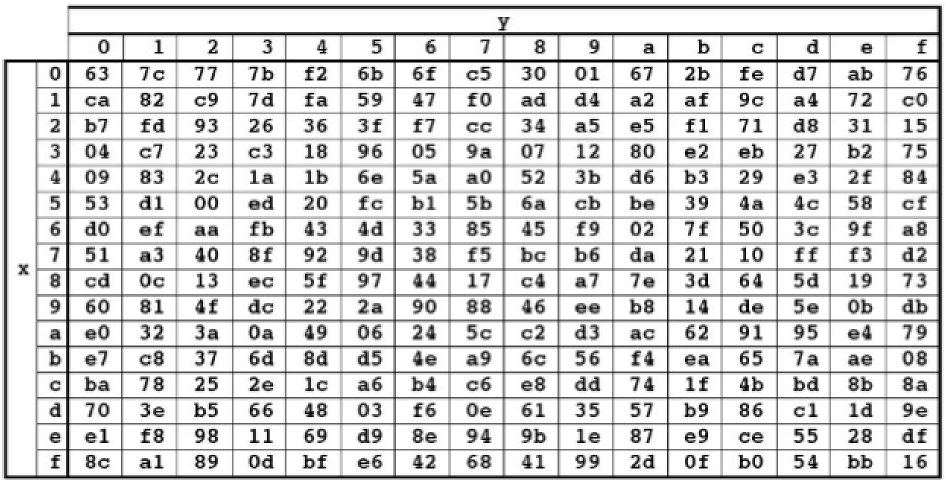


SubBytes

The **SubBytes** step in the Rijndael (AES) encryption algorithm involves substituting each byte in the input matrix with a corresponding byte from the S-box, a predefined substitution table. This introduces non-linearity and enhances the algorithm's resistance to attacks. During decryption, an inverse SubBytes step is performed using the inverse S-box.



Here is the S-Box:

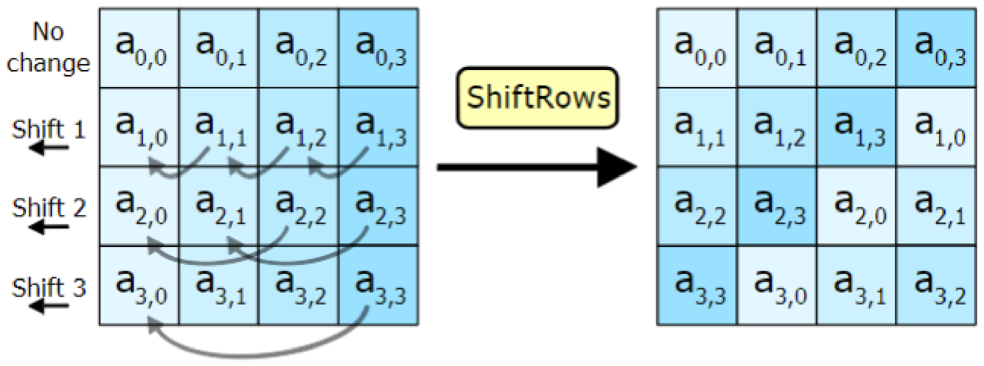


		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
x	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 7. S-box: substitution values for the byte xy (in hexadecimal format).

ShiftRows

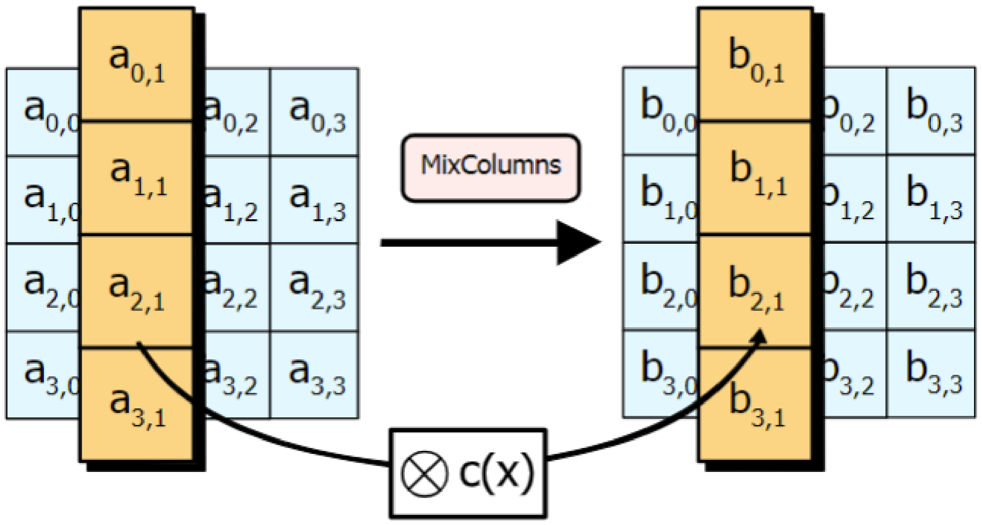
In the **ShiftRows** step, the bytes in each row of the matrix are shifted to the left. The *first row is not shifted*, *the second row is shifted one position to the left*, the *third row is shifted two positions to the left*, and the *fourth row is shifted three positions to the left*.


