Computer & Network Security

Giuliano Abruzzo

January 14, 2020

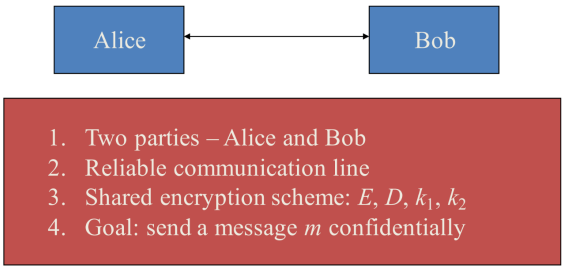
Contents

1 Introduction

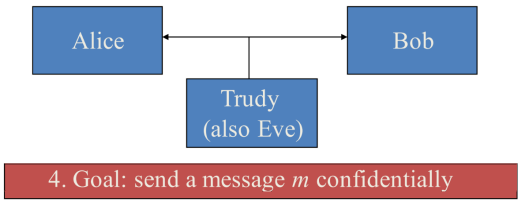
Cryptography and **Security** differ, in fact *Cryptography* deals with **secrecy** of information, *Security* deals with problems of *fraud*, like *message modifications* or *user authentication*. *Security* might use *Cryptography*, and **encryption** doesn't live alone without some form of *authentication*. We will call:

- **Encryption function**: E;
- **Decryption function**: D;
- **Encryption key**: k_1;
- **Decryption key**: k_2;
- For every *message* m: $D_{k_2}(E_{k_1}(m)) = m$;
- **Secret Key** (*symmetric*): $k_1 = k_2$;
- **Public Key** (*asymmetric*): $k_1 \neq k_2$;

A **Threat** is a *menace*, a source of danger, instead an **Exploit** is a *software* or a *chuck of data*, or a *sequence of commands* that take advantage of a *vulnerability* to cause *unintended* or *unanticipated behavior* to occur on computer. The **communication model** is:



Instead a **Threat** (*attack*) **model** is:



The **Adversary** can be **Passive**, that means that will reads the exchanged *messages* (without changes), or **Active**, that means that can modify the *message* between *Alice and Bob*, or can send *fake messages* claiming that they have been sent by someone else (*Alice or Bob*). The *passive adversary* is also called **Packet Sniffing**, instead an *active adversary* is also called **IP Spoofing**, in which T is able to forge *messages* that look like *messages* sent by A or B by modification of the

IP header. A *security threat* is **Denial of Service** or **DoS**, in which *attackers* send a lot of *packets* to the *attacked host*, a **DDoS** is a *distributed attack* through infection of *unaware computers*, often with the use of *SYN packets*.

If the *keys* are unknown then:

- It's **hard** to obtain even partial information on the *message*;
- It's **hard** to find the *key* even if we know *clear text*;

For *hard* we intend **Computationally hard**, that means that it takes a long time even with the most *powerful computers*. It's important to note that the *goal* is to:

- No *adversary* can determine *message* M (*not enough*);
- No *adversary* can determine some *information* about *message* M (*not enough*);
- No *adversary* can determine any *meaningful information* about *message* M (*good*);

The *adversarial* attempts to discover information about the *message* M, and knows the *algorithms* E, D, the *message space*, and has at least *partial information* about $E_{k_1}(M)$, but doesn't know about k_1, k_2. The **Plaintext** is the *message* prior to *encryption*, instead the **Ciphertext** is the message after *encryption*.

The **Perfect Cipher** is a *cipher* in which given a *Plaintext space* $= \{0,1\}^n$, where D is known, given a *ciphertext* C the probability that exists k_2 such that $D_{k_2} = P$ for any *plaintext* P is equal to the apriori probability that P is the *plaintext*. In other worlds. the *ciphertext* doesn't reveal any information about the *plaintext*:

$$Pr(P|C) = Pr(P)$$

And for the *conditional probability* we have that in a *perfect cipher*:

$$Pr(C) = Pr(C|P)$$

An example of a *perfect cipher* is the **One Time Pad** in which we have:

- Plaintext space and Key space: $\{0,1\}^n$;
- Symmetric scheme and k is chosen random;
- $E_k(P) : C = P \bigoplus K$;
- $D_k(P) : P = C \bigoplus K = P \bigoplus K \bigoplus K$;

Where $\bigoplus$ is an *exclusive OR*, bit by bit. The problem is the size of *key space* since for the **Theroem of Shannon**: a *cipher* cannot be *perfect* if the size of its *key space* is less than the size of its *message size*.

The **proof** is done by *contradiction*:

- We assume that the number of the keys l is less than the number of messages n so $l < n$ and we consider ciphertext $C_0 : Pr(C_0) > 0$;
- For some key k, consider $P = D_k(C_0)$ So there exists at most l keys such messages, one for each key;
- Choose message P_0 such that it is not of the form $D_k(C_0)$ since there exists $n-l$ such messages;
- Hence $Pr(C_0|P_0) = 0$ but since in a perfect cipher $Pr(C_0|P_0) = Pr(C_0) > 0$ we have a contradiction;

We have several different **attack models** like:

- **Eavesdropping**: in which the *attacker* secretly listening private conversation of others;
- **Known Plaintext**: *attacker* has samples of both *plaintext* and its *encrypted version* so will use them to reveal information like *secret keys*;
- **Chosen Plaintext**: *attacker* has the capability to choose arbitrary *plaintexts* to be *encrypted* and obtain the corresponding *ciphertexts*. The goal is to game information which will reduce the security of the *encryption scheme*.
- **Adaptive Chosen Plaintext**: the *crypt-analyst* makes a series of interactive *queries*, choosing *subsequent plaintexts* based on the information from the previous *encryptions*;
- **Chosen Ciphertext**: the *crypt-analyst* gathers information, at least in part, by choosing a *ciphertext* and obtaining its *decryption* under an unknown key;
- **Physical Access**;
- **Physical modification of messages**;

2 Secret Key: Stream Ciphers & Block Ciphers

The idea is that *Alice and Bob* share:

- a **crypto protocol** E;
- a **secret key** k;
- they *communicate* using E with *key* k;
- *adversary* knows E and some *messages* but ignores k;

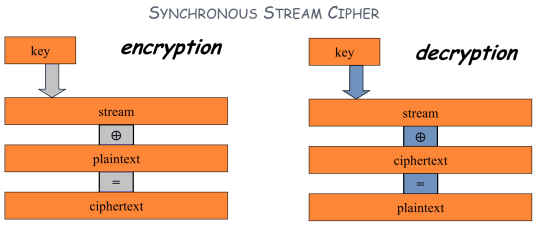
There are two different approaches **Stream Cipher** and **Block Cipher**:

2.1 Stream Cipher

The idea is try to simulate the *one-time-pad*, so we define a *secret key* called **Seed**, and we use the *seed* to generate a *byte stream* called **Keystream**:

- The i^{th} byte is function of:
  - Only key, **synchronous stream cipher**;
  - Both key and first $i-1$ bytes of the *ciphertext*, **asynchronous stream cipher**;

In a **synchronous stream cipher** the output of the *encryption function* is generated independently from the *plaintext* and the *ciphertext*. So we have for **synchronous stream cipher**:



2.1.1 A5

A5 is an *encryption algorithm* (1987) used to provide *over-the-air communication privacy* in the *GSM cellular telephone standard*. Initially it was kept secret, but the general design was leaked in 1994 and the *algorithms* were entirely *reverse engineered* in 1999.

2.1.2 RC-4

RC or *Ron's Code* was an *algorithm* easy to program, fast and very popular considered *safe* between 1987 and 1994, in which we have:

- *Variable key length* (*byte*);
- *Synchronous stream cipher*;
- Starting from the *key* it generates a *random permutation*;
- Eventually the *sequence* will repeat but for long period 2^100 so its simulate *one-time-page*;
- It's very fast in fact 1 byte of *output* requires 8-16 *instructions*;
- The goal is to generate *random permutation* of the first *256 natural numbers*;

Algorithm RC-4 Init

$j = 0$;
$S_0 = 0,\ S_1 = 0,\ \ldots, S_{255} = 255$;
assume a key of 256 bytes: $k_0,\ \ldots, k_{255}$ (if the *key* is shorter *repeat*);
for $i = 0$ **to** 255 **do:**
    $j = (j + S_i + k_i)\ mod\ 256$;
    exchange S_i and S_j;

So in this way we obtain a **permutation** of $0, 1, ..., 255$, an the *resulting permutation* is a function of the *key*.

Algorithm RC-4 Key-Stream Generation

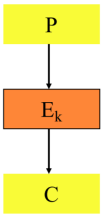
Input: *permutation* S of $0, 1, ..., 255$;
while (*true*):
$i = (i + 1)\ mod\ 256$;
$j = (j + S_i)\ mod\ 256$;
exchange S_i and S_j;
$t = (S_i + S_k)\ mod\ 256$;
$k = S_t$;

So at every *iteration* we compute the **XOR** between k and next *byte* of *plaintext* (or *ciphertext*). This *algorithm* is used in order to generate the *key* used for *encrypt* the next *byte* of the *plaintext*.

2.2 Block Ciphers

In the **Block Ciphers** instead we have:

- A **block** P of text of h bits (where h is fixed);
- A **key** k of fixed number of *bits*;
- A **cryptographic protocol** E_k produces a *block* C of h bits, function of P and k;



There are several examples of **Block Ciphers** like:

- **DES**: (1976) with 64 *bit block* and 56 *bit key*;
- **RC-2**: (1987) designed by IBM, with 64 *bit block* and variable *key size*, vulnerable by an attack of 2^{34} chosen *plaintexts*;
- **IDEA**: (1991) with 64 *bit block* and 128 *bit key*, strong, only weaker variants have been broken;
- **Blowfish**: (1993) with 64 *bit block* and variable *key size* from 32 up to 448 bits, still strong;
- **RC-5**: (1994) with variable *block size* (32, 64, 128 bits) and variable *key size* (0 to 2040 bits) and *rounds* (0 to 255).
- **AES**: (2001) with 128 *bit block* and 128-256 *bit key*, very strong;

2.2.1 AES

AES or **Advanced Encryption Standard** is an *algorithm* based on *symmetric block cipher* in which we have a *block size* of 128 bits and a *key lengths* of 128, 192 or 256 bit that use **finite fields algebra**, and now have fun with *settordici* mathematical formulas and almost useless concepts:

- A set G with *binary operation* $+$ (*addition*) is called a **commutative group** if:
  - $\forall a, b \in G, a + b \in G$;
  - $\forall a, b, c \in G, (a + b) + c = a + (b + c)$;
  - $\forall a, b \in G, a + b = b + a$;
  - $\exists 0 \in G, \forall a \in G, a + 0 = a$;
  - $\forall a \in G, \exists - a \in G, a + (-a) = 0$;
- Let $(G, +)$ be a group, then $(H, +)$ is a **sub-group** of $(G, +)$ if it is a group and $H \subseteq G$;
  - The *Theorem of Lagrange* says that: if G is finite and $(H, +)$ is a *subgroup* of $(G, +)$ then $|H|$ divides $|G|$;
- Two natural a and b are said to be **congruent modulo** n (with n a positive integer): $a \equiv b (mod\ n)$ if:
  - If $|a - b|$ is multiple of n or the integer division of a and n and of b and n yield the *same remainder*;
  - The *congruence relation* is *reflexive*, *symmetric* and *transitive*, hence it is an *equivalence relation*;
  - The *quotient set* Z_n is the set of n classes of *equivalence congruent* to $0, 1, ...n - 1$: $-1 \equiv n - 1 (mod\ n), -2 \equiv n - 2 (mod\ n)$, etc..;
  - The properties of **congruence** are:
    - **Invariance over addition**:
      $a \equiv b \cdot (mod\ n) \quad \Leftrightarrow \quad (a + c) \equiv (b + c) \cdot (mod\ n) \quad \forall\ a, b, c \in \mathbb{N}, \forall\ n \in \mathbb{N}_0$
    - **Invariance over multiplication**:
      $a \equiv b \cdot (mod\ n) \quad \Leftrightarrow \quad (a \times c) \equiv (b \times c) \cdot (mod\ n) \quad \forall\ a, b, c \in \mathbb{N}, \forall\ n \in \mathbb{N}_0$
    - **Invariance over exponentiation**:
      $a \equiv b \cdot (mod\ n) \quad \Leftrightarrow \quad a^k \equiv b^k \cdot (mod\ n) \quad \forall\ a, b, k \in \mathbb{N}, \forall\ n \in \mathbb{N}_0$
- Let's a^n denote $a + a + \ ... \ + a$ for n times (*Why a^n and not $\sum_n a$? Only D'Amore knows, we spent two days wondering why the fuck he used this notation*), we say that a is **of order** n if $a^n = 0$:
  - For any $m < n, a^m \neq 0$ then all elements of finite groups have **finite order**, $a^n = 1$ for *multiplicative operator* (where a^n denotes $a \cdot a \times \ ...$);
  - Z_m is the set of *natural numbers mod m* and the elements of Z_m are the classes of *equivalence* of **congruent integers**;
  - Z_m^* is the set of *natural numbers mod m* that are **coprime** to m, the **multiplicative group** of Z_m;

- $\phi(m)$ is the **Euler's totient function** and it's equal to $= |Z_m^*|$;
- The **Euler Theorem** says that for all a in $Z_m^*, a^{\phi(m)} = 1 \cdot mod\ m$ so we have: $a^{k \cdot \phi(m)+1} = a \times mod\ m,\ k \geq 0$;
  - We can also extend it to Z_m where $m = pq$ and p, q are **prime number** and we have: $a^{k \cdot \phi(m)+1} = a \times mod\ m,\ k \geq 0$;
- *Quindi in poche parole, Z_m è il set dei resti ottenuti dalla divisione tra i numeri naturali e m, Z_m^* è il set dei resti ottenuti dalla divisione tra i numeri naturali e m che sono coprimi con m (due numeri sono coprimi se non hanno nessun divisore comune apparte 1), la funzione toziente di eulero è la funzione che restituisce il numero di interi coprimi tra 1 e il numero m tipo $\phi(8) = 4$ perchè $1, 3, 5, 7$ sono coprimi)*;

- Let G be a **group** and a an element of order n, the set: $\langle a \rangle = \{1, a, \dots, a^{n-1}\}$ is a **sub-group** of G, a is called the **generator** of $\langle a \rangle$ (it's the number from which we can generate all the *elements* of the *subgroup*). If G is *generated* by a then G is called **Cyclic**, and a is the **primitive element** of G.
  - The *theorem* says that for any *prime number* p the **multiplicative group** of Z_p^* is cyclic;
- A set F with two *binary operations* $+$ *addition*, and $\times$ (or $\cdot$) *multiplication* is called a commutative **ring** with identity if:

1	$\forall\ a,b \in F,\ a+b \in F$	6	$\forall\ a,b \in F,\ a \cdot b \in F$
2	$\forall\ a,b,c \in F,\ (a+b)+c=a+(b+c)$	7	$\forall\ a,b,c \in F,\ (a \cdot b) \cdot c=a \cdot (b \cdot c)$
3	$\forall\ a,b \in F,\ a+b=b+a$	8	$\forall\ a,b \in F,\ a \cdot b=b \cdot a$
4	$\exists\ 0 \in F, \forall\ a \in F,\ a+0=a$	9	$\exists\ 1 \in F, \forall\ a \in F,\ a \cdot 1=a$
5	$\forall\ a \in F, \exists\ -a \in F,\ a+(-a)=0$	10	$\forall\ a,b,c \in F, a \cdot (b+c)=a \cdot b+a \cdot c$

- A set F with two *binary operations* $+$ *addition*, and $\times$ (or $\cdot$) *multiplication* is called a commutative **field** if:

1	$\forall\ a,b \in F,\ a+b \in F$	6	$\forall\ a,b \in F,\ a \cdot b \in F$
2	$\forall\ a,b,c \in F,\ (a+b)+c=a+(b+c)$	7	$\forall\ a,b,c \in F,\ (a \cdot b) \cdot c=a \cdot (b \cdot c)$
3	$\forall\ a,b \in F,\ a+b=b+a$	8	$\forall\ a,b \in F,\ a \cdot b=b \cdot a$
4	$\exists\ 0 \in F, \forall\ a \in F,\ a+0=a$	9	$\exists\ 1 \in F, \forall\ a \in F,\ a \cdot 1=a$
5	$\forall\ a \in F, \exists\ -a \in F,\ a+(-a)=0$	10	$\forall\ a,b,c \in F, a \cdot (b+c)=a \cdot b+a \cdot c$

11 $\forall\ a \neq 0 \in F, \exists\ a^{-1} \in F,\ a \cdot a^{-1}=1$

  - A **field** is a *commutative ring* with *identity* where each non-zero element has a *multiplicative inverse*. $(F, +)$ is a **commutative (additive) group** and $(F \setminus \{0\}, \cdot)$ is a **commutative (multiplicative) group** (with $\cdot$ distributive over $+$)

- Let $f(x) = a_0 + a_1 \cdot x + ... + a_n \cdot x^n$ be a **polynomial** of *degree* n in one *variable* x over a *field* F, the **theorem** says that $f(x) = 0$ has almost n solution in F. It's important to note that this *theorem* doesn't hold over *rings with identity*.
  - Let $f(x) = a_0 + a_1 \cdot x + ... + a_n \cdot x^n$ and $g(x) = b_0 + b_1 \cdot x + ... + b_m \cdot x^m$ two *polynomials* over F such that $m \leq n$. There is a *unique polynomial* $r(x)$ of degree $< m$ such that: $f(x) = h(x) \cdot g(x) + r(x)$ where $r(x)$ is called the **remainder** of $f(x)$ **modulo** $g(x)$;
- A field $(F, +, \cdot)$ is called a **finite field** if the set F is *finite*.
  - The *theorem* says that for every **prime power** p^k with $k = 1, 2, ...$ there is a **unique finite field** containing p^k elements. These field are called **Galois Fields** and are denote with $GF(p^k)$. There aren't *finite filed* with other *cardinalities*.
  - *Polynomial equations* and *factorizations* in *finite fields* can be different than over the *rationals*;
  - A *polynomial* is **irreducible** in $GF(p)$ if doesn't *factor* over $GF(p)$, otherwise is *reducible*.
- For a *theorem*: let $f(x)$ be an **irreducible polynomial** of *degree* k over Z_p, the *finite field* $GF(p^k)$ can be realized as the set of **degree** $k-1$ **polynomials** over Z_p, with *addition* and *multiplication* done *modulo* $f(x)$;
  - By the *theorem* the **finite filed** $GF(2^5)$ can be realized as the set of *degree* 4 *polynomials* over Z_2 with *addition* and *multiplication* done *module* the *irreducible polynomial* $f(x) = x^5 + x^4 + x^3 + x + 1$;
  - The *coefficientes* of *polynomials* over Z_2' are 0 or 1, so a *degree* k can be written down by $k+1$ *bits* so for example with $k = 4$:
    - $x^3 + x + 1 \rightarrow (0, 1, 0, 1, 1)$;
    - $x^4 + x^3 + x + 1 \rightarrow (1, 1, 0, 1, 1)$;
  - The *addiction* can be done through **XOR** since $1 + 1 = 0$:

$x^4 + x^3 + x$	$(1,1,0,1,0)$	+
$x^3 + x + 1$	$(0,1,0,1,1)$	=
$x^4 + 1$	$(1,0,0,0,1)$	

  - The **multiplication** is done by a *Polynomial multiplication*, and then remainder *modulo* the defining *polynomial*. For small size *finite field*, the *lookup table* is the most efficient method for implementing *multiplication*.