No change
Shift 1
Shift 2
Shift 3
ShiftRows
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,0}$
$a_{2,2}$ $a_{2,3}$ $a_{2,0}$ $a_{2,1}$
$a_{3,3}$ $a_{3,0}$ $a_{3,1}$ $a_{3,2}$

MixColumns

Each column undergoes the following matrix multiplication.

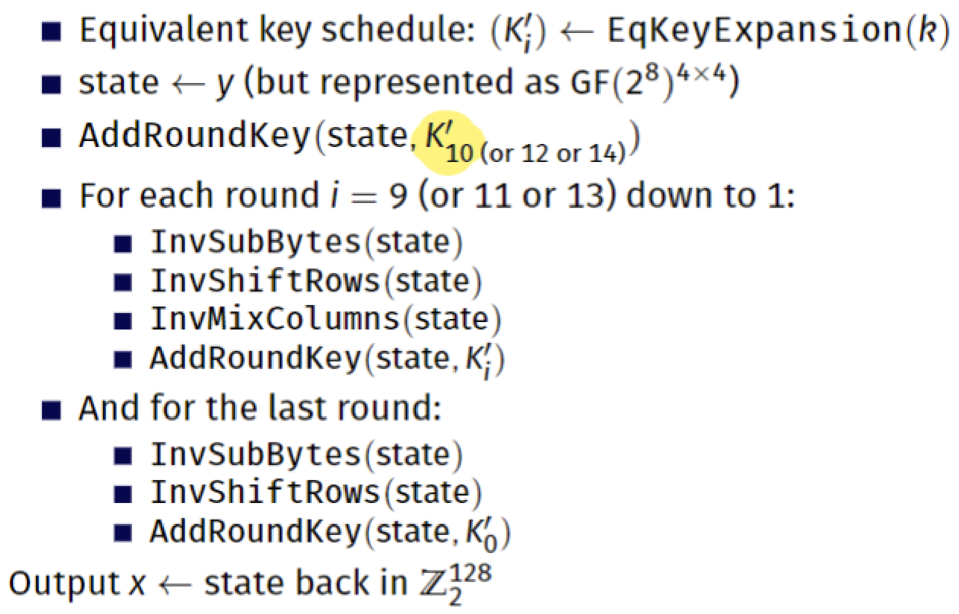
$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} x & x+1 & 1 & 1 \\ 1 & x & x+1 & 1 \\ 1 & 1 & x & x+1 \\ x+1 & 1 & 1 & x \end{pmatrix} \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}$$



$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$
MixColumns
$\otimes$ c(x)
$b_{0,0}$ $b_{0,1}$ $b_{0,2}$ $b_{0,3}$
$b_{1,0}$ $b_{1,1}$ $b_{1,2}$ $b_{1,3}$
$b_{2,0}$ $b_{2,1}$ $b_{2,2}$ $b_{2,3}$
$b_{3,0}$ $b_{3,1}$ $b_{3,2}$ $b_{3,3}$

Rijndael inverse data path

This is very often used:



- Equivalent key schedule: $(K'_i) \leftarrow$ `EqKeyExpansion`(k)
- state $\leftarrow y$ (but represented as $GF(2^8)^{4\times 4}$)
- `AddRoundKey`(state, $K'_{10 \text{ (or 12 or 14)}}$)
- For each round $i = 9$ (or 11 or 13) down to 1:
  - `InvSubBytes`(state)
  - `InvShiftRows`(state)
  - `InvMixColumns`(state)
  - `AddRoundKey`(state, K'_i)
- And for the last round:
  - `InvSubBytes`(state)
  - `InvShiftRows`(state)
  - `AddRoundKey`(state, K'_0)

Output $x \leftarrow$ state back in $\mathbb{Z}_2^{128}$

Note that the inverse of the matrix used for MixColumns is the following:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} \texttt{0E} & \texttt{0B} & \texttt{0D} & \texttt{09} \\ \texttt{09} & \texttt{0E} & \texttt{0B} & \texttt{0D} \\ \texttt{0D} & \texttt{09} & \texttt{0E} & \texttt{0B} \\ \texttt{0B} & \texttt{0D} & \texttt{09} & \texttt{0E} \end{pmatrix} \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}$$

2.3.4 Modes of operations

A #mode-of-operation allows to go from a #block-ciphers to an #encryption-scheme that **works for plaintexts of any lengths** or to an #authentication-scheme that **works for messages of any lengths**.

This is achieved by repeatedly connect block ciphers together to encrypt/decrypt the data. For example the output of a block cipher becomes the input of another one etc.

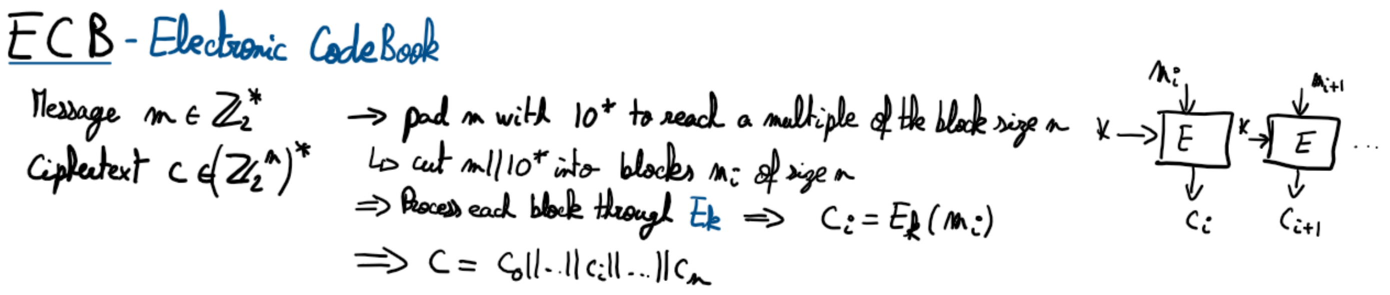
Take a **message** M, a **key** K and split the message into r small messages $M_0 \mid\mid M_1 \mid\mid \ldots \mid\mid M_r$. Then #encryption is made by multiple #block-ciphers depending on the #mode-of-operation. Here are some examples:

- #ECB , #CBC , #CTR ,...

Similarly, the #decryption can also be made.

Bad example: ECB

An #ECB (Electronic Codebook) is a bad example of #mode-of-operation :



ECB - Electronic CodeBook

Message $m \in \mathbb{Z}_2^*$

Ciphertext $c \in (\mathbb{Z}_2^n)^*$

$\rightarrow$ pad m with 10^* to reach a multiple of the block size n

↳ cut $m||10^*$ into blocks m_i of size n

$\Rightarrow$ Process each block through E_k $\Rightarrow$ $C_i = E_k(m_i)$

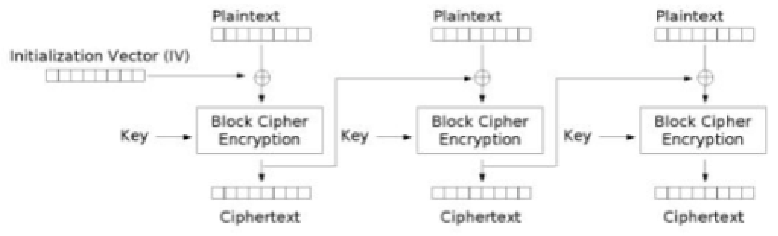
$\Rightarrow$ $C = C_0||..||c_i||...||C_n$

m_i
m_{i+1}
k
E
k
E
...
C_i
C_{i+1}

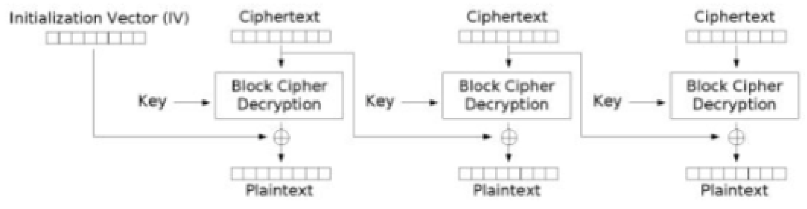
#ECB is considered **insecure** because it *encrypts identical plaintext blocks to identical ciphertext blocks,* **revealing patterns and making it susceptible to analysis**. This **determinism and lack of diffusion** make it **less secure than other block cipher** modes like CBC or GCM, which provide better security properties.
To win #IND-CPA-secure game with an #ECB #encryption (because it is **deterministic**),
The attacker:
- Chooses two arbitrary distinct strings A and B whose length is equal to the underlying block cipher.
- Queries the encryption of $A||B$ and receives the ciphertext $X||Y$
- Then submits $m_0 = A$ and $m_1 = B$ to the challenger
- If $c = X \rightarrow m = m_0$ else, $m = m_1$. The **attackers always wins!**
- This exercise comes from #TP2-ex8

CBC

#CBC (Ciphertext block chaining) is a block cipher #mode-of-operation where **each plaintext block is XORed with the previous ciphertext block before encryption**. This *introduces diffusion* and **prevents identical plaintext blocks from producing the same ciphertext**, addressing vulnerabilities associated with modes like ECB. The use of an **initialization vector (IV) ensures uniqueness in the encryption process**.