Now finally the **AES**, that is a *symmetric block cipher* with *key lengths* of 128, 192 or 256 bit, and it's *resistance* to all known *attacks*. It's *simple* and it's *fast*, so it's very good for devices with limited *computing power*. When we have *Input* and *Output block length* of 128 bits the *State* of 128 bits we can arrange it as a **4-by-4 matrix of bytes**. 128 since we have 16 elements in the *matrix* and each element is in byte that is formed by 8 bits so $(16 \cdot 8 = 128)$.

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

When we have a *key length* of 128,196,256 bits the *Cipher Key Layout* that is arranged in a 4-by-$fracn32$ matrix of *bytes* (where $n = 128, 196, 256$ bit):

$K_{0,0}$	$K_{0,1}$	$K_{0,2}$	$K_{0,3}$	$K_{0,4}$	$K_{0,5}$
$K_{1,0}$	$K_{1,1}$	$K_{1,2}$	$K_{1,3}$	$K_{1,4}$	$K_{1,5}$
$K_{2,0}$	$K_{2,1}$	$K_{2,2}$	$K_{2,3}$	$K_{2,4}$	$K_{2,5}$
$K_{3,0}$	$K_{3,1}$	$K_{3,2}$	$K_{3,3}$	$K_{3,4}$	$K_{3,5}$

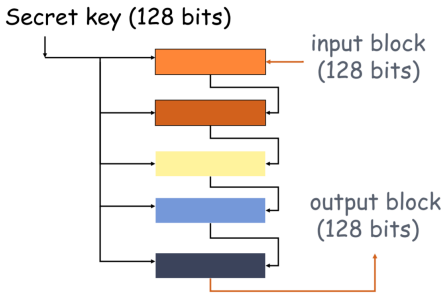
Initial layout: $K_{0,0}$, $K_{1,0}$, $K_{2,0}$, $K_{3,0}$, $K_{0,1}$,...

The *algorithm* at **high level** (*without explanations in details that we will see later*) is:

Algorithm High level AES

$AES(State, Key)$;
$KeyExpansion(Key, ExpandKey)$;
$AddRoundKey(Key, ExpandKey\,[0])$;
for $(i = 1, i < R, i++)$ **do**:
    $Round(State, ExpandKey\,[i])$;
$FinalRound(State, ExpandKey\,[R])$;

So, this *code* is repeated for each *block* of the *plain text* (each *block* is of 128 bits), so we start with a *key* of 128 bit, but before we start the *rounds* (that are 10) since we don't want to use the same *key* of 128 bits for all the different *blocks*, we need to expand the *key* in such a way for each *block* we have a different *key* of 128 bits. The **encryption** is done by following this scheme:



We can see that every *state* is done by the **encryption** of the *precedent round* with the *key extended* of the *current block*. 128 bits AES uses 10 *rounds*, in which the *secret key* of 128 bits is expanded to 10 *round keys* of 128 bits each. Each *round* changes the **state**, then we **XOR** the *round key*, if we have *longer keys* we add one *round* for every extra 32 bits. Now we will see in details what happen in each single *round*, we start from the assumption that the *plain text* that we are going to *encrypt* is divided in *state* of 128 bit and each *state* can be divided in a matrix 4×4. We will **transform the state** by applying:

1. **Substitution**;
2. **Shift rows**;
3. **Mix columns**;
4. **XOR round key**;

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$
$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	$A_{1,3}$
$A_{2,0}$	$A_{2,1}$	$A_{2,2}$	$A_{2,3}$
$A_{3,0}$	$A_{3,1}$	$A_{3,2}$	$A_{3,3}$

The **Substitution** operates on every *byte* separately: $A_{i,j} \leftarrow A_{i,j}^{-1}$, so we apply a *transformation* in place in every single *block*, the operation is a *multiplicative inverse* which is highly *non-linear*. If $A_{i,j} = 0$ then we don't change $A_{i,j}$. It's important to note that the **substitution is invertible**.

We will **Shift** each element of each *row* to the **right**, 0 for the *first row*, 1 for the *second row*, 2 for the *third row*, 3 for the *fourth row*. This shift is invertible.

$A_{0,0}$	$A_{0,1}$	$A_{0,2}$	$A_{0,3}$	no shift
$A_{1,3}$	$A_{1,0}$	$A_{1,1}$	$A_{1,2}$	shift 1 position
$A_{2,2}$	$A_{2,3}$	$A_{2,0}$	$A_{2,1}$	shift 2 positions
$A_{3,1}$	$A_{3,2}$	$A_{3,3}$	$A_{3,0}$	shift 3 positions

In the **Mixing Columns** operation every state *column* is considered as a *Polynomial* over $GF(2^8)$, we will multiply with an *invertible polynomial* $03x^3 + 01x^2 + 01x + 02 (mod\ x^4 + 1)$ the *Inv* instead is $0Bx^3 + 0Dx^2 + 09x + 0E$.

The **Key Expansion** instead will generate a different *key* per *round*, and need a 4×4 matrix of values per *round*. It's based upon a *non-linear transformation* of the *original key*.

Breaking 1 or 2 *rounds* of the **AES** is easy, actually is not known how to break 5 *rounds*, and breaking the full 10 *rounds* efficiently is considered impossible.

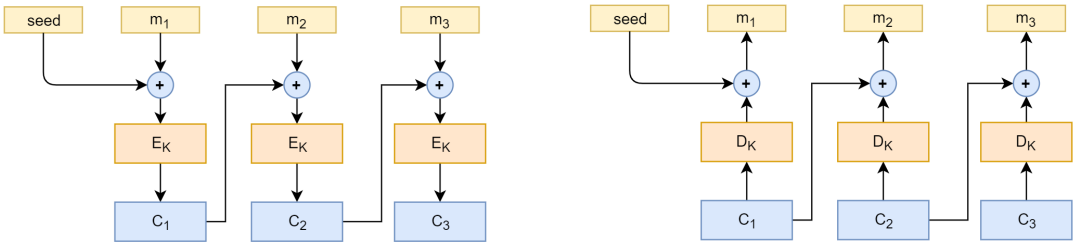
All *block ciphers* operate on *blocks* of *fixed length* cause *message* can be of any *length* and because **encrypting** the same *plaintext* under the same *key* always produces the same *output*. Several modes of *operation* have been invented which allow *block ciphers* to provide **confidentiality** for *messages* of *arbitrary length*.

2.2.2 ECB

ECB or also called **Electronic Code Book** is an stream cipher in which we *encrypt* each *plaintext block* separately, and it's very simple and efficient, and also it's possible to implement it in a *parallel way*. It doesn't conceal **plaintext patterns**, and it's possible to attack it with *active attacks*, in fact *plaintext* can be manipulated by removing, repeating or interchanging *blocks*.

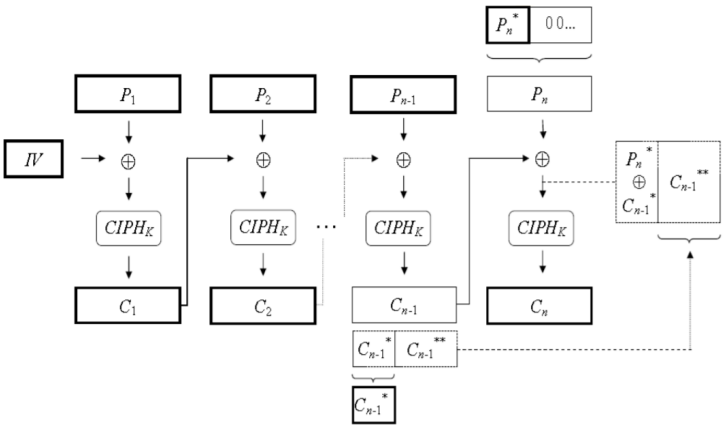
2.2.3 CBC

CBC is the **Cipher Block Chaining**, created by *IBM* in the 1976, is an *asynchronous stream cipher*, in which *errors* in one *ciphertext block* will propagate, that conceals *plaintext patterns* and there aren't *parallel implementation*. *Plaintext* cannot be easily manipulated and it's a standard in a lot of systems. In this cipher the previous *ciphertext* is **XORed** with current *plaintext* before *encrypting* current *block*. We use a *seed* to start the process, and this *seed* can be sent without *encryption*. If we set $seed = 0$, in most case is safe, but a *random seed* is better.



The *CBC* can lead to *problems*, in fact if there is a **transmission error** that changes just one bit of the C_{i-1} then block m_i changes in a *predictable way* (and this is be used by an *adversary*), but there are *unpredictable changes* is m_{i-1}, so the *solution* is to always use an **error detecting code**, that check the quality of the *transmission*. The *message* of the *CBC* must be **padded** to a multiple of the *cipher block size*, and in order to handle this issue we need to use the **Ciphertext Stealing**. A *plaintext* can be *recovered* from just two adjacent *blocks* of *ciphertext*, so as a consequence *decryption* can be **parallelized**, in fact usually a *message* in encrypted once, but *decryption* is done many times.

Ciphertext Stealing is a general method that allows for processing of *messages* that are not evenly divisible into *blocks*, so it doesn't result in an expansion of the *ciphertext*, it cost only in *complexity*. It consists of altering processing of the last two *blocks* of *plaintext*, so it re-order the *transmission* of the last two *blocks* of *ciphertext* and this method is suitable for *ECB* and *CBC*.



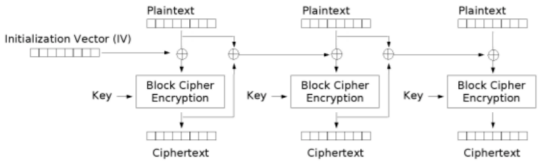
We can see that the last *plaintext block* is not divisible for 128 (*plaintext block size*, our *block* is made by n bits) so we fill the *block* with all 0, the **XOR** will be done only by the first n bits of the *previous ciphertext*. So during the *decryption* the *block* will be $\left[P_n^* \bigoplus C_{n-1}^* | C_{n-1}^{**}\right]$ so in order to get the original plaintext the result will be XORed with C_{n-1} in this way we obtain:

$$\left[P_n^* \bigoplus C_{n-1}^* | C_{n-1}^{**}\right] \bigoplus \left[C_{n-1}^* | C_{n-1}^{**}\right] = \left[P_n^* | \ 000000...\right]$$

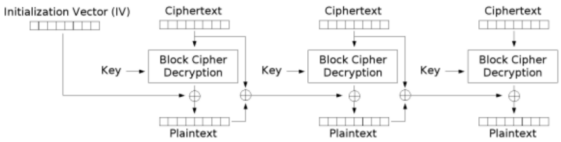
- **Encryption**:
  - If the *plaintext* length is not a multiple of the *block size*, we will pad it with enough zero until it is.
  - We *encrypt* the *plaintext* using the *Cipher Block Chaining mode*;
  - We swap the last two *ciphertext blocks (? swap? where? in the figure there isn't any swap)*;
  - We truncate the *ciphertext* to the length of the original *plaintext*;
- **Decryption**:
  - If the *ciphertext* length is not a multiple of the *block size*, (like n bits short), then we pad the it with the last n bits of the *block cipher decryption* of the last full *ciphertext block*;
  - Swap the last two *ciphertext blocks*;
  - *Decrypt* the *ciphertext* using the *Cipher Block Chaining mode*;
  - We truncate the *plaintext* to the length of the original *ciphertext*;

2.2.4 PCBC

PCBC is designed to extend or propagate a *single bit error* both in *encryption* and *decryption* (before if there was a *bit error* it will propagate to change all the *message* here only a *bit*). Here we have that the current *plaintext* (i) before *encryption* is *XORed* with: $plaintext_{i-1} \bigoplus ciphertext_{i-1}$. In the *decryption* will happen the same procedure, but the *XOR* will happen after the *decryption*, so since the *decryption* need the *plaintext* of the precedent *block* is not possible anymore to do it in *parallel* like in the *CBC*.



Propagating Cipher Block Chaining (PCBC) mode encryption



Propagating Cipher Block Chaining (PCBC) mode decryption

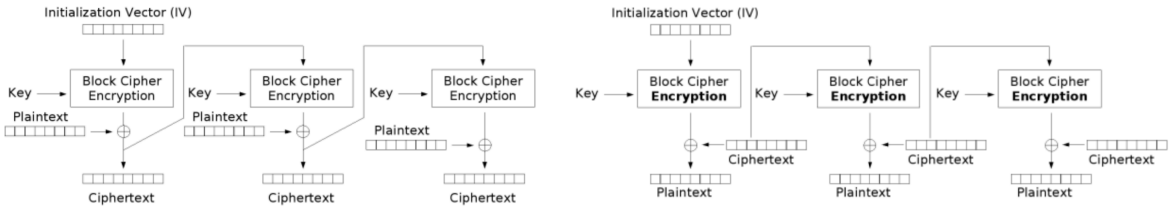
On a *message encrypted* in **PCBC mode**, if two adjacent *ciphertext blocks* are exchanged, this doesn't affect the *decryption* of *subsequent blocks* (this doesn't happen to *CBC*):

- $I_{i+2} = C_{i+1} \bigoplus (D_k(C_{i+1}) \bigoplus I_{i+1})$;

- $I_{i+1} = C_i \bigoplus (D_k(C_i) \bigoplus I_i)$;
- $I_{i+2} = C_{i+1} \bigoplus (D_k(C_{i+1}) \bigoplus C_i \bigoplus D_k(C_i) \bigoplus I_i$

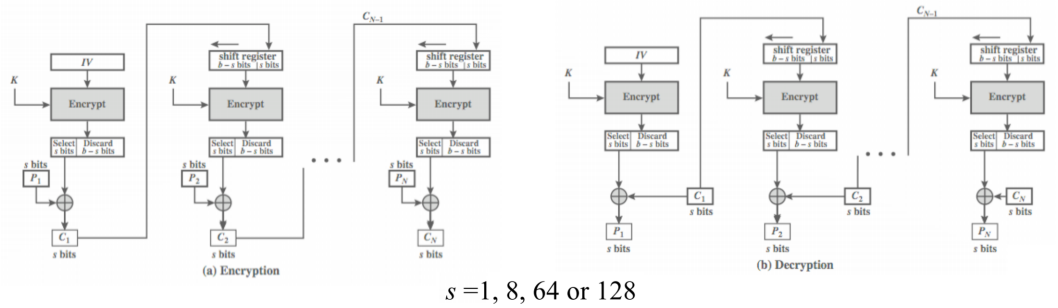
2.2.5 CFB

CFB is similar to *CBC* but makes a *block cipher* into an **asynchronous stream cipher**, so it supports some *re-synchronizing* after error if the input to the encryptor is given by through a **shift-register**. In the *encryptor* the *plaintext* will be *XORed* after applying the *encrypt* (in the *CBC* we do it before). Since the *XOR* is done after *encryption* we don't need anymore to pad last *block* with 0 when the *plaintext size* is not a multiple of the *block size*:



The *decryption* can be made in *parallel* since in order to obtain the *plaintext* I only need the *current block* and the precedent *ciphertext*. If there is an error in the output of the *encryption algorithm* this error will not be propagated in all the other *blocks*, since it will only infect the current *XOR* and the *decryption* of the next *block*. So **CFB** shares two advantages over *CBC*, the *block cipher* is only ever used in the *encryption* direction and the *message* doesn't need to be *padded* to a multiple of the *cipher block size*.