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

Input: secret key $k \in \mathbb{Z}_2^m$, plaintext $p \in \mathbb{Z}_2^*$ and diversifier $d \in \mathbb{N}$
Output: ciphertext $c \in (\mathbb{Z}_2^n)^*$

- Pad p with 10^* and cut into blocks $(p_1, p_2, \dots)$
- Define $c_0 = E_k(d)$ and then for $i \geq 1$

$$c_i = E_k(p_i \oplus c_{i-1})$$

To decrypt:

$$p_i = E_k^{-1}(c_i) \oplus c_{i-1}$$

Here again, the diversifier d must be a **nonce**.

There are **limitations**, if the *same key for two different inputs give the same ciphertext blocks*, then *information is revealed on the plaintext*.

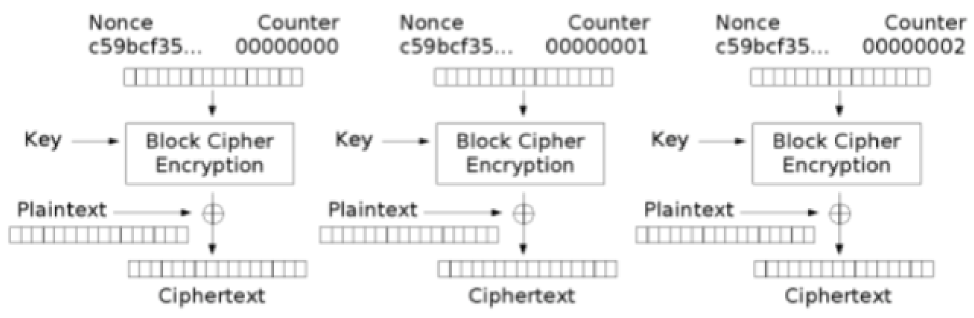
$$\begin{aligned} c_i &= c'_j \\ E_k(p_i \oplus c_{i-1}) &= E_k(p'_j \oplus c'_{j-1}) \\ p_i \oplus c_{i-1} &= p'_j \oplus c'_{j-1} \\ p_i \oplus p'_j &= c_{i-1} \oplus c'_{j-1} \end{aligned}$$

For that to happen, we must have $c_i = c'_j$ and this becomes **very likely as the number of blocks encrypted under the same key reaches the** *order of* $2^{n/2}$, with n the block size.

This is more of a problem for #DES (2^{32} blocks) than for #AES (2^{64} blocks).
#TP2-ex10

CTR

A #CTR (counter) is a block cipher of #mode-of-operation where **a counter value is encrypted and XORed with the plaintext to generate ciphertext**. This *allows for **parallel encryption and decryption, offering efficiency***. The counter value, combined with a nonce or initialization vector, ensures uniqueness for each block, providing security benefits.



Counter (CTR) mode encryption

Input: secret key $k \in \mathbb{Z}_2^m$, plaintext $p \in \mathbb{Z}_2^*$ and diversifier $d \in \mathbb{N}$
Output: ciphertext $c \in \mathbb{Z}_2^*$

- Cut into blocks of n bits $(p_1, p_2, \ldots, p_x)$, except for the last one that can be shorter
- Generate the keystream $k_i = E_k(d \| i)$

$$c_i = k_i \oplus p_i$$

Note: For the last block, the keystream block k_x is truncated to the size of p_x.

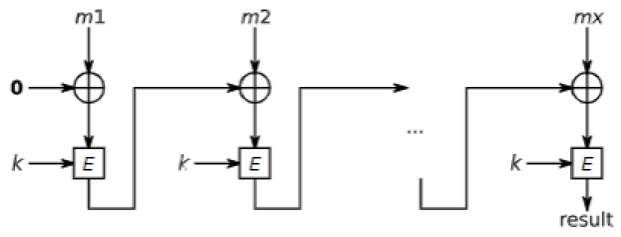
To decrypt, generate the same keystream and

$$p_i = k_i \oplus c_i$$

#TP2-ex9

CBC-MAC

#CBC-MAC (Cipher Block Chaining Message Authentication Code) is a construction that *uses Cipher Block Chaining in a modified way to produce a fixed-size authentication tag for a message*. It involves **encrypting the entire message with a block cipher in CBC mode**, and the **final block's output serves as the authentication tag**. CBC-MAC *provides integrity and authenticity* but *should be used carefully to avoid vulnerabilities, such as avoiding reuse of keys for encryption and authentication*.



Input: secret key $k \in \mathbb{Z}_2^m$, message $m \in \mathbb{Z}_2^*$
Output: MAC $\in \mathbb{Z}_2^n$

- Prepend m with its length
- Pad $\text{len}(m) \| m$ with 10^* and cut into blocks $(m_1, m_2, \ldots, m_x)$
- Define $z_0 = 0^n$ and compute

$$z_i = E_k(m_i \oplus z_{i-1})$$

- Output MAC $= z_x$

Authenticated encryption

The generic method of encrypting first and then applying a Message Authentication Code (MAC) is a common approach to achieve confidentiality and integrity in cryptographic systems. Here's an explanation of this method and how specific modes like CCM and GCM implement it:

1. **Encrypt-then-MAC Generic Method:**
   - **Encryption (with k1):** The plaintext is encrypted using a symmetric key (k1) and a block cipher in a specific mode of operation (e.g., CTR or CBC).
   - **MAC the ciphertext (with k2):** The resulting ciphertext is then authenticated using a MAC algorithm with a different key (k2), ensuring data integrity and authenticity.
2. **CCM (Counter with CBC-MAC):**

- **Encryption (with k):** CCM combines CTR mode for encryption with a block cipher and CBC-MAC for authentication. It encrypts the plaintext using a symmetric key (k) and CTR mode.
- **MAC the ciphertext (with k):** The CBC-MAC is then applied to the ciphertext to generate an authentication tag, ensuring integrity and authenticity.

3. **GCM (Galois/Counter Mode):**
   - **Encryption and Authentication (with k):** GCM combines CTR mode for encryption with a polynomial MAC in Galois Field (GF(2^128)). It encrypts the plaintext using a symmetric key (k) and CTR mode.
   - **Automatic Authentication:** GCM inherently provides authentication during the encryption process, generating an authentication tag without the need for a separate MAC step. This offers efficiency and simplicity.

In both CCM and GCM, the combination of encryption and authentication in a single step provides a secure and efficient way to protect data. These modes address the vulnerabilities associated with using separate keys for encryption and MAC, ensuring that the overall security of the system is maintained.

Birthday paradox

#birthday-paradox is the following: #TP3-ex2
Say we have 2^n objects and we make L draws with replacement. How much does L need to be to have a good chance to get a collision?
- After L draws, there are $\frac{L}{2} = \frac{L(L-1)}{2} \approx \frac{L^2}{2}$ pairs
- For each pair, the probability of getting the same object is 2^{-n}
- The probability of collision is therefore: $\approx \frac{L^2}{2^{n+1}}$
- When $L = \sqrt{2^n} = 2^{\frac{n}{2}}$, this probability is $\approx \frac{1}{2}$
$\implies L \approx 2^{n/2}$

2.4 Permutations

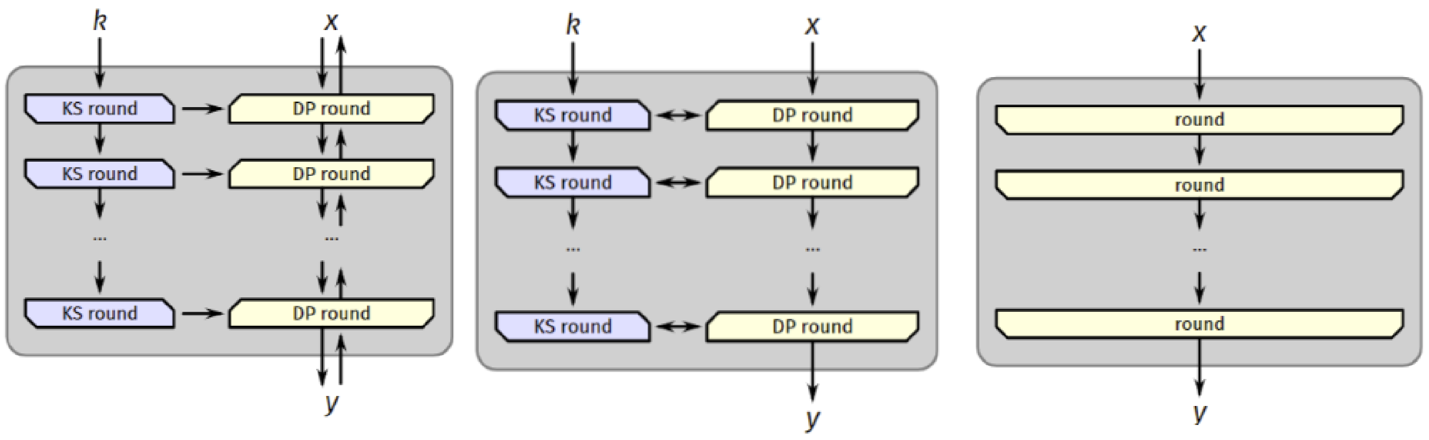
A cryptographic #permutation is a *bijective mapping*