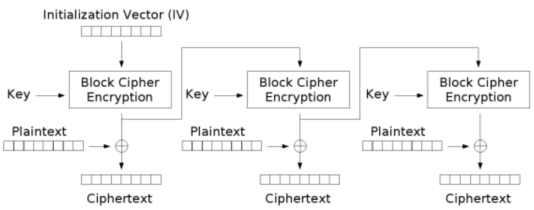
When we want to use a **CFB with shift register**, we start by initializing a *shift register* the size of the *block size* with the *initialization vector*, this will be *encrypted* with the *block cipher* and the highest s bits of the result are *XORed* with s bits of the *plaintext* to produce s bits of *ciphertext*, that then we be *shifted* in the *register* and this *process* will be repeated for the next s bits of *plaintext*. The *decryption* is similar, we start with the *initialization vector* we *encrypt* and *XOR* the high bits of the result with s bits of the *ciphertext* to produce s bits of *plaintext*, then we *shift* the s bits of the *ciphertext* into the *shift register* and *encrypt* again:



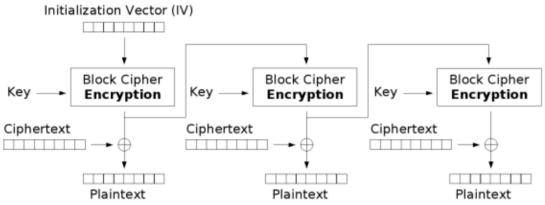
s =1, 8, 64 or 128

2.2.6 OFB

OFB or *Output FeedBack* makes a *block cipher* into a **synchronous stream cipher**, it generates *keystream blocks* which are then *XORed* with the *plaintext blocks* in order to get the *ciphertext*, similar to *CFB* the *plaintext* is *XORed* after the *encyrption*. Flipping a *bit* in the *ciphertext* produces a *flipped bit* in the *plaintext* at the same location, and this property allows many error correcting codes to function normally even when applied before *encryption*. Contrarily to *CFB* in output in the second *block* I will not give the *XORed* with the *plaintext* but I will pass directly the *encrypted output* of the algorithm, in this way if in the *CFB* two *blocks* of *plaintext* are influenced by an *error*, in *OFB* only the *current block* is influenced by the *error*.



Output Feedback (OFB) mode encryption



Output Feedback (OFB) mode decryption

Since we have *symmetry* of *XOR operation* **encryption** and **decryption** are exactly the same:

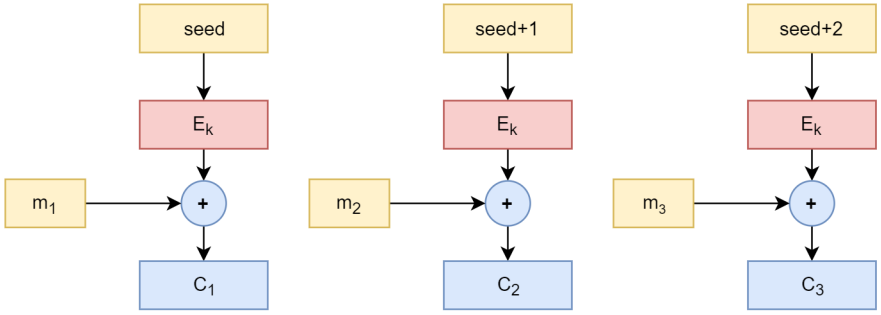
- $C_i = P_i \bigoplus O_i$;
- $P_i = C_i \bigoplus O_i$;
- $O_i = E_k(O_{i-1}$;
- $O_0 = IV$;

If E_k is public, we need to **encrypt** the initial *seed*. If E is a *cryptography function* that depends on a *secret key* then the *seed* can be in clear. The *initial seed* must be modified for every new *message*, in fact if the *adversary* knows a *pair message*, initial *seed* then he can encode every *message*. So with **OFB** we have:

- *Synchronous stream cipher*;
- *Errors* in *ciphertext* do not propagate;
- *Pre-processing* is possible;
- Conceals *plaintext patterns*;
- No *parallel implementation* known;
- *Active attacks* by manipulating *plaintext* are possible;

2.2.7 CTR

CTR or *Integer Counter Mode* turns a *block cipher* into a *stream cipher*, in fact it generates the next *keystream block* by *encrypting* successive values of a *counter*. This *counter* can be any *function* which produces a *sequence* which is guaranteed not to repeat for a long time. It has similar characteristics to *OFB* but it also allows a *random access property* during *decryption*, so it's well suited to operation on a *multi-processor machine* where *blocks* can be *encrypted* in *parallel*. The generation of each *ciphered block* is independent from any other *blocks* since we use a different *seed* for each *block*.



Of course we have problems if we repeat the *seed* like *OFB*. When we use *CTR* we can *decrypt* the *message* starting from *block* i for any i, since we don't need to *decrypt* from the first *block*.

2.2.8 Initialization Vector

Most *modes* (except *ECB*) require an **Initialization vector**, or **IV**, that is a sort of *dummy block* used to kick off the *process* for the first real *block*. It doesn't need to be secret in most case but it's important that it is never reused with the same *key*, in fact, in *CBC* and *CFB*, reusing *IV* leaks some information about the first *block* of *plaintext*. In *CBC* mode the *IV* must be unpredictable in *encryption time*. For *OFB* and *CTR* reusing an *IV* completely destroys *security*.

2.2.9 Strengthening a cipher

There are two ways in order to **strengthening a cipher**:

- **Key Whitening**:
  - Consist of steps that combine the *data* with portions of the *key* (usually by using *XOR*) before the *first round* and after the *last round* of *encryption*;
- **Iterated Ciphers** (*Triple DES, 3-DES*):
  - In which the *plaintext* undergoes *encryption* repeatedly by *underlying cipher*, in which ideally we use a different *key* so we have for *triple cipher*:
    - $C = E_{k_1}(E_{k_2}(E_{k_1}(P)))$ called EEE mode;
    - $C = E_{k_1}(D_{k_2}(E_{k_1}(P)))$ called EDE mode;

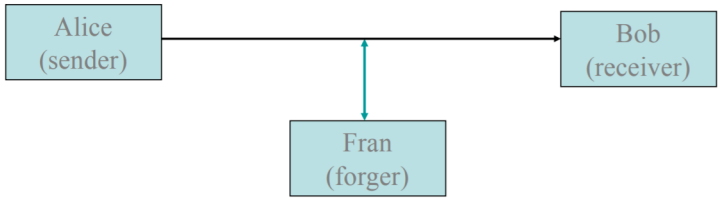
– So we use *three algorithms* of *encryption* or *two algorithm* of *encryption* and one of *decryption*. Sometimes only *two keys* are used in *3-DES*, and identical key must be at beginning and end.

Since the *goal* of the **adversary** it to find the *secret* key, the **double DES** cannot be used, in fact if we have a *double cipher* with two different *keys* the *adversary* can use the **meet-in-the-middle attack**. If fact if *plaintext/ciphertext* pairs are known, we have 2^n *encryption* and 2^n *decryption* so 2 keys of n bits. So it's possible to try all possible 2^n *encryptions* of the *plaintext* and all possible 2^n *decryptions* of the *ciphertext*. In other words, the presence of two different *keys* is represented as a *function* $h(x) = g(f(x))$, and the *middle attack* try to *map* the *co-domain* given the *domain* of the *function* f with the *counter-image* of the *co-domain* of the g function, so if initially you think to have a set of 2^{2n} keys in reality the attacker will make 2^n attempts, cause assuming that the *adversary* knows some *ciphertexts* C and some *plaintexts* P, the attacker can **in parallel** *encrypt* P and *decrypt* C trying to find some correspondences and if he find them he found the *key*.

It's even possible to use the *triple encrypting* with the **CBC**, and this can be done in two ways, by **external CBC**, in which we use a *triple encoding*, or **internal CBC**, in which we have an *encryption* after a *decryption* (with a different key) and another *encryption*.

3 Data Integrity & Authentication

The *goal* is to ensure the **integrity** of *messages* even in presence of an *active adversary* who sends own *messages*. It's important to note that *authentication* is orthogonal to *secrecy* (so they are independent between each other), so the *secrecy* of a *messages* doesn't not imply that the *sender* of the *message* is actually the one with who you think you are communicating.



3.1 MAC

The **authentication algorithm** is called A, the **verification algorithm** is called $V(accept/reject)$, the **authentication key** is k, and the **message space** is usually a *binary strings*. Every *message* between Alice and Bob is a *pair*: $(m, A_k(m))$, where $A_k(m)$ is called the **authentication tag** of m. The *authentication algorithm* is called **MAC**, *Message Authentication Code*, in fact $A_k(m)$ is frequently denoted as $MAC_k(m)$ and the *verification* is done by executing *authentication* on m and comparing it with $MAC_k(m)$. The *security requirement* is that *adversary* can't construct a new **legal pair** $(m, MAC_k(m))$ even after seeing a $(m_i, MAC_k(m_i))$. The *output* should be as short as possible and the *MAC function* is not 1-to-1.

The **adversary** knows the *MAC algorithm*, knows *plaintext pairs* $(m, MAC_k(m))$, and knows *chosen plaintext* (unrealistic) so choose m get $MAC_k(m)$ and the **goal** is given n *legal pairs* find a **new**

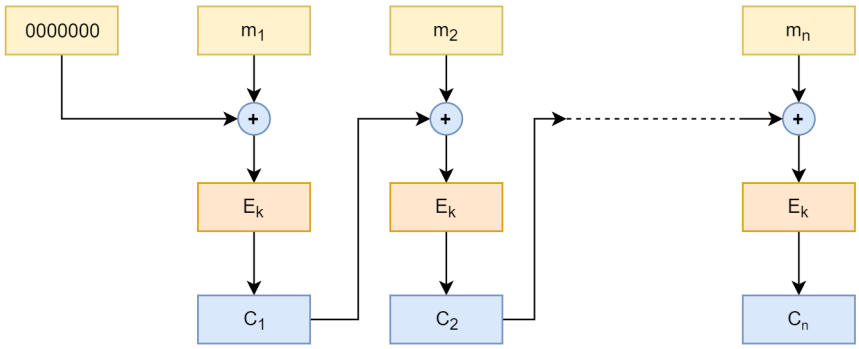
legal pair $(m, MAC_k(m)$ **efficiently** and with **non negligible probability** (so, *non sparato a cazzo*).

In practice **MAC** used are based on two types:

- **MAC based on CBC encryption**, so uses a block cipher (slow);
- **MAC based on cryptographic hash functions**, no restriction on export (fast);

3.2 MAC based on CBC

In CBC the previous ciphertext is XORed with current plaintext before encrypting the current block. With CBC with MAC's we start with the all zero seed, and given a message m consisting of n blocks $M_1, M_2, ..., M_n$ we apply CBC using a secret key k.



So this produces n *ciphertexts block* $C_1, C_2, ..., C_n$ and we decide that $MAC_k(m) = C_n$, so we take only the last *ciphertext*. This is a very *secure* way, but it's **very slow** since the *encryption* cannot be done in *parallel*. A **pseudo random function**, it's a function that looks random, and a good *encoding scheme* transforms the *message* in an apparently *random string*. If A_k is a *pseudo random function*, the **fixed length CBC MAC** is resilient to forgery. *CBC MAC* is secure for *fixed-length messages* but insecure for *variable-length messages*, in fact, if an *attacker* knows correct *message-tag pairs* (m, t) and (m', t') then can generate a *longer messages* m'' whose CBC-MAC will also be t':

- *XOR first block* of m' with t and then *concatenate* m with the modified m';
- $m'' = m \mid m'_1 \bigoplus t \mid m'_2 \mid ... \mid m'_x$;

So in this way the *tuple* sent by the *attacker* will be accepted by the *receiver*. A solution in order to avoid this attack is to produce *ciphertext blocks* with a *different key*, not the same of the *key* of the *authentication tag*.

3.3 MAC based on hash functions

The **hash functions** are *functions* in which we map *large domains* to *smaller ranges*, used extensively for *searching* (*hashing tables*), we have the problem of *collisions* that are resolved with

chaining or *double hashing*, etc. The goal is to compute MAC of a *message* by using an *hash function* h for a *message* m with a *secret key* k, so MAC must be a *function* of the *key* and the *message*. For example $MAC_k(m) = h(k \parallel m)$.

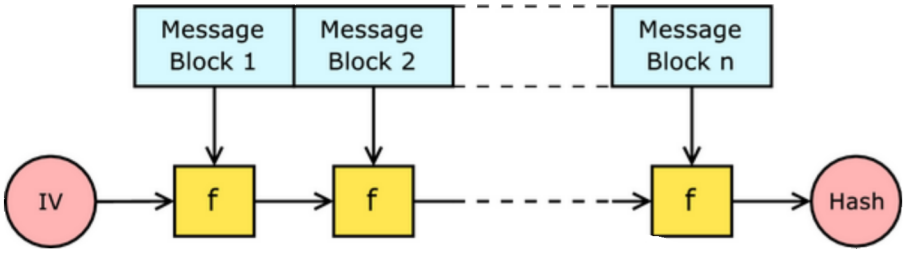
- An *hash function* $h : D \to R$ is called **weakly collision resistant** for $x \in D$ if it's hard to find $x \neq x^{'}$ such that $h(x^{'}) = h(x)$;
- An *hash function* $h : D \to R$ is called **strongly collision resistant** if it's hard to find $x, x^{'}$ such that $x \neq x^{'}$ but $h(x^{'}) = h(x)$;

The difference between the two definitions is that in the *weakly* the *element* is given, instead in the *strongly* we need to find two *elements*. It's important to note that $Strong \Rightarrow Weak$ and the proof is:

- If I have an **non-weak** I can use an *algorithm* that takes x as *input* and returns $x^{'}$ that *collide* with x. I can also use an *algorithm* that finds both $(x, x^{'})$ that *collide*, so it's *not strong*;

The **birthday paradox** says: if 23 people are chosen at random the *probability* that two of them have the same birthday is > 0.5. In other worlds, if $h : D \to R$ if we choose randomly $1.1774|R|^{1/2}$ *elements* of D the probability that two of them *collide* is > 0.5. This means that if we have 2^n possible *message tags*, we have to try only $2^{n/2}$ *messages* to have a **collision probability** > 0.5. The **Birthday Attack** uses this paradox, in fact given a *function* f find two different *input* x_1, x_2 such that $f(x_1) = f(x_2)$ with a 64-bit hash, would take at least 5.38×10^9 attempts to generate a *collision* by using *brute force*. This limit is called **birthday bound**.

Cryptographic hash function are *hash functions* that are *strongly collision resistant*, they don't use a *secret key*, usually they are defined by a *compression function* that maps n bits in m bits with $m < n$. We talk about **Merkle-Damgard construction**:



A mode of *operation* that returns a **digest** from a *document*. f is an *hashing function* that has a *domain* of n bits and a *codomain* of m bits. This means that the *message* has to be divided in *blocks*. In the figure we can see that the *block size* is $n - m$. To avoid *birthday attacks* we simply have to define a large **codomain size**, (more than 160 bits) in such a way *brute-force* is not feasible.

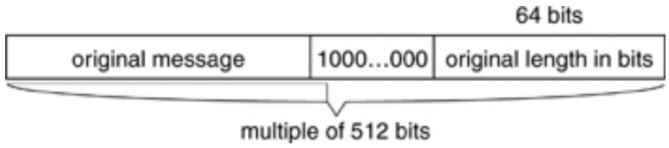
The **Keyed Hashing Functions** have the goals to combine *messages*, *keys* and *hash functions* in order to produce *MAC*. These provide both *data integrity* and *authentication*. We have to mix the *message* and the *key*, in a concatenation, and then use *hash-based algorithms*. We have different ways:

- $MAC_k(m) = h(k \parallel m)$, **insecure**, cause it suffers the *length-extension attack*, in which an *attacker* can add extra bits to the *message* and compute correct *MAC*;
- $MAC_k(m) = h(m \parallel k)$, **insecure**, cause if *attacker* finds a *collision* in the *hash function* (*birthday attack*) between two *messages*, then he will know two *messages* with the same *MAC* for all *keys*;
- $MAC_k(m) = h(k \parallel m \parallel k)$, **secure**;
- $MAC_k(m) =$ first bits of $h(k \parallel m)$ or $h(m \parallel k)$, **secure**;

3.3.1 SHA-1

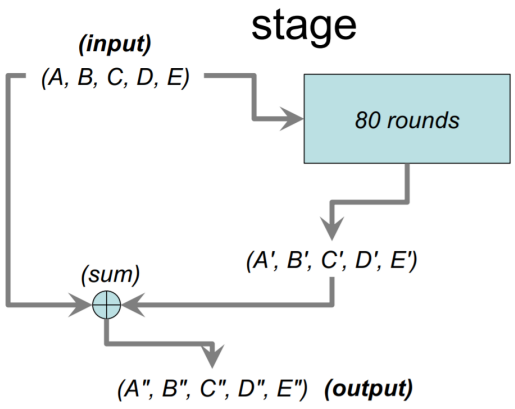
As any *hash function*, **SHA-1** produces a *digest* of fixed length (160 bit) starting from a *message* of variable length, that will be *padded*. There are 4 phases:

- **Padding**:
  - In this phase, *padding bytes* are added to the original *message* until it is congruent in bit to 448 *mod* 512;



- **Adding length**:
  - To the *final message* (*original message* plus *padding*) we add an *unsigned integer* of 64 bits that denoting the *full length* of the *message*, so we obtain a *message* that is multiple than 512 bit;
- **MD buffer initialization**:
  - A 160 *bit buffer* is subdivided in 5 *registers* of 32 bits each one that are:
    - $A = 67452301$;
    - $B = EFCDAB89$;
    - $C = 98BADCFE$;
    - $D = 10325476$;
    - $E = C3D2E1F0$;

- **Block Elaboration**:
  - Now that we have the *5 registers* we can start computing *hashing*. The *complete message* is subdivided into several *blocks* of 512 bits. The core of the **SHA-1** *algorithm* is the **compression function** which is composed by 4 *cycles* of 20 *steps* each one. Starting from the *first block* of 512 bits we give as *input* to the *compression function* the *block*, a *constant* k, and the values of the *five registers*. At the end of the *compression function* a new state of the *five registers* is observed, and we go on with the next *block*. When the *last block* of the *message* is compressed the *final values* of the *five register* will be summed to the initial values of the *five register* and the resulting values creates the **digest** that will be composed as $A|B|C|D|E$.