$$f : \mathbb{Z}_2^b \to \mathbb{Z}_2^b$$

from an input block $x \in \mathbb{Z}_2^b$ to an output block $y = f(x) \in \mathbb{Z}_2^b$. It also has an inverse $x = f^{-1}(y)$.

2.4.1 Block ciphers to permutations

#permutation are employed to perform various operations like substitution, transposition, and diffusion within #Block-ciphers .

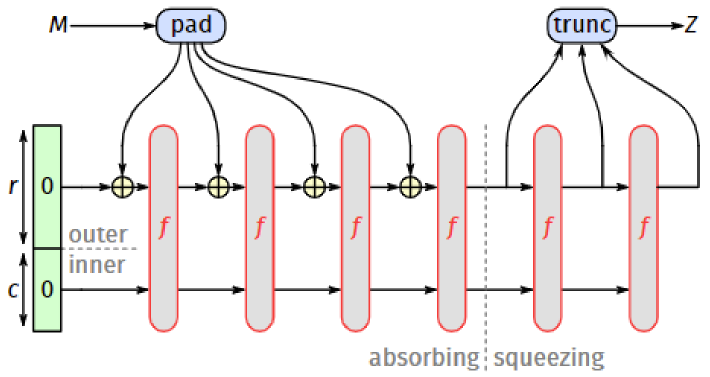


2.4.2 The sponge construction

The #sponge construction is a cryptographic framework *used in the design of hash functions and authenticated encryption algorithms*. It **operates on a fixed-size input block and absorbs data into an internal state through a permutation function**.

The **construction alternates** between **absorbing** and **squeezing phases,** *providing flexibility and security*. The **final output is obtained by squeezing the internal state**. #TP2-ex12

This metaphor is quite good: *you fill the sponge with your message, and then you squeeze it to obtain your hash result.*



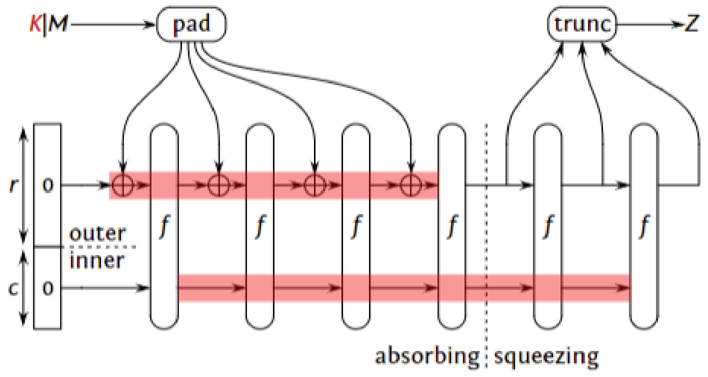
- Calls a b-bit permutation f, with $b = r + c$
  - r bits of *rate*
  - c bits of *capacity* (security parameter)

A #sponge-function is a concrete instance with a given f, r, c. It **implements a mapping from** $\mathbb{Z}_2^*$ to $\mathbb{Z}_2^\infty$ truncated at an arbitrary length.

```
- $s \leftarrow 0^b$
- $M||10^*1$ is cut into $r$-bit blocks
- For each $M_i$ do (absorbing phase)
    - $s \leftarrow s \oplus (M_i||0^c)$
    - $s \leftarrow f(s)$
- As long as output is needed do (squeezing phase)
    - Output the first $r$ bits of $s$
    - $s \leftarrow f(s)$
```

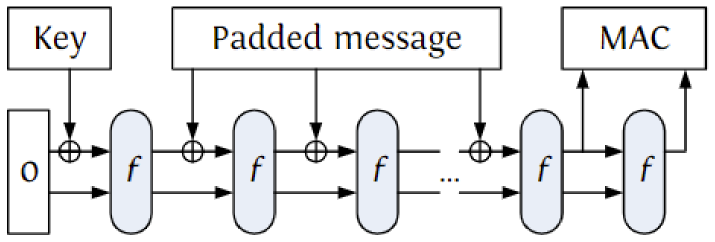
2.4.3 Keyed sponge

A #keyed-sponge function **implements a mapping from** $\mathbb{Z}_2^m \times \mathbb{Z}_2^*$ to $\mathbb{Z}_2^\infty$ (truncated). It is the same as a #sponge-function but the **input is prefixed with the secret key** K.



$$Z \leftarrow \text{sponge}_K(M) = \text{sponge}(K||M)$$

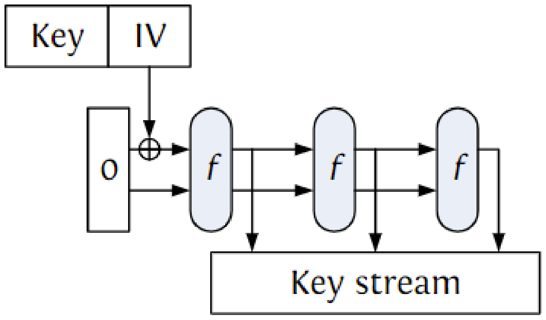
Keyed sponge for authentication



$$\text{sponge}_K(M) \text{ gives a MAC on } M \text{ under key } K$$

It is an *extension of the sponge construction* where **a secret key is introduced**. The k**ey is used to initialize the internal state of the sponge**. The construction involves *absorbing the key and input data alternately*, followed by a *squeezing phase to generate an authentication tag*. This **approach provides a secure way to authenticate messages**, and it's commonly used in authenticated encryption schemes, such as HMAC (Hash-based Message Authentication Code), where the key ensures both confidentiality and integrity.

Keyed sponge as a stream cipher

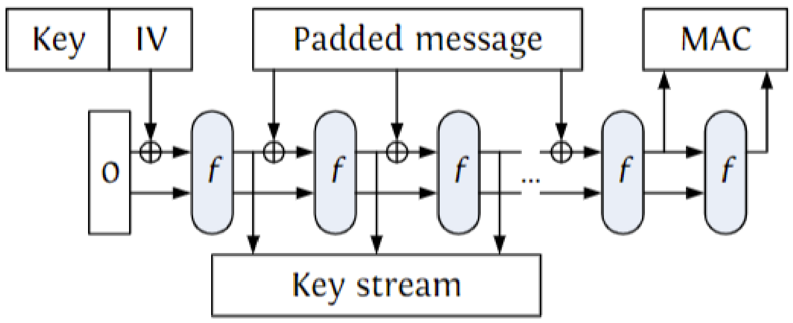


$$s_i \leftarrow \text{sponge}_K(d) \text{ produces keystream from key } K \text{ and diversifier } d$$

$$c_i = m_i + s_i \quad (\text{in } \mathbb{Z}_2)$$

In a keyed sponge as a #stream-cipher, a **secret key initializes the sponge's internal state**. The *construction alternates between absorbing the key and plaintext*, and *squeezing keystream output*. This **keystream, XORed with the plaintext, produces ciphertext**. The **key ensures confidentiality** and the **sponge's properties provide a secure and efficient stream cipher**, used in encryption and communication protocols.

Authenticated encryption: spongeWrap



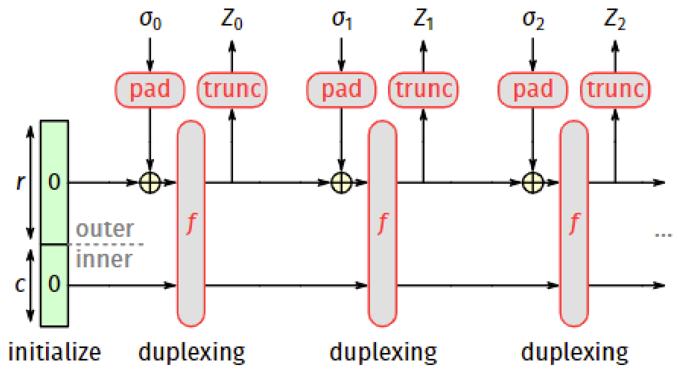
Variant of the sponge construction:

- Keystream: $s_i \leftarrow \text{sponge}_K(d \| \text{previous plaintext blocks})$
- MAC: $\text{sponge}_K(d \| \text{plaintext})$

A #spongeWrap is not known by ChatGPT and I have not listened to this part in the class RIP. But I think its a mix between the use of sponge function as a stream cipher and for authentication.