3.3.2 HMAC

It's a **Keyed Hashing MAC** which uses a pattern similar tot last one we listed. It's a *mode of operation* that we can use inside an *hash-algorithm*, so it's not an *hash-algorithm*. *HMAC* receives as input a *message* m a *key* k and an *hash function* h and then it computes:

$$HMAC_k(m,h) = h(k \bigoplus opad \parallel h(k \bigoplus ipad \parallel m)$$

Where *opad* is the **outer padding**: $0x5c5c...5c$, and *ipad*, **inner padding**: $0x3636...36$ both long as a *block*.

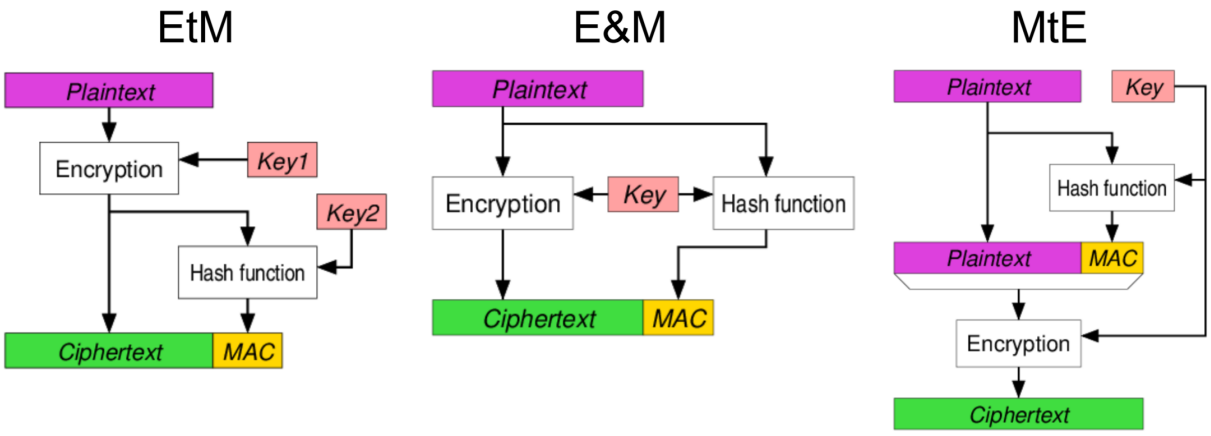
We can note that if two *messages* collide in the *inner hash* $h(k \bigoplus ipad \parallel m)$ they will collide for the whole *hashing scheme*, cause the *outer hash* adds the same staff to both *messages*. So, *HMAC* still suffers the **birthday paradox**, but the *adversary* cannot understand which are the *colliding pairs*, in fact, he is able to find two strings which *collide*, but since the *adversary* doesn't know the *key* k he cannot retrieve the *messages* of the strings.

3.3.3 AE

AE, or **Authenticated Encryption**, is a form of *encryption* which simultaneously assures the *confidentiality* and the *authenticity of data*, often is offered as single primitive in modern *API*, and it

makes *chosen-ciphertext attack* less dangerous, since the *attacker* is not able to choose a *ciphertext* and present it to the *decryption* in a proper way. There are three different approaches to AE:

- **Encrypt-then-MAC (EtM)** :
  - The most secure today, in which we use two *different keys* k_1 and k_2 where the first one is used for the *encryption*, the second is used to find the MAC starting from the *encrypted message*;
- **Encrypt-and-MAC (E&M)** :
  - *Safety* has not yet been proven, in which the *same key* is used for the *encryption* and in order to find the the MAC, starting from the *plaintext*;
- **MAC-then-Encrypt (MtE)**:
  - Proven to be *secure* in specific setting, like in $E\&M$ we use only one *key*, but here first we generate the MAC starting from the *plaintext*, and after *plaintext* and MAC are *encrypted* together with the *key*;



4 Public Key Cryptography

We have seen a *model* in which Alice and Bob share the same *secret key* $k_{A,B}$, and it must be *secretly generated* and *exchanged* before using the *communication channel*. Then in 1976, *Diffie and Hellman* came up with with a great idea: the **Public Key Infrastructure**. It's a model that uses **asymmetric encryption**, where each part has *two keys* K_E and K_D used for *encrypting* and for *decrypting*. They are *asymmetric*, this means that once we *encrypt* with a *key*, we must *decrypt* with the other *key*, and it's important to note that one of the *two keys* is *public* and the other is *private*. The implementation where *keys* are *asymmetric* it's hard and there are several ways:

- **One Way Function**:
  - Are *functions* for which given the input it's **easy** to compute the *output* (in *polynomial time*), but given the *output* is **difficult** to retrieve the *input* that generated that *output*. An example of such a *function* is multiplication: given two numbers p, q compute $N = p \times q$ it's easy but having N it's hard to retrieve p and q, the mathematical definition is:
  - A function $f : \{0,1\}^* \to \{0,1\}^*$ is **one-way** if f can be computed in *polynomial time*, but for every *randomized polynomial time algorithm* A, and for every *polynomial* $p(n)$ with sufficiently large n:

$$Pr\left[f(A(f(x))) = f(x)\right] < \frac{1}{p(n)};$$

  - So the *function* must be **hard** to invert in the *average case*, rather than *worst-case* sens, and that the *probability* to find the correct x that returned $f(x)$ trough A, it's very small;
  - *A very typical question in the exam is: SHA-2 is a one way function? The answer is yes, and we cannot use it to make encryption since it's not a one-to-one, since is not injective, if you don't know the key the ciphertext is hard to be inverted to find the plain text, you can brute force it but it's very hard;*
- **Discrete Log**:
  - Let G be a *finite cyclic group* with n elements and g a generator of G. Let y be any element in G, then it can be written as $y = g^x$ for some integer x (for example Z_n of 14 is {1,3,5,9,11,13} then the *generator* of this *group* is $g = 3$ cause from this value we get all the values of the *group*: $3 \ mod \ 14 = 3, 3^2 \ mod \ 14 = 9, etc$). Let $y = g^x$ and x the *minimal non negative integer* satisfying the equation, then x is called **discrete log** of y to base g;
  - Let $y = g^x \ mod \ p$ in Z_p^* (set of **positive integers** less than p and **coprime** to p):
    - Given x calculating $y = g^x \ mod \ p$ it's **easy and computable** in $O(log^3 P)$ steps;
    - Instead given g, y and p it's believed to be **hard** to retrieve x, the **discrete log**;
  - So retrieve the **Discrete Log** is believed to be a **One Way Function**.

4.1 Public Exchange of Keys

The **goal** is to have the possibility that two *parties*, *Alice and Bob*, who don't share any *secret information*, could perform a *protocol* in order to **derive** the same *shared key*, without exchanging

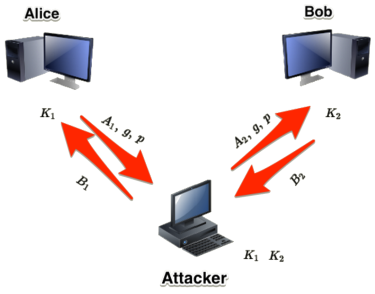
it in *clear*. *Eve*, the *attacker*, that is listening, cannot obtain the new *shared key* in *polynomial time* (unless $P = NP$ *fuori di testa zi*). This *protocol* was created by Diffie and Hellman, and the idea is to use a **session key**, that is a **shared secret key**, used for a *confidential conversation* for a certain period and then it's dropped. The protocol works as:

- The **public parameters** are: a *prime* p, a number g (possibly a *generator* of Z_p^*), and it works better if p is a **safe prime**, a *prime* number for which $p = 2q+1$ where even q is *prime*;
- Alice chooses a value a at random $\in [1, p-1]$, and sends $g^a \ mod \ p$ to Bob.
- Bob chooses a value a at random $\in [1, p-1]$, and sends $g^b \ mod \ p$ to Alice.
- Both the parties computes the **shared key** $g^{ab} mod \ p$ in fact:
  - Alice that holds a computes $(g^b \ mod \ p)^a = (g^b)^a \ mod \ p = g^{ab} \ mod \ p$ (for the properties of *mod*);
  - Bob that holds b computes $(g^a \ mod \ p)^b = (g^a)^b \ mod \ p = g^{ab} \ mod \ p$ (for the properties of *mod*);
- So at the end they have the **same key**;

This can be applied even to *multiple parties*, of course the number of *exchanged message* will increase. Despite 25 years this is still **secure**. At the end of the session the *values* a and b are *discarded*, so the *Diffie-Hellman* achieves the **perfect forward secrecy**, that means that if an *attacker* discovers secret information about a *previous session* of *message exchanging*, he cannot be able to retrieve any information for the next session of *message exchanging*. The *computation time* is $O(log^3 p)$.

Diffie-Hellman is effective only against a *passive adversary* (**sniffing**), instead suffers the **Man-In-The-Middle attack**, which is *lethal* for this *protocol* and the attacker can control the whole conversation.The *attacker* makes *independent connections* with the victims, and relays *messages* between them, making them believe that they are talking directly to each other over a *private connection*.

Eve **intercepts** Alice's *public value* and sends her own *public value* to Bob. When Bob transmits his *public value*, Eve substitutes it with her own and sends it to Alice. Eve and Alice thus agree on one **shared key** and Eve and Bob agree on another **shared** key. After this *exchange*, Eve simply decrypts any *messages* sent out by Alice or Bob, and then *reads* and possibly *modifies* them before *re-encrypting* with the appropriate *key* and transmitting them to the other party. This *vulnerability* is present because **Diffie-Hellman key exchange** doesn't **authenticate** the participants.



In order to have a secure **DH protocol**, an **authentication system** must be added, in fact, most *cryptographic protocols* include some form of *endpoint authentication* to prevent this *attack*. *Key Enchange* is used by **VPN**, that are made by two *protocols*, an *Handshake Protocol*, in which we have a *key exchange* between parties sets *symmetric keys*, and a *Traffic Protocol*, in which *communication* in *encrypted* and *authentication* is made by *symmetric keys*.

5 RSA

RSA is the implementation of the idea of Diffie and Hellman on the Public Key Infrastructure, and these implementation need some mathematical properties:

5.1 Math considerations

- **Multiplicative Group Z_{pq}^***:
  - Let p and q be two large *prime numbers*, and let's denote their *product* with $N = pq$, N it's not *prime*, but it's a **semiprime** (a product of two primes);
  - The **multiplicative group** $Z_{pq}^* = Z_N^*$, contains all *integers* in the range $[1, pq-1]$ that are **coprime** to both p and q. It's **cardinality** it's not $pq-1$ cause N it's not *prime*. There are $(p-1)$ multiples of q and $(q-1)$ multiples of p so the *cardinality* (the size of the group) is:

$$\phi(pq) = |Z_{pq}^*| = pq - 1 - (p-1) - (q-1) = pq - (p+1) + 1 = (p-1)(q-1);$$

  - So for the **theorem of Euler**, we have that for every $x \in Z_{pq}^*, x^{(p-1)(q-1)} = 1$;

- **Exponentiation in Z_{pq}^***:
  - We notice that not all the integers between $[1, pq-1]$ are in Z_{pq}^*, this means that each element in that range doesn't have necessarily have a *unique inverse* in Z_{pq}^*. We want an e that is in the range $[1, (p-1)(q-1)]$, such that the power operation $x \to x^e$ is a **one-to-one operation** in Z_{pq}^* (*every element of the* function's codomain *is the image of at most one element of its* domain);
  - Since an element in the range $[1, pq-1]$ can have more than one inverse in Z_{pq}^*, then we cannot choose e at random;
  - If we set e as a coprime to $(p-1)(q-1)$, then since the $gcd(e, (p-1)(q-q)) = 1$ (*from the definition of coprime, gcs is mcd*), e has a **modular multiplicative inverse** called d of *mod* $(p-1)(q-1)$ (*a number for which* $(e \cdot d)$ *mod* $(p-1)(q-1) = 1$);
  - Then we have that $ed = 1 + C(p-1)(q-1)$ where C is a constant;
  - Let $y = x^e$, then:

$$y^d = (x^e)^d = x^{1+C(p-1)(q-1)} = x^1 \cdot x^{C(p-1)(q-1)} = x(x^{(p-1)(q-1)})^C = x \cdot 1 = x;$$

  - This means that the power $y \to y^d$ is the inverse of $x \to x^e$, so $x \to x^e$ is a **one-to-one operation**.

5.2 RSA Protocol

- Choose two *prime numbers* p and q and let $N = pq$ be their *product*;
- Choose an e such that $1 < e < \phi(N)$ and $gcd(e, \phi(N)) = 1$) so e is *coprime* with $\phi(N)$;
- Let d such that $de \equiv 1\ mo\ \phi(N)$, so d is the *modular multiplicative inverse*;

- Then the public key is (e, N);
- The private key is (d, N);
- We will encrypt $M \in Z_n$ by $C = E(M) = M^e \ mod \ N$;
- We will decrypt $C \in Z_n$ by $M = D(C) = C^d \ mod \ N$;
- Decryption is valid since:
  - If $ed = 1 \mod \phi$ then there is an integer k that $ed = 1 + k\phi$, where $\phi = (p-1)(q-1)$;
  - If $gcd(m, p) = 1$ then by **Fermat theorem** $m^{p-1} = 1(mod\ p)$ so if we power both the sides for $k(q-1)$ we obtain: $m^{k(p-1)(q-1)} = 1^{k(q-1)} \ mod\ p$, ($1^{k(q-1)}$ does 1) and then we multiply both sides for m and we get: $m^{1+k(p-1)(q-1)} = m(mod\ p)$;
  - If instead $gcd(m, p) = p$ then still equal cause both sides are congruent to 0 $mod\ p$ cause $m = jp$ for some $j \geq 1$ so m is multiple of p;
  - In both cases we have that $m^{ed} = m(mod\ p)$, similarly $m^{ed} = m(mod\ q)$;
  - This means that $m^{ed} = m(mod\ N)$
  - So this proof says, since m is *coprime* with p, and since m is *coprime* to q then it will be *coprime* even with N;

RSA uses a *variable-length key*, usually 512 bits, and a *block size* also variable, but it's important that the **size of the plaintext is less than N**, $|plaintext| < N$, and that the $|ciphertext| = N$, it's slower than *DES*, and it's not used in practice for *encrypting long messages*, and it is used to *encrypt* a *secret key* (instead of a *message*);

An example of RSA: $p = 47, q = 59$ so $N = 2773$ and $\phi(N) = 46 \times 58 = 2668$, we pick a $e = 17$ and $d = 157$, cause $(17 \cdot 157 \ mod\ 2668 = 1)$, for $N = 2773$ we can encode two letters using a two digit number per letter: $A = 01, B = 02, ...$

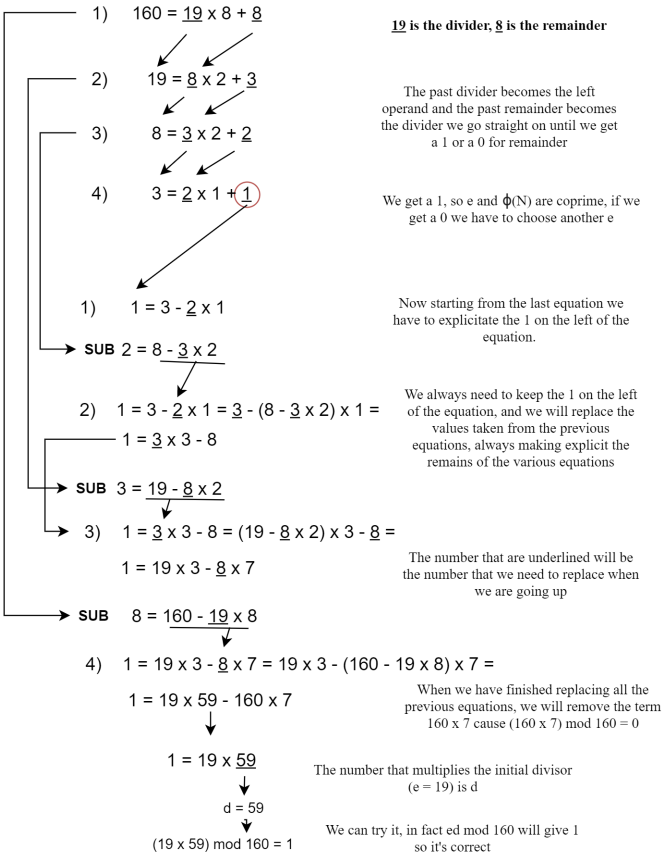
Since e and d are **symmetric**, since they are *bounded* by the *relation* $ed = 1 mod \phi(N)$, it doesn't matter which you choose first, but from a *security prospective* these values are not *symmetric*: in fact e is *public* (used for *encryption*), and d is *private* (used for *decryption*), so we need that d is a hard value to guess:

- If we choose e and then compute d, we can choose e to be a small value, and this is a real advantage, cause the *encryption* is much faster and without any loss of security;
- If we choose d and then compute e, we have to select a large value for d, and this slows down the operation;

In order to compute d we use the **Beazout's identity** and the **Extended Euclid's Algorithm**:

- $au + bv = d$ where $a > b$ and u, v are signed integers then there exists d that is the common greatest divisor of a and b;
- If a and b are coprime, then $d = 1$, and u is the modular multiplicative inverse of a instead v is the modular multiplicative inverse of b;

- The **Extended Euclid's Algorithm** uses this *identity* in order to compute the d starting by the e, and since the *math proof* is too incomprehensible I will just put an exercise to expose how it works:
  - We have two *prime numbers* $p = 11$ and $q = 17$, so we have that $N = p \cdot q = 187$ and $\phi(N) = (p-1)(q-1) = 160$;
  - Now have to choose an e that is **coprime** with $\phi(N)$ that is in range $[1, \phi(N)]$ (*usually we take a prime number, so we have sure that is coprime with* $\phi(N)$) in this case we choose $e = 19$;
  - Now we know that $ed \equiv 1 \mod \phi(N)$ so we have: $19 \cdot d \equiv 1 \mod 160$;
  - We start using the **Extended Euclid's Algorithm**: in the equation at the left we put the $\phi(N)$ and at the right we will divide it for e plus the *remainder* so:



  - If we obtain a d (that can be also negative) that is not in the range $[1, \phi(N)]$ we can sum the value of $\phi(N)$ (how many times you need). In fact it's easy to see, that also with this exercise if we sum $\phi(N) = 160$ to $d = 59$, we obtain $d_1 = 59 + 160 = 219$ and we compute $e \cdot d_1 \equiv 1 \mod 160$ it still gives 1 as result: $(19 \times 216) \mod 160 = 1$, and this is also valid for $d_2 = 59 + 160 \times 2, d_3 = ...$;

5.3 Attacks against RSA

RSA can be seen as a **one-way trapdoor function**, that is a *function* for which there is a *trapdoor* s such that:

- Without knowledge of s the *function* is a *one-way function*;
- Given s inverting the *function* is easy;

So in the **RSA** the **trapdoor** is of course d in fact given d going back from the *encryption* is very easy, instead without it is **very hard**.

5.3.1 Factorization

A first, trivial attack against *RSA* could be try to **factorize** N is order to get p and q from that $\phi(N)$ and then d. So it's recommended to take p, q large enough like 100 digits each, and to make sure that p and q are not too close together. (Since *prime number* are relatively frequent in $[N, 2N]$ and there are at least $fracNlnN$ *primes* in this range). It's important that $(p-1)$ and $(q-1)$ have large *prime factors* to foil **Pollard's rho algorithm**.

5.3.2 Weak Messages

There are weak message that are not good messages to be encrypted with RSA protocol and they are:

- $m = 0$ cause 0 to the power of anything is 0 and so the ciphertext will always be exactly equal to the plaintext;
- $m = 1$ for the same reason as above;
- $m = N - 1$ cause if the exponent is $e = 3$ then we have that $(N-1)^3 \ mod \ N = (N-1)^2(N-1) \ mod \ N$, that can be written as $(N-1)^2 \ mod \ N = 1 \ mod \ N$, but since $(n-1)^2 \ mod \ n = 1$ then $(N-1)^2(N-1) \ mod \ N = (N-1) \ mod \ N$. Therefore the *ciphertext* is equal to the *plaintext*, and this works for every odd $e \geq 3$. We need to use **salt** that is a sequence of *randoms bits*.
- If both m and e are *small* then we might have that $m^e < N$ and therefore we have $m^e \ mod \ N = m^e$ so the *adversary* can compute it and find m, the solution is to add *non-zero bytes* to avoid small *messages*.
- If we have a small e (suppose $e = 3$) and two *messages* are **similar**, let's suppose one is m and the other is $m + 1$ then the two ciphertexts are $C_1 = m^3 \ mod \ N$ and $C_2 = (m+1)^3 \ mod \ N$ then it's easy to write a system of *equations* and we can get:

  $$m = \frac{(C_2 + 2 \cdot C_1 - 1)}{C_2 - C_1 + 2)}$$

  So the *adversary* will be able to get the two *messages* easily, and this works for every *pair* m_1 and $m_2 = \alpha m_1 + \beta$. The solution is choose a large e.

5.3.3 Chinese Reminder Attack

The **Chinese Remainder Attack** is an *attack* in which an user send the same *message* to three different users with the same **public key** and we assume $e = 3$, let's say $(3, n_1), (3, n_2), (3, n_3)$. *Adversary* knows *public keys* and will know: $m^3 \ mod \ n_1$, $m^3 \ mod \ n_2$ and $m^3 \ mod \ n_3$. So he can compute $m^3 \ mod \ (n_1 \cdot n_2 \cdot n_3)$ by using the **Chinese Reminder Theorem**, so we have that $m < n_1, n_2, n_3$ and this means that $m^3 < n_1 \cdot n_2 \cdot n_3$ and therefore we have that: $m^3 \ mod \ n_1$, $m^3 \ mod \ n_2 = m^3$. So the *adversary* will compute *the cubic root* and will get the result, the *solution* is to add *random bytes* to avoid equal *messages*.

5.3.4 Same N

Another *attack* is when two *users* choose the **same N**, this means that they have different p and q, so one of them can try to compute p and q starting from his e, d, N. Then the user could easily discover the *secret key* of the other user given his *public key*. The *solution* is that each person chooses his own N.

5.3.5 Multiplicative Property of RSA

Another *weak property* is that if a message M is the *product* of two other *messages* M_1 and M_2 then by the **invariance over multiplication** we have that:

$$M^e \ mod \ N = (M_1 \cdot M_2)^e \ mod \ N = (M_1^e \ mod \ N \cdot M_2^m od \ N) \ mod \ N$$

So an *adversary* can proceed using **small messages**. The *adversary* wants to *decrypt* $C = M^e \ mod \ N$, so the *adversary* will compute $X = (C \cdot 2^e) \ mod \ N$. He will then use X as chosen *ciphertext* and will ask to the *oracle* (owner of the *private key*) to encrypt the new *message* $Y = X^d \ mod \ N$, but we have that:

$$X = (C \ mod \ N) \cdot (2^e \ mod \ N) = (M^e \ mod \ N) \cdot (2^e \ mod \ N) = (2M)^e \ mod \ N$$

But this means that the *adversary* will get $Y = 2M$

5.3.6 Chosen Ciphertext Attack

Similar to the precedent the *adversary* knows a **ciphertext** $C = M^e \ mod \ N$, and will **randomly choose** an X. Then will compute $C^{'} = C \cdot X^e$ and will ask to the *oracle* to *decode* it. Then the *adversary* will compute $(C^{'})^d = C^d \cdot (X^e)^d = M \cdot X \ mod \ N$. So the *adversary* since it does know X can compute the *plaintext* M. The *solution* is that the *oracle* should verify if the *message* incoming respect a given *structure*.

5.3.7 Chosen Plaintext Attack

Also called **CPA**, is an *attack model* which presumes that an *attacker* has the capability to choose an **arbitrary plaintexts** to be *encrypted* and obtain the corresponding *ciphertexts*. The goal is to gain further *information* which reduces *security* of *encryption scheme*. There are two forms of *CPA*, **Batch CPA** where all *plaintexts* are chosen before any of them are *encrypted*, **Adaptive CPA** where *subsequent plaintexts* are based on information from the previous *encryptions*.

5.4 RSA Standards

At the end of all this discussion, we got that **RSA** presents some *weaknesses*, and therefore a good *approach* that has to be followed when using RSA is the following:

- Choose good numbers p, q, e, d such that they satisfy the *properties* discussed so far;
- Before *encrypting message* M, perform some *preprocess* on M in order to obtain $M^{'}$, ($M^{'}$ has not to change the meaning of M) and then perform the encryption of $M^{'}$;

For this reasons some **standards** have been invented in order to perform a good *RSA usage*.

5.4.1 Public-Key Cryptography Standard

Public-Key Cryptography Standard or **PKCS**, is a set of *standards* that specify how *RSA* should be used. We will be seeing only the **PKCS#1 standard** in which a message is pre-processed as:

$$m = 0 \parallel 2 \parallel at\ least\ 8\ non-zero\ bytes \parallel 0 \parallel M$$

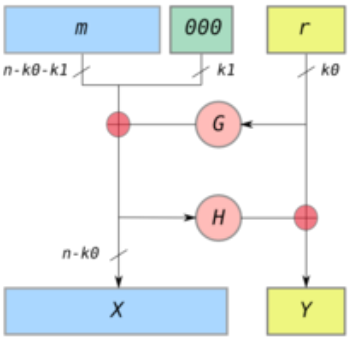
Where M is the *original message*. We set the *first byte* to 0 so we are sure that the *message* is less than N, the *second byte* equals to 2 denotes *encryption* of a *message* (if it is 1 it denotes *digital signature*). The *random bytes* imply that the same *message* sent to different people will result in different *ciphertext*.

5.4.2 Optimal Asymmetric Encryption Padding

OAEP is a *padding scheme* often used with RSA encryption that uses a pair of **random oracles** G and H to process the *plaintext* prior to *asymmetric encryption*. *OAEP* satisfy two goals:

- Add an element of *randomness* which can be used to convert a *deterministc encryption scheme* (the *traditional RSA*) in a *probabilistic scheme*;
- It prevent *partial decryption* of *ciphertexts* by ensuring that an *adversary* cannot recover any portion of the *plaintext* without being able to invert the *trapdoor*;

In the **encryption** we have that m is the *original message* a *string* of $(n - k0 - k1)$ bits, where $k0$ and $k1$ are two *integers* fixed by the protocol. G and H are two *cryptographic hash functions* and n it's the number of bits in *RSA modulus*. In order to **encode** the *original message* is *padded* with $k1$ zeroes, and G will expand r (a random $k0$ bit string) to $n-k0$ bits, then the *XOR* is done between it and the *padded message* which results in X. Then H reduces the $n-k0$ bits of X in $k0$ bits that will be *XORed* with r in order to get Y. The *output* will be $X \parallel Y$ which will be then *encrypted* with *RSA*. In the **decryption** the *outcome* will be *decoded* in order to retrieve m in the following way: $r = Y \bigoplus H(X)$ and then $m|000... = X \bigoplus G(r)$ then we discard the $k1$ zeroes and we obtain the *message*.



This scheme is also called **All-Or-Nothing security**, in fact, in order to retrive m we must have X and Y, since we need X in order to retrieve r from Y and we need r in order to retrieve m from X.

5.4.3 El Gamal Encryption

El Gamal it's a mode of *encryption* strongly based on *Diffie-Hellman key exchange approach*, indeed we will see that the underlying math is the same. *El Gamal* is an alternative to *RSA* and so is able to generate a *pair of keys public/private*, in order to perform **encryption** and **decryption**. The protocol is the following: given a *prime* p and a *generator* of the Z_p *cyclic group* of order p, *Alice* chooses an x in the range $[1, p-1]$ as the **private key**, and she chooses the number $g^x \ mod \ p$ as the **public key**. If *Bob* wants to send a *message* $m \in Z_p$ to *Alice*, he chooses a number y in the range $[1, p-1]$ and he sends to *Alice* the pair: $(g^y \ mod \ p, m \cdot g^{xy} \ mod \ p)$. He can send the second *message* cause he knows m, y and $g^x \ mod \ p$ which is the *public key* of Alice. When *Alice* gets the *pair* of *messages*, she uses the first one in order to compute $g^{xy} \ mod \ p$ then she finds a number $(g^{xy})^{-1} \ mod \ p$ and so she can get the *message* as follows: $m = m \cdot g^{xy} \cdot (g^{xy})^{-1} \ mod \ p = m \ mod \ p = m$. This scheme requires two *exponentiations* per each *transmitted block*, so it require a little bit demanding in terms of *computational effort*.

6 Digital Signature - DSA

A **Digital signature** is a mathematical scheme for demonstrating the **authenticity** of *digital messages* or *documents*. A valid *digital signature* gives a recipient reason to believe that the *message* was created by a *known sender* (**authentication**), that the *sender* cannot deny having sent the *message* (**non-repudiation**), and that the *message* was not altered in transit (**integrity**). In few words, a *digital signature* provides *authentication*, *integrity* and *non-repudiation*. We already studied an approach for *data integrity*, it was based on *MACs*. Let's suppose that Alice and Bob share a secret key K, then $(M, MAC_K(M))$ will convince Alice that the message was created by Bob. But in case of dispute A cannot convince a judge that $(M, MAC_K(M))$ was sent by Bob, since A could generate it herself.

Then we discussed *RSA* and in general about the usage of **public and private keys**:

- **Encryption with other party's public key**: you get **confidentiality** (he *decrypts* with his *private key*);
- **Encryption with my private key**: **no confidentiality**, but **authentication** (everyone is enabled to *decrypt* my *ciphertext*, it is just sufficient to use my *public key* in order to do it, but in this way other *parties* can be sure about the fact that it was me who *encrypted* the *message*);

So, it's easy to **forge signatures** of *random message* even without holding D_A, in fact *Bob* can pick an R arbitrarily and can compute $S = E_A(R)$, this means that the pair (S, R) is a **valid signature** of *Alice* (even if it was created by *Bob*), so the scheme is subject to **Existential Forgery**, that is the ability of an *adversary* to create a *pair*, *message* and *signature*, where the *message* has not been signed in the past by the *legitimate signer* but the *adversary* doesn't have any control over the *message*. In the **Selective Forgery** instead, the *adversary* creates a pair *message signature*, where the *message* has been chosen by the *adversary* prior to the *attack*, if an *adversary* can do the *selective forgery* it can also do the *existential forgery*. In the **Universal Forgery** instead an *adversary* can create a *valid signature* for any given *message*, and this is the strongest ability in *forging* and it implies all the other types of *forgery*. So we need a method to prevent the *Existential*

Forgery and this method is the **hashing**.

An *adversary* can use the **forgery attack**, in fact he can generate a *random file* T and will *encrypt* it with the K_p (*public key of Bob*). The *ciphertext* generated will be $R = E_{K_p}(T)$ then the *attacker* sends R as *plaintext* and T as *digital signature*. *Alice* will take the *digital signature* T and so she *encrypts* T with K_p of *Bob* and she gets $E_{K_p}(T) = R$, but since the *attacker* sent R as *plaintext*, *Alice* sees that the *decryption* of the *digital signature* matches with the *plaintext* sent, so she accepts the *document*. That's why all this is not good.

Let E_A be *Alice public encryption key*, and D_A the *private decryption key*:

- In order to **sign** the *message* M, *Alice* will compute two *strings*: $y = H(M)$ and $z = D_A(y)$ and then will send to *Bob* (M, z);
- In order to **verify** the *Alice's signature*, *Bob* will compute the *string* $y = E_A(z)$ and will check if $y = H(M)$;

Alice create z with D_A (the *decrypt function*) cause it need to use it's **private key** to *encrypt* it, instead *Bob* will read y using E_A (the *encrypt function*) to *decrypt* it with the **public key** of *Alice*. The function H should be **collision resistant** so that cannot find another $M^{'}$ with $H(M) = H(M^{'})$. With **hashing** the process of *signing* is much faster, we are using *asymmetric encryption*, which is slow, but here it is performed on the *hash* of the document which is much smaller than the *plain document*.

6.1 Standards for Digital Signatures

The structure of the *general digital signature* is divided in three steps, the **generation** of *private and public keys* (in a *random* way), the **signing** (*deterministic* or *randomized*), and the **verification** (*accept* or *refuse*) usually *deterministic*. There are three different schemes used in practice: **RSA**, **El Gamal** and the **DSS**.

6.1.1 RSA and PKCS#1

We already saw the **PKCS#1 standard**, the *signature* is made by *hashing* using the *private key*, only the person who knows the *secret key* can *sign*, everybody can verify the *signature* by using the *public key*. We construct the *digital signature* by *encrypting* with our *private key* the following *message*:

$$0 \parallel 1 \parallel at\ least\ 8\ byte\ FF\ base\ 16 \parallel 0 \parallel specs\ of\ hash\ function \parallel hash(M)$$

Where:

- The first byte 0 ensures that the *message* is less than N;
- The second byte 1 denotes *signature*, if it's 2 it means that the *message* is *encrypted* for *confidentiality* purpose;
- The bytes 1111111..... implies that the *encoded message* is large;
- And M is the *original message*;

As a conclusion for **RSA** I want to stress the concept that when you are using *RSA* you want to *sign*, the other typical usage of *RSA* is send an **encrypted key**. You don't use *RSA* for *confidentiality* purpose, for *encrypt* a *document*, you *encrypt* just a *key*. Because of its *complexity*, *computational effort,* much bigger than Rijndael. PKCS#1 define a **Signature Scheme with Appending** or **SSA**, in which the *appendix* is the *signature* added to the *message*, and we don't consider *signature schemes* with *message recovery* (*message* is embedded in the *signature*). There are two approaches that differ in how the **encoded message** is obtained:

- **RSASSA-PSS**: a *probabilistic signature scheme* that uses **EMSA-PSS** an *encoding method* for *signature appendix*, inspired to *OAEP* that makes use of *random salt* added per *signature*, in which first the message is hashed, after the output is padded and we add a salt at the end so we generate a new message, and this message will be hashed and then XORed with another padding and salt that will be concatenated to $h(M)$;
- **RSASSA-PKCS1-v1.5**: that is *deterministic*;

6.1.2 El-Gamal Signature Scheme

El-Gamal scheme can be divided into three parts: *generation*, *signing* and *verification*. The **generation** of the *key* is made as:

- Choose a *prime* number p of 1024 bits such that the **Discrete Logarithm (DL)** in Z_p^* is *hard*, the only way to get the *private key* x is by computing the *discrete logarithm* of p in base g;
- Find a **generator** g og Z_p^*, an integer for which the *powers* are congruent with the numbers *coprime* with p;
- We pick a x in $[2, p-2]$ at random;
- Compute $y = g^x \ mod \ p$;
- Then the public key $K_p = (p, g, y)$ and the private key $K_s = x$;

In order to **generate** the *signature* instead, if we want to sign a *message* M we define a new pair of *keys* (r, k), where r is a *public key* instead k is a *private key* used in order to have a different *signature* every time:

- We will start by compute the *digest* $m = H(M)$;
- We will pick k in $[1, p-2]$ coprime to $p-1$ at random;
- After we will compute $r = g^k \ mod \ p$ and $s = (m - rx)k^{-1} \ mod \ (p-1)$, if s is 0 then we need to restart;
- The *output signature* will be (r, s);

The **verification** follows this pattern:

- First we compute $m = H(M)$;
- We will accept only if $0 < r < p \wedge 0 < s < p-1 \wedge (y^r \cdot r^s = g^m) \ mod \ p$;

- Since we have that $s = (m - rx)k^{-1} \ mod \ (p-1)$ we have that $sk + rx = m$ *(mod(p-1) disappears cause mistero della fede)*. Now:
  - We know that $r = g^k \ mod \ p$ so if we power left and right operands for s we obtain: $r^s = g^{ks} \ mod \ p$;
  - We know that $y = g^x \ mod \ p$ so if we power left and right operands for r we obtain: $y^r = g^{rx} \ mod \ p$;
  - If we multiply we obtain: $y^r \cdot r^s = g^{ks+xr} \ mod \ = g^m \ mod \ p$;

6.2 DSS, Digital Signature Standard

DSS is important because is the choice made by *NIST*, it is inspired on *El-Gamal approach* and according to the original formulation *DSS* is using *SHA*, just because the *hashing function* chosen by the *NIST* is *SHA*. There are two acronyms: **DSS**, **Digital Signature Standard** and **DSA** , **Digital Signature Algorithm**. The second is just an *algorithm* in order to get the *digital signature*.

The idea of **DSA** is to use two prime numbers p and q such that:

- p is a L bit prime number such that the Discrete Log problem $mod \ p$ is intractable;
- q is a 160 bit prime that divides $p-1$, so we have that $p = j \cdot q + 1$, so j is a constant equal to $j = \frac{p-1}{q}$;

Then let α be a q^{th} root of 1 $mod \ p$, so we have that $\alpha^q = 1 \ mod \ p$. In order to compute α we need to take a random number h such that $1 < h < p-1$ and we compute $g = h^{\frac{p-1}{q}} \ mod \ p = h^j \ mod \ p$, since we need to apply the *little theorem of Fermat*, we need that h is *coprime* with p. If $g = 1$ we have to choose a different h for *security reason*. Then it holds that $g^q = h^{p-1} = 1 \ mod \ p$ by the **Fermat Theorem**, since p is prime and every number less than p raised to $p-1$ are equal to 1 $mod \ p$. Finally we choose $\alpha = g$.

Math proof: $g = h^j \ mod \ p = h^{\frac{p-1}{q}} \ mod \ p$ so if we power both the operators with q we have: $g^q = h^{p-1} \ mod \ p$ and that from the Fermat Theorem $g^q = (1 \ mod \ p) \ mod \ p$ so we obtain that $g^q = 1 \ mod \ p = \alpha^q$.

So now let's talk about the algorithm, we have to choose two prime numbers p, q such that $p - 1 = 0 \ mod \ q$ (so $p-1$ is a multiple of q) and we consider α as the q^{th} root of 1 $mod \ p$. Then the **private key** will be s a random integer $1 \leq s \leq q-1$ and the **public key**: $(p, q, \alpha, y = \alpha^s \ mod \ p)$. s will be hard to find since an *adversary* should compute the *discrete log*, and choosing a t that computing the discrete log of $mod \ p$ of the multiplicative group Z_p^* is hard. The *signature* is inspired to *El-Gamal*, in fact we choose a random integer *secret* in the range $[1, q-1]$ and we have two parts:

- Part 1: $P_1 = (\alpha^k \ mod \ p) \ mod \ q$;
- Part 2: $P_2 = (SHA(M) + s \cdot P_1) \cdot k^{-1} \ mod \ q$;

Also the **verification** is done in two parts:

- $e_1 = SHA(M)(P_2)^{-1} \ mod \ q$;
- $e_2 = P_1 \cdot (P_2)^{-1} \ mod \ q$;

And the *signature* will be accepted only if $(\alpha^{e_1} \cdot y^{e_2} \ mod \ p) \ mod \ q = P_1$. The *proof* of the *correctness* of this is:

Accept if $(\alpha^{e1}\ y^{e2} \bmod p) \bmod q$ = PART I
$e1$ = SHA(M) / (PART II) $\bmod q$
$e2$ = (PART I) / (PART II) $\bmod q$
Proof : 1. definition of PART I and PART II implies
SHA(M)= (- s (PART I) + k(PART II)) $\bmod q$ hence
SHA(M)/(PART II)+ s (PART I)/(PART II)=k $\bmod q$
2. Definit. of $y = \alpha^{s} \bmod p$ implies $\alpha^{e1}\ y^{e2} \bmod p = \alpha^{e1}\alpha^{(s\,e2)} \bmod p$
$=\alpha^{\text{SHA(M)/ (PART II) + s (PART I) /(PART II)} \bmod q} \bmod p = \alpha^{(k+\,cq)} \bmod p$
$=\alpha^{k} \bmod p$ (since $\alpha^{q} = 1$).
3. Execution of $\bmod q$ implies
$(\alpha^{e1}\ y^{e2} \bmod p) \bmod q = (\alpha^{k} \bmod p) \bmod q$ = PART I

So, starting from the *Part 2*, we isolate the $k \ mod \ q$, after from the definition of the y and from the *verify formula* we obtain $\alpha^{e_1}(\alpha^{s \cdot e_2} \ mod \ p) \ mod \ p$ that we can rewrite as $\alpha^{e_1 + s \cdot e_2} \ mod \ p$. Now we will use the formula obtained before and we get $\alpha^{k \ mod \ q} \ mod \ p$ that is equal to $\alpha^k \ mod \ p$ cause k is a value less than q. Note that the external $mod \ q$ there was since the beginning of the proof but we omitted it. At the end: $(\alpha^{e_1} \cdot y^{e_2} \ mod \ p) \ mod \ q = (\alpha^k \ mod \ p) \ mod \ q = P_1$.

The **security of DSS** is *ensured* cause the *secret key* s is not *revealed* and it cannot be *forged* without knowing it, and we use a *random number* for *signing* (k) that is not *revealed*, and we don't have any *duplicates* of the same *signature* even with the same *message*. It's important to note that if k is known then you can compute $s \ mod \ q = s$, since s is less than q, and that we have two *signed messages* with the same k then k value can be *reveled*, so combined with the precedent equation it's possible to find s.

Finding two *primes* p and q such that $p - 1 = 0 \ mod \ q$ it's **not easy** and can take time, and p and q are *public* so they can be used by *many persons*. So, we have that *DSS* is faster than *RSA* for *signing* if we have some *preprocessing*, but the problem is to generate *random numbers* that may depend on the *message* so in this case we can't have *preprocessing*. *RSA* is usually used for *signature* and for *key management*.

When a *document* needs to be *signed* in many cases it's important for a formal point of view that also the the **time** is certified. This is called **Timestamping**, so we associate a *timestamp* to a *document*, in many cases we use a *third party* (an *authority*) also called **TSA** (**TimeStamping Authority**), who signs the *document* with the *timestamp*. If *Alice* want to *timestamp* a *document*:

- *Alice* compute *hash* of *document* and sends it to *TSA*;
- *TSA* will add the *timestamp*, will compute a new *hash* (of *timestamp* and *received hash*) and will *sign* the obtained *hash* and after will reply to *Alice*;
- *Alice* will keep the *TSA signature* as a proof, so everybody can check the *signature*, and *TSA* doesn't know *Alice's document*;

7 Authentication

Authentication is the procedure of providing the **identity** to a *party* in order to *access services* that this *party* provides to the *allowed ones*, in our example *Alice* needs to show her *identity* to *Bob*, cause *Bob* wants to show *information* only to *authorized people*. One of the most common *attacks* against *authentication* is **Spoofing**, that is the ability to cheat about the *identity* of the creator of the *message*, in fact with *authentication* we want to prevent *spoofing attacks*. We have two interesting cases of *authentication*:

- A *computer* is authenticating another *computer*;
- A *person* is using a *public workstation*, so we don't want to store *secrets* for all the possible *user*;

The **Closed World** assumption, it's an idea based on the sentence: **what is not currently known to be true, is false**. So it basically means that everything is *false* until we can prove that it's *true*. Many people don't like it because they prefers the concept of *black list* (forbidden or bad things) and *white list* (allowed things). Your *environment* is considered a **Closed Environment** when you are working in a *closed place*, like your enterprise. In many cases you are somewhat *dealing* with a **third-party** (a **trusted server**) that is protected by many methods, this *server* can support *authentication* of many *users*, we will see *Kerberos* that is a very *secure environment*. If the *attacker* belongs to the same company, it is called an **insider**, and it has a lot of power compared to an *external one*.

There are many differences between **authenticating a human and a computer**.

- **Computers** can store a *high-quality secret*, such as a *long random-looking number* (*key*), and they can do *cryptographic operations*;
- **Humans** instead are not able to do this, they use some *passwords* that are very different from a *key*;

The first difference between a *password* and a *key*, is that *humans* are stupid, so they use *password* that contains *words*, and *words* belongs to a *dictionary*, so an *adversary* can run an attack called **Dictionary Attack**, a *probabilistic attack* that use most common *words* in order to find a *password*. Also *password* are usually *short* and *easy to be guessed* by an *attacker*. Instead *computers* stores **long keys**, usually *hidden* or *encrypted* or also *one-time password*. One of the most common *attack* is the **Login Trojan Horse**, in which an *attacker* prompts a *fake login window* that induces the innocent user to prompt the *password* that is collected by the *attacker*.

The **authentication of people** can be based on *password*, *smart card*, *biometric tools* or *network address*. *Biometrics* are usually used, like *retina examination*, *fingeprinting reader* or *voice*, but we can have *false positive* and *false negative* (like *fake fingerprints*, or *modified voices*).

There are three main approaches in order to achieve **authentication** and they are:

- **Shared secret key**: *Alice* and *Bob* share a *secret key* and use it to *authenticate* each other;
- **Authentication through a Third Party**: there is an *external server* and if two *users* want to authenticate each other they have to pass through the *server*;

- **Public Key Criptography**: users use the *Public Key Infrastructure* in order to achieve *authentication*;

We can use different *techniques* in order to *authenticate* an *user* through a *key*, like **Timestamp**, or a **Nonce**, a random number chosen by the *user* that *authenticates* to verify the other knows the *key*, or **Sequence Numbers**, like in *TCP* that uses *sequence numbers* in order to do some things like tracking the order of *connections*. It's very important that we need to guarantee *authentication* and *integrity* that **Encryption doesn't guarantee Authenticity of a message**.

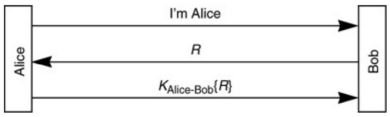
7.1 Authentication by Symmetric Key

We denote with K_{AB} the *secret key* shared between A and B, and with $K\{M\}$ we denote that M is *encrypted* with the *secret key* K, with R we denote the *Nonce*, but the challenge can be called R or K_{AB}. There are several ways to obtain **authentication** through **symmetric keys:**

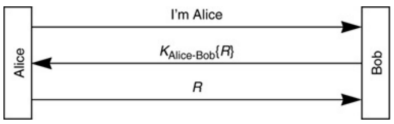
- **One Way authentication using Nonce (Challenge)**;
- **One Way authentication using Timestamps**;
- **Two Ways (mutual) authentication using Nonce**;

7.1.1 Challenge/Response Authentication

Alice want to **authenticate** *Bob*, she will send to him a *plaintext*, presenting herself, *Bob* will respond with a **challenge** (that is R, the *nonce*), then *Alice* will reply with the *encrypted version* of R with the *key* $K_{AB}\{R\}$ (the correct answer of the *challenge*). If *Bob* receives the right *encryption* then he authenticate *Alice*. This scheme is **not mutual** (in fact *Bob* authenticates *Alice*, but *Alice* doesn't). This scheme can be *easily attacked*, cause *Alice* cannot be sure that she is *authenticating* actually to *Bob*, so maybe she talking with the *attacker* (*Trudy*) that is able to send a *challenge* R and will get K_{AB} that can be derived from a **password** that *Trudy* can try to obtaining from R and K_{AB}.



There is **variant** in which *Alice* again start presenting herself, but now the *challenge* sent by *Bob* is the encrypted R and then *Alice* is expected to find the *nonce* R, then she sends R to *Bob* in order to be *authenticated*. This *variant* provides a little bit more *security*, because when *Bob* sends the *challenge*, we understand that only *Bob* can generate that *challenge* (only him knows the **private shared key**), so *Alice* can be sure it is *Bob* that is sending the *challenge*. So *Alice* can trust the *party* with who she is *authenticating*. This doesn't change a lot for the adversary cause he still can obtain R and K_{AB}. If R has a **limited lifetime**, like *random numbers* plus *timestamp*, meaning that we can consider it valid for a small amount of time after which it is not valid anymore, then *Alice* trusts *Bob*, because the limited lifetime of R does not allow **replay attacks**.

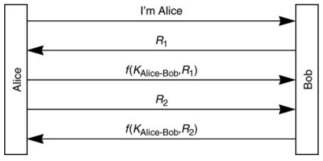


7.1.2 Timestamp Authentication

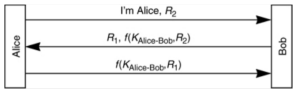
In which *Alice* and *Bob* have **reasonably synchronized clock** (this can be a weakness). *Alice* sends to *Bob* the *encrypted timestamp*, then if *Bob* can *decrypt* he can authenticate *Alice*, the *timestamp* has a *timeout* after which it expires, therefore if *Bob* saves that *timestamp* until expires, no *replay attack* is possible. It is very efficient in fact there are no *intermediate states* like before.

7.1.3 Mutual Authentication

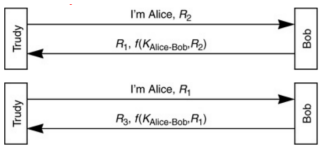
In the **Mutual Authentication** as the *one-way authentication*, *Alice* presents to *Bob*, and *Bob* reply with the *nonce* R_1, and *Alice* responds with the *encrypted version* of R_1. Now *Alice* is challenging *Bob* so she sends him the *nonce* R_2 and *Bob* sends back the *encrypted* R_2. But we have too many *messages*.



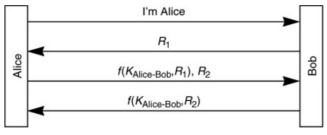
This **variant** is a compact way to do the *mutual authentication*, *Alice* presents to *Bob*, together with the *challenge* R_2, *Bob* replies with his *challenge* R_1 and with the response to *Alice's challenge*. Finally *Alice* sends back the response to *Bob's challenge*. In this way we only exchange 3 *messages*, and the result as before suffer for the **Reflection Attack**.



The **Reflective Attack**, is an *attack* that aims to *construct* an extra *session* useful just for the *adversary*. The *second session* will be left *uncompleted*, but is used by the *attacker* to complete the first one. *Trudy* is pretending to be *Alice* (*messages* are **spoofed**), so she sends to *Bob* "*I'm Alice, this is my challenge* R_2", *Bob* is correcting responding with *encrypted* R_2, but *Trudy* cannot send back the *encrypted* R_1 cause it doesn't know the *key*. So she start another *session* with *Bob* with the *challenge* R_1 that received from *Bob* in the *first session*, and in the *second session* he will get the *correct result* of the *encryption* of R_1 with another *challenge* R_3, that he will *ignore*. *Bob* can obviously suspect because the *second session* will be uncompleted. We can have two different solutions, like using *different keys* for each *session* (K_{AB} and $K_{AB}+1$) or *different challenges*.



This is another **variant less optimized**, in which the number of *messages* exchanges is 4. *Trudy* can set up an **offline-password attack**, so she plays as *Bob*, she sends the *challenge* and *Alice* sends her back the *correct response*, so *Trudy* by knowing the *pair* $(challenge, response)$ can guess the **secret key** with brute force.

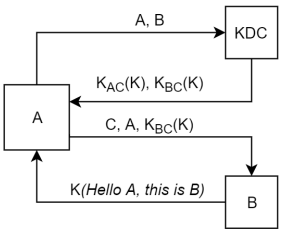


7.2 Authentication trough Third-Party

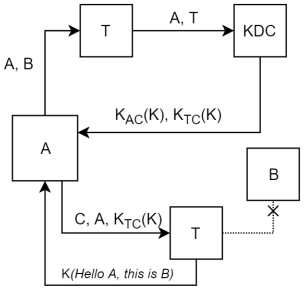
Since each *user* must have a *different key* for every other *user*, we have a *quadratic* number of *keys*, and this is a problem. For this reason we will use a **Trusted Party** that realizes the *authentication*. It is a very typical scenario used by companies and enterprises, and this *third party* is called **Key Distribution Center** (**KDC**), or **Authentication Server**, usually we denote it with C. Every *participant* in the *network* is sharing a *secret key* only with this *server*, so with n parties there are n *secret keys* that the *server* maintains and this means that every *user*, every *party* has to be registered to the *server*. Now having this *server*, we want to implement *single* or *mutual authentication* as before. An optional would be to use a **session key** for *short time communications*. If the *attacker* is an **insider**, he is sharing a *private key* with the *server*, he can *sniff* and spoof *messages*, but he cannot guess *random numbers* generated by the *parties* and cannot be able to get the *private keys* of other *parties* and cannot *decrypt messages* in short time without knowing the *key* because the *attacker* is expected not to have high *computational power*.

7.2.1 First schema

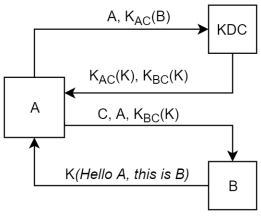
The **first schema** is presented just for educational purposes, in fact it's **very simple** and **weak**. *Alice* wants to talk with *Bob*, so she sends a message to $C : A, B$, saying *I am Alice, I want to talk with Bob*, C chooses the **session key** K and sends *Alice* two *messages* that are K *encrypted* with K_{AC} and K encrypted with K_{BC}. *Alice* will *decode* and will compute K, and will send to *Bob* a message constituted by three elements $(C, A, K_{BC}(K))$ like "*I'm Alice, I obtained this thing by* C, *this thing is* $K_{BC}(K)$". So Bob can decrypt $K_{BC}(K)$ and will find K and will reply to *Alice*, a *message* encrypted with the *session key*: K("*Hello Alice this is Bob*").



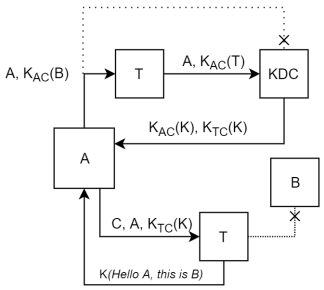
This *schema* can be **attacked** in fact, if we assume that *Trudy* can **sniff** and **spoof** then she can make the **Man In The Middle Attack**, with *Trudy* between them, so she can intercept *messages*, and create a *session key* between her and *Alice*, and another between her and *Bob*. *Alice* sends to *Trudy* (instead of C): A, B (*I am Alice, I want to talk with Bob*). Immediately *Trudy* sends to C: A, T (*I am Alice, i want to talk with Trudy*), so C will choose a K and will send to Alice: $(K_{AC}(C), K_{TC}(C))$. When *Alice* sends to *Bob* the *message* of three elements: $C, A, T_{TC}(K)$ *Trudy* will intercepts it and will reply to *Alice* with the *session key*: K("*Hello Alice this is Bob*").



In order to *solve* this **problem** we can modify the *protocol* in such a way that when *Alice* wants to talk with *Bob*, she send to $C : A, K_{AC}(B)$ so the *receiver* is **encrypted**, so an *attacker Trudy* cannot know that *Alice* wants to talk with *Bob*.



Even this scheme is *violable*, in fact if Alice want to talk with *Bob* like before $A, K_{AC}(B)$, but in precedence *Trudy* and *Alice* had already talk, *Trudy* can use these **previously exchanged messages**, and will intercept the *message* of A, and will send to $C : A, K_{AC}(T)$, even if *Trudy* doesn't know with who *Alice* wants to talk. C will choose a K and will send to Alice: $K_{AC}(K), K_{TC}(K)$. Now *Alice* will *decode* K and will send to B the message of three elements: $C, A, K_{TC}(B)$, *Trudy* will get it and finds that *Alice* want to talk with *Bob*, so she will now acts in place of *Bob*.



7.2.2 Needham-Schroeder Protocol

The **Needham-Schroeder Protocol** it's a very important *protocol* that poses the basis for **Kerberos** and it's used to create a *session key* between two *parties* by means of a *third party* and it's based on *Nonce* and *challenge response*. If *Alice* wants to talk with *Bob*, she chooses a *Nonce* N and will send to C: A, B, N. When C receives the *request*, will generate a *session key* K, and replies to *Alice* an *encrypted message* by means of the *secret key* of *Alice* $K_{AC}(N, K, B, K_{BC}(K, A))$, the *message* contains the *nonce*, the *session key*, the identity of *Bob*, and the *encryption* of K and A by means of the *key* of *Bob*. *Alice* will *decodes* the *message*, will check N and B, and will send to *Bob*: $K_{BC}(K, A)$. *Bob* will *decode* this *message*, will choose another *Nonce* $N^{'}$ and sends back to *Alice*: $K(\textit{This is Bob,}N^{'})$. *Alice* can reply with a *message encrypted*: $K(\textit{This is Alice,}N^{'} - 1)$ so *Alice* replies with the *nonce* decremented by one. It's important to note that *Bob* is not communicating with C. Also this *protocol* can be *attacked*, and this is called **Denning & Sacco Attack**, it works as a *replay attack* and uses an old session key. Like before *Alice* chooses a N and sends to C: A, B, N, then C chooses K and sends to A: $K_{AC}(N, K, B, K_{BC}(K, A))$ but now *Trudy* will acts in place of *Alice*. *Alice* will *decode* and check N and B and will send to *Bob* $K_{BC}(K, A)$. *Trudy* as *Alice* replays to Bob $K_{BC}(K^{'}, A)$, with $K^{'}$ an **older session key**. *Bob* will *decode*, will choose an $N^{'}$ but will send to *Trudy* instead of Alice $K^{'}(\textit{This is Bob,}N^{'})$. So Trudy will reply to Bob: $K^{'}(\textit{This is Alice,}N^{'} - 1)$.

We can also have *Trudy* that uses the *Man in the Middle Attack*, if we have a *session key* that is compromised. There are many possible variants of this *attack*, and all these *variants* are important just to let us understanding why we need to introduce more ingredients like **timestamps**, **sequence numbers** and so on. Therefore a *variant* of the *protocol* has been invented:

7.2.3 Needham-Schroeder Protocol variant

Since the *original protocol* was *weak* and we have seen some attacks against it, the *protocol* has been **revisited** and here is the new version:

- C exchanges *messages* with both A and B;
- *Timestamp* t is used only between B and C;
- K *session secret key*;

1. *Alice* chooses a *nonce* N and sends to *Bob*: (A, N);

2. *Bob* chooses a *nonce* $N^{'}$ and sends to the *Server* $(B, N^{'}, K_{BC}(N, A, t))$;

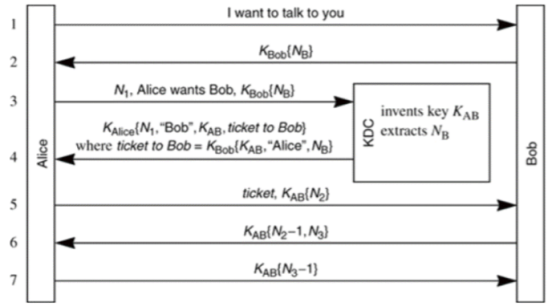
3. The *Server* sends to *Alice*: $(K_{AC}(B, N, K, t), K_{BC}(A, K, t), N^{'})$;

4. *Alice* sends to *Bob*: $(K_{BC}(A, K, t), K(N^{'}))$;

So in few words, *Alice* wants to talk to *Bob*, so she sends him a *nonce* N. *Bob* sees this *message* and create a *nonce* $N^{'}$ and a *timestamp* t and sends to the *Server* the *encrypted version* by means of K_{AC} of the two elements, so only *Bob* and the *Server* knows the *timestamp*, and the *Server* will have the power to say if the *timestamp* is valid or not. Then the *Server* sends to *Alice* the *encrypted session key* and the *encrypted timestamp* by means of K_{AC}, and another *encrypted message* by means of K_BC that *Alice* has to forward to *Bob*, with the *nonce* of *Bob encrypted* with the new *session key*.

7.2.4 Needham-Schroeder Protocol Expanded

Alice want to talk with *Bob*, *Bob* generate a *nonce* and sends it to *Alice* encrypted by the *secret key* of *Bob*, now *Alice* sends a *message* to the *Server*. Then the *Server* generates the *session key* K_{AB} and extracts the *nonce* created by *Bob*. Then the *Server* replies to *Alice* with a *message* that contains the *ticket* to *Bob*. *Alice decrypts* in order to get the *ticket* and sends it to *Bob* together with a new *nonce encrypted* by means of the *session key*. *Bob* replies with a *message encrypted* by the *session key*. The last *message* is not really necessary. This last version has inspired **Kerberos**.



8 Kerberos

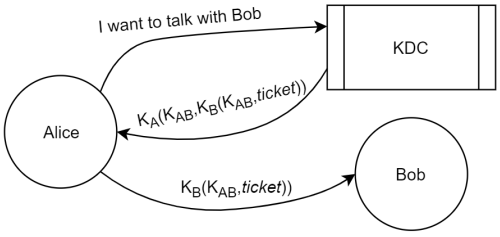
Kerberos is a *general framework* used in order to provide *authentication* in *distributed systems*, this *framework* is providing **safe access** to *resources* of a *network*. The main idea of this framework is that *users* that want to use the *service* of another party have to make a *request* to to the *KDC* for a **ticket**, and then they have to present this *ticket* to that *party* who offers the *service*. Of course every *user* share a *secret key* with *KDC*. Not only human are *users*, many times also *applications* but the difference is that while humans have not strong capabilities and their *keys* are derived from *passwords*, *applications* can store *long keys* (not predictable) that are not stored in *clear*.

The **Scenario** is: A needs to access *service* provided by B, so A need to be *authenticated* in order to use these *services*. In some cases we could also need B's *authentication*. Then there is a *trusted authority* C, that is a **reference Server trusted** by every party also called **KDC**, that

provides **session keys**. *KDC* is storing a **Master Key** for every *participant*, so every *user* has a *master key* that share with the *KDC*. The approach is to generate **tickets** allowing *users* to to access *services* and also there can be a **timeout** for *expiring tickets*. Even the *KDC* has a own *master key*, which *uses* to *encrypt* all the other *keys*, so the *KDC* stores all the *master keys* of the *users* encrypted by *KDC's master key*. It's also important that all *machines* must have their **clocks synchronized** cause **timestamps** are used in the *protocol*.

8.1 Kerberos preliminary implementation

The *idea* is **simple**, *Alice* wants to use a *service* of *Bob*, she is sending a *request* to *KDC* for the **ticket** and the *KDC* is providing it obviously *encrypted*, after that *Alice* will use this *information* with *Bob*. All the *information* provided by *KDC* is *encrypted* by the *master key* of *Alice* K_A and such *information* contains the *session key* K_{AB} and the **ticket** for *Bob*. This *ticket* contains further *information*, and is *encrypted* by means of the **master key** of *Bob* K_B, therefore only *Bob* can read such a *ticket*. Such *ticket* contains the *identity* of *Alice* of course, the *session key* and some other stuff like the *lifetime*. So basically when *Alice* asks to *KDC* to want to talk with *Bob*, *KDC* replies with $K_A(K_{AB}, K_B(K_{AB}, ticket))$. So *Alice* will *decrypt* the *message* obtaining the *session key* and the *ticket encrypted*, and will send the *encrypted ticket* $K_B(ticket)$ to *Bob* in order to show she is allowed to use *Bob's service*.



8.2 Kerberos simplified version

In the **simplified version of Kerberos** we have:

1. A sends to $C : A, B, N$;
    - So *Alice* asks to *KDC*, "I am *Alice*, i want to talk with *Bob*, here my *nonce* N" (this *nonce* will be used in order to verify the *authenticity* of C);
2. C sends to $A : [TicketB, K_A(K_{AB}, N, L, B)]$;
    - So the *KDC* reply to *Alice*, with a *ticket encrypted* in K_B (the *master key* of *Bob*), and with the *encryption* by K_A (the *master key* of *Alice*) of the *session key* K_{AB}, the *nonce* N, the **lifetime** L and the *identity* of *Bob*;
3. A will check N and known the lifetime, sends to $B : [TicketB, K_{AB}(A, t_A)]$;
    - So *Alice* will check the *nonce*, and learns about *ticket lifetime*, then *Alice* sends to *Bob* the *TicketB* and the **authenticator** which is $K_{AB}(A, t_A)$ and t_A is the *timestamp* of *Alice*;

4. B will check that A's *identity* in *TicketB* and in *authenticator* are the same, *time validity* of *ticket*;
    - So *Bob* will *decrypt* the *ticket* with his *master key*, obtaining the *identity* of *Alice* and the *session key*, then with the *session key* will check if the *authenticator* and the *identity* of *Alice* corresponds to the *identity* in the *ticket*, and will also check if the *ticket* is still valid with the *timestamp*;
5. B sends to $A : K_{AB}(t_A)$;
    - So *Bob* will reply to *Alice* with the *encryption* of the *timestamp* with the *session key* K_{AB};

But we have some **practical problems**, because for every *session* between *user* and *KDC*, the *messages* should be protected with the **master key** (derived from the *user's password*), so every time *Alice* requires a *ticket*, this *message* has to be *encrypted* but this cause too much overhead. We have 2 possible solutions:

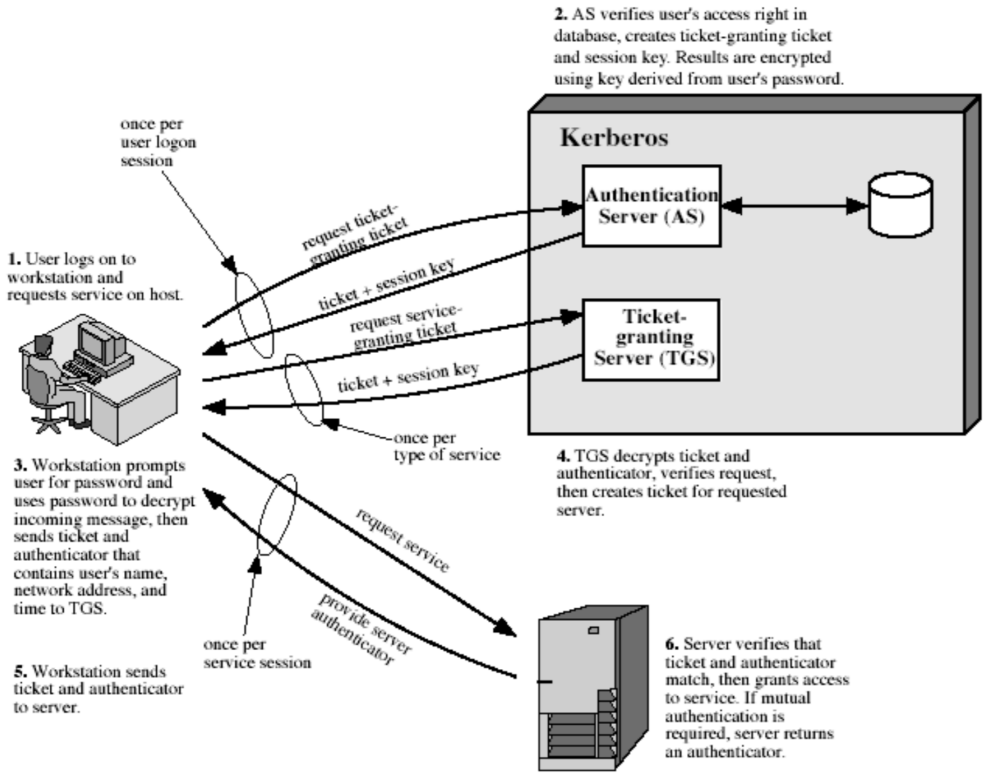
- Every time a *user* needs the *Server* he has to retype the *password*;
- The *user's password* is **temporarily stored locally**, in such a way that an *user* doesn't need to retype it;

These two solutions are *inadequate*, so we need to introduce another type of **Kerberos**:

8.3 Ticket-granting Ticket

TGT, or *ticket-granting ticket*, uses a *meta-ticket* used to ask the real *ticket*, the idea is that A is having a *ticket* for asking *tickets*, so every time he needs a *ticket* he shows a certain *ticket* instead of retyping the *password*. We have different steps:

1. *Alice* login for the first time and requests the *ticket-granting ticket* to the *authentication server* *AS*, that will check its local database;
2. *AS* will send to *Alice* the *ticket-granting ticket* and the *session key* S_A, with a fixed *lifetime* and other infos;
3. *Alice* now wants to use a particular *service*, so she will request it by showing the TGT to the *TGS* (*ticket-granting server*, the *server* that provides *tickets* for *services*);
4. *TGS* will answer with the *requested ticket* and a *session key* valid between *Alice* and the *service*;
5. *Alice* now can request the *service* to the *server* cause he has the *service-granting ticket*;
6. The server will answer with *server authenticator*;



In details can summarize all these *operations* in three *phases*:

- **Login phase**:
  - *Alice* does the *login* typing her *password* and the *KDC* derives from this *password* a *session key* S_A (with a *lifetime*, cause *KDC* will not keep it), then *KDC* will answer to *Alice* with a TGT and the *session key* S_A encrypted in K_A. This TGT is also *encrypted* with the *master key* of the *server* K_{KDC}, so *Alice* will have a message like: $K_A(S_A, TGT) = K_A(S_A, K_{KDC}(''Alice'', S_A))$;
- **Ticket request phase**:
  - In which *Alice* request to the *KDC* a *session key* in order to talk with *Bob*, so she need to be *authenticated* by sending to the server the TGT and an *authenticator* (given by the *timestamp* and the *session key* S_A). Once received the *server* will: decrypt TGT to get S_A, will decrypt the *authenticator* in order to verify the *timestamp*, will find *Bob's master key* K_B, and will create a *session key* K_{AB}, so with all these information will create a *ticket* to *Bob*: $K_B(''Alice'', K_{AB})$. So will send back to *Alice*: $S_A(''Bob'', K_{AB}, ticket\ to\ Bob)$ encrypted in S_A;

- **Ticket usage phase**:
  - Now *Alice* has the *ticket*, so in order to use it she sends it to *Bob*, together with the *timestamp* encrypted by K_{AB} (like in *simplified version of Kerberos*). So *Bob* will decrypt the *ticket* which contains the *session key* K_{AB} and so he will be able to decrypt the *timestamp* in order to trust *Alice*, and at the end will reply with an incremented *timestamp* in order to get authenticated by *Alice*.

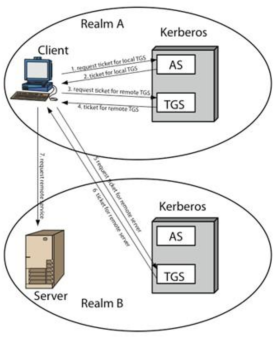
Authenticator is often the *encryption* of a *timestamp*, so we need a *Global Synchronous Clock*, and it is used to avoid the *replay* of *messages* sent to the same *server* by the *adversary* (old *message* are eliminated) and to avoid *replay* to a *server* (when there are many *servers*) and it's important to note that the *authenticator* doesn't guarantee *data integrity* so we need a *MAC*.

KDC and **TGS** are not the same, in fact they are separate *entities* for historical reasons, in fact one *KDC* can serve different systems of *TGS*. Commonly the *server* who provides the *TGT* is the **Authentication Server** (*AS*) and the *users* interact with it once a day (or once until the *TGT* is valid), while the *server* who provides the *service tickets* is the **Ticket-Granting Server** (*TGS*). The important thing is that we understand that the *Server* who provides the *TGTs* and the *Server* that provides the *Service Tickets* are **different entities**.

8.4 Kerberos Realms

In several cases an *user* want to use a *service* of another *network*, **Realms** is a way to implement that, and is like a *network* where there are *users*, *KDCs* and the *Master KDC*. Each *Realm* has a different **master KDC** (we have seen that in addition of a *KDC* there can be other *KDCs*), if you have let's say 2 realms, your *master KDC* should be a principle of the *master KDC* of the other *realm* and vice-versa. This means that there must be some *key agreed* between different *master KDCs*. You can setup so that you have just one *master key* shared between all possibles *KDCs*, but it is actually unrealistic. A very critical part of using a service in *Kerberos* is being *authenticated*, but who can authenticate a user? Only *KDC* of the same *Realm*! So once the *authentication* is made locally, then the *user* can ask some *service* coming from another *Realm*. The main scheme is the following:

Suppose we have a *client* in the *Realm A* which wants to use a *service* provided by the *server* in *Realm B*. First of all the *client* authenticates to its *local KDC* (in particular to the *AS*). Once *authenticated*, the client gets the *TGT* which will be used to ask to the *local TGS* for the *ticket* in order to communicate to the *KDC* of the other *realm*. Once obtained the *ticket*, the client can talk with the *TGS* of the other *realm* in order to obtain the *ticket* for the *server*. These cause *KDCs* store information only about users of the *local realm*, so first of all the client has to authenticate locally then a *TGS* can provide *tickets* only for *local services* or *ticket* for *TGSs* of the other *realms*, not for *services* of other *realms* all this for *scalability* reasons.



9 Authentication based on public keys & X.509 & PKI

The idea is to use **signed messages** containing *challenges* or *timestamps*, and the *signatures* can be verified using the **public key**. The problem is that is how can *Alice* be sure that the *public key* stored is the one of *Bob* and not the *public key* of an *attacker* (that if gives its own *public key*, then can be authenticated as another user), so we need to use a **trusted third party** that guarantees **correctness** of *public keys*.

9.1 Needham-Schroeder public key

We call K_{PX} the *public key* of an user X, and Sig_C the digital signature of C (the *trusted authority*). **Needham-Schroeder** tried implementation of *authentication* through *public key*. The scheme is simulating the presence of a *trusted authority* that provides *public keys* to users:

- A to C: $\langle A, B \rangle$;
  - So, In order to be **authenticated** *Alice* can ask to the *authority* "I am Alice, I want to talk to Bob";
- C to A: $\langle B, K_{PB}, Sig\_C(K_{PB}, B) \rangle$;
  - The reply is the **identity** of B, its **public key**, and the **digital signature** of this whole *message* (so the *encryption* of the whole message by means of *authority's private key*, so if we succeed in *decrypting* with *authority's public key* we can trust the content of the *message*);
- A checks **digital signature** of C, generate **nonce** N and sends to B: $\langle K_{PB}(N, A) \rangle$;
- B decode the *message* and send to C: $\langle B, A \rangle$;
  - So B will decode the *message* with its own *private key*, and will ask to the *authority* in order to check the **identity** of A, "I am Bob, I want to talk to Alice";
- C to B: $\langle A, K_{PA}, Sig\_C(K_{PA}, A) \rangle$;
  - So the *authority* will reply with the *identity* of A, its *public key* and the *digital signature* of the *message*;
- B checks *digital signature* of C, retrieve K_{PA} generate *nonce* $N^{'}$ and sends to A: $\left\langle K_{PA}(N, N^{'}) \right\rangle$;
- A will decode the *message*, checks N and sends to B, $K_{PB}(N^{'})$;
  - So at the end with this last message, **both the parties are authenticated**;

9.1.1 Needham-Schroeder public key Attack

This is a **Man in the middle attack**, in which just like *Alice* and *Bob*, *Trudy*, the *attacker*, can talk to the *server*. This *attack* is made in two sessions:

- **R1**: *authentication* between *Alice* and *Trudy*;
- **R2**: *authentication* between *Trudy* (that pretends to be *Alice*) and *Bob*;

The **attack** can have place only on a certain precondition: *Trudy* must be able to induce *Alice* to start an **authentication session** with *Trudy*.

- *Trudy* has succeed to convince *Alice* to issue an **authentication** with her, so *Alice* sends $K_{PT}(N, A)$ to *Trudy*;
- Then the *attacker* can use this *message* in order to start a **communication** with *Bob*, so will send to him: $K_{PB}(N, A)$;
- Then *Bob* will reply to *Trudy* (thinking that she is *Alice*) with $K_{PA}(N, N^{'})$;
- Of course *Trudy* **cannot decrypt** this, so she will forward the *message* $K_{PA}(N, N^{'})$ to *Alice*;
- *Alice* will **decrypt** it, and will send to *Trudy*: $K_{PT}(N^{'})$;
- At this point *Trudy* has got the **nonce** $N^{'}$, she can send to *Bob* $K_{PB}(N^{'})$ so she won his **challenge**;

So with this **attack** *Bob* thinks that he is talking with *Alice*, but instead there is *Trudy* in the *middle*. The only required thing is that *Trudy* is able to force *Alice* to issue an *authentication* with her, and if they belong to the same enterprise or something like this it is not so difficult because *Trudy* maybe has some reasons to invite *Alice* to be *authenticated* by *Trudy*.

9.1.2 Needham-Schroeder public key fixed variant

There is so a **fixed variant**, that permits to avoid the *attack* just described, that is very similar to the previous one, in which the only thing that changes is that when *Bob* is sending back the two *nonce* instead of $\left\langle K_{PA}(N, N^{'}) \right\rangle$ will send $\left\langle K_{PA}(B, N, N^{'}) \right\rangle$ so he send his **identity**. We can avoid the *attack* cause *Trudy* cannot send to *Alice* the identity of *Bob* while she is talking to *Alice*, in fact if *Trudy* will forward this message *Alice* will know that she is talking with *Bob* and not with *Trudy*.

We have not introduced yet the definition of **Digital Certificate**. The *Digital Certificate* is the message, sent by the *trusted authority* C, which contains the *public key* of someone together with the *digital signature* of this *message*. When *Alice* asks to the *authority* C for the key of *Bob*, and the *authority* replies with $\langle K_{PB}, Sig\_C(K_{PB}, B) \rangle$, this message can be considered a **Digital Certificate** that certifies that K_{PB} is the *public key* of *Bob*.

9.2 X.509 Standard

X.509 is a **standard** for *authentication* of different types, which means that they define *protocols*, not *encryption* or *hashing functions* specially related to protocol's steps. In this standard *3 authentication protocols* have been defined:

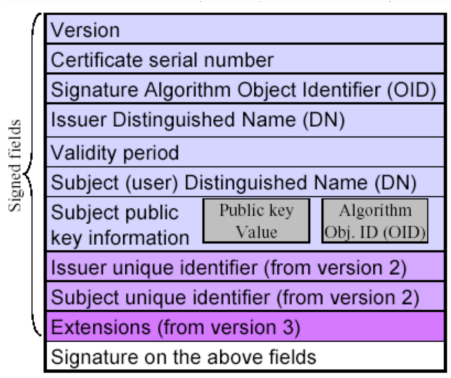
- **One-Way authentication protocol**:
  - Is very simple, if *Alice* wants to be authenticated by *Bob* she sends a single *message* that contains: the **digital certificate** of *Alice* (taken by the *trusted authority*), the **digital signature** of D_A and the *message* D_A that contains:
    - *Timestamp* t_A;
    - *Identity* of B;
    - The *session key* K_{AB} encrypted with the *public key* of *Bob*: K_{PB};
  - So, *Alice* must know in advance the key of *Bob*, so we can see this like *Alice* is the client and *Bob* the server, since this scheme is not symmetric. The *message* sent by *Alice* is: $\langle CertA, D_A, Sig\_A(D_A)\rangle$ where D_A is: $\langle t_A, B, K_{PB}(K_{AB})\rangle$;
  - **Nonce** can also be included in the *message* in order to detect *replay attacks*;
- **Two-Way authentication protocol (*mutual authentication*)**:
  - There will be a *message* from *Alice* to *Bob* ($A \rightarrow B$), and another *message* from *Bob* to *Alice* ($B \rightarrow A$). In the two *messages* D_A and D_B will appear something like $K_{PB}(k)$ (for Alice) and $K_{PA}(k^{'})$ (for Bob), these are **proposed session key**. This because it is not rare to encrypt *messages* $A \rightarrow B$ with a *session key* and $B \rightarrow A$ with another *session key* and both parties must know both *session keys*. We have to notice that in the first message *Alice* is sending the *proposed session* key encrypted by *public key* of *Bob*, but without having the *digital certificate* of *Bob*, and this is *unrealistic*;
  - So during $A \rightarrow B$ Alice will send $\langle CertA, D_A, Sig\_A(D_A)\rangle$ where $D_A = \langle t_A, N, B, K_{PB}(k)\rangle$;
  - Instead from $B \rightarrow A$ Bob will send $\langle CertB, D_B, Sig\_B(D_B)\rangle$ where $D_B = \left\langle t_B, N^{'}, A, N, K_{PA}(k^{'})\right\rangle$;
- **Three-Way authentication protocol**
  - The must robust version, in which the authentication is based on *nonces*, and it is useful for *unsynchronized clocks*. We have the exchange of **three different messages** (O represents a *timestamp* since it is *optional* in this *protocol*) :
    - *Alice* will send to *Bob* $\langle CertA, D_A, Sig\_A(D_A)\rangle$ where $D_A = \langle O, N, B, K_{PB}(k)\rangle$;
    - *Bob* reply to *Alice* with: $\langle CertB, D_B, Sig\_B(D_B)\rangle$ where $D_B = \left\langle O, N^{'}, A, N, K_{PA}(k)\right\rangle$;
    - The last *message* is from *Alice* to *Bob*: $\left\langle B, Sig\_A(N, N^{'}, B)\right\rangle$ so she send the two *nonces* and the identity of *Bob* signed by the **digital signature**;

9.3 PKI: Public Key Infrastructure

Certificates are provided by trusted **Certification Authorities**. The specific *Certification Authority* (**CA**) provides *certificates* just for the *users* in its domain, when someone asks for the *public key* of an *user*, the *CA* replies with the *public key* of the *requested party* signed by the *private key* of the *CA*. This also means that we have only to remember just one *public key*, that is the **CA's public key**. It's important to note that if *CA* is not *trusted* all its *certificates* are useless, *keys* are not used forever and the length of the *keys* is related to security level.

9.3.1 X.509 Certificate's Fields

The **X.509 standard** describes the structure of a **Digital Certificate**. The scheme is the following:

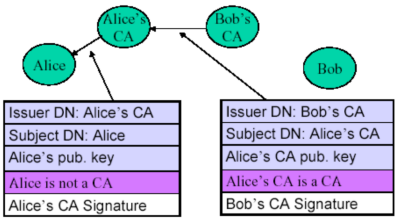


The **certificate** is composed by many **fields**, this cause *fields* are related to the management of such *digital certificate*, the practical management. For instance there is the **serial number field**: every *certificate* is having a *serial number* that is unique among all the *digital certificates* of the same *CA*. Then there is the *validity period*. There is also the *name* of the *user* (the *identity*), the *public key* of the *user*, and information about the *algorithm* in order to use that *public key*. The *extensions field* can contain *extensions*, that are *additional fields*. All the *fields* are signed by the **certification authority** but in order to check the *digital signature* you need the *public key* of the *CA* that is given by another *CA*, but you want to trust also that different *CA* so this generates a *chain* and this is a problem that we will be studying how to solve it.

- **VERSION**: there are three versions, *version 1, version 2, version 3*. If *version 1* is used we find 0 in this filed, if *version 2* we find 1, if *version 3* we find 2;
- **SERIAL NUMBER**: integer number concatenated with the *CA's name* and this is a unique identifier;
- **SIGNATURE ALGORITHM**: it describes the **algorithm** used for signing this *certificate*;
- **ISSUER**: it is the name of the *CA* creating the *certificate*, it is a name that follows the *X.500 standard*;
- **VALIDITY**: it contains 2 *subfields*, the time the *certificate* becomes *valid* and the time for which after it the certificate is *not valid* anymore, so it is an **interval of validity**;
- **SUBJECT**: name (according to *X.500 standard*) of the entity whose *key* is being certified;
- **SUBJECT PUBLIC KEY**: it contains two *subfields* that are the *algorithm* with which the *public key* has been created, and the *public key* itself.
- **...**

9.3.2 Hierarchy of CAs

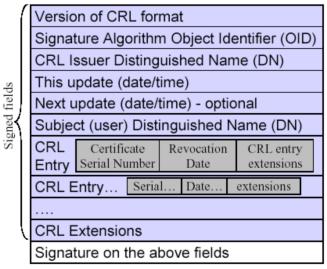
Now let's come back to the problem of how *Alice* and *Bob* can check their *certificates*, if *Alice* and *Bob* use the same *CA* there is no problem, but if they use different *CA* then we can have a problem. In general *CA's* form a **hierarchy**, for example it may be that the *public key* of *Alice* is certified by the *certificate* provided by *Alice's CA*, and the *certificate* provided by this *CA* is certified by *Bob's CA*. Note: certifying a *certificate* of a certain *CA* means providing the *public key* of that *CA*.



9.3.3 Certificate Revocation

We also need to **revoke certificates** for several reasons, since *certificate* are **valid** for a limited time (*CA's* want to be paid), and they can be revoked before the *deadline* if *user's secret key* is not *safe* anymore, or *user* is not certified by *CA*, or the *secret key* of the *CA* is now compromised.

In order to implement this whole feature there are the so called **certification revocation lists** (**CRL**). The *CRL* has a certain format specified again by **X.509 standard**. There is an important part of the structure in the middle, where there are the entries. Each entry has the *Certification Serial Number* (the *serial number* of the *revoked certificate*), the *Revocation Date* (when the *certificate* was removed), and the *CRL entry extensions*. The last fields is the **signature** of the **digital certificate**. There are two fields that are very important: **THISUPDATE** that represent the date of the publication of the list, and **NEXTUPDATE**, the expected date of the next *CRL* to be issued.



9.3.4 OCSP

OCSP or **Online Certificate Status Protocol**, it is a new *protocol*, an alternative to *CRL* more *agile*, used for obtaining the **revocation status** of an *X.509 digital certificate*. If the *certificate* is *Version 3*, there is an *extension* where the browser can get an *URL* and connect to it in order to make *queries*, and this is done according to this *protocol OCSP