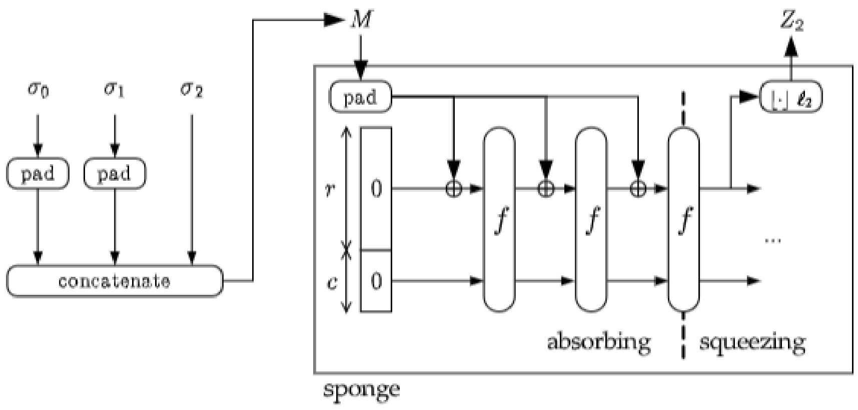
2.4.4 Duplex

The #duplex construction **extends the sponge model** to ***provide both confidentiality and authenticity***. It **allows for simultaneous processing of input and output in a single pass**, making it suitable for real-time applications.



- Object: $D = \text{DUPLEX}[f, \text{pad}, r]$
- Requesting ℓ-bit output $Z = D.\text{duplexing}(\sigma, \ell)$
  - input σ and output Z limited in length
  - Z depends on all previous inputs

Here is an example of a generating duplex with a sponge. The input is written in the pads σ_i, then concatenated and given as input for the #sponge-function .

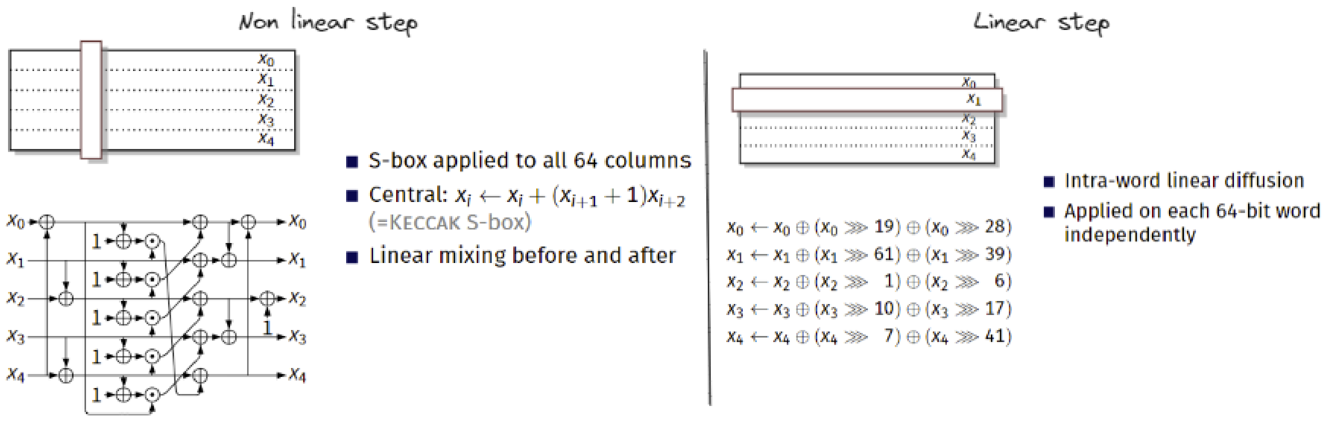


$$Z_2 = \text{sponge}(\text{pad}(\sigma_0)||\text{pad}(\sigma_1)||\sigma_2, \ell_2)$$

2.4.5 Ascon

#ASCON (Authenticated Sponge Construction) is a lightweight authenticated encryption algorithm designed for software and hardware implementations. It **uses a sponge construction** and **provides both encryption and authentication in a single algorithm**. It is designed to be resistant against various cryptographic attacks and is **well-suited for use in lightweight and constrained environments**.

#ASCON is a #permutation based authenticated #encryption-scheme with a **permutation width** $b = 320 = 5 \times 64 bits$.



- S-box applied to all 64 columns
- Central: $x_i \leftarrow x_i + (x_{i+1} + 1)x_{i+2}$ (=KECCAK S-box)
- Linear mixing before and after

$x_0 \leftarrow x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)$
$x_1 \leftarrow x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)$
$x_2 \leftarrow x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6)$
$x_3 \leftarrow x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)$
$x_4 \leftarrow x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)$

- Intra-word linear diffusion
- Applied on each 64-bit word independently

2.5 Pseudo-random function (PRF)

#PRF see slides. Slide90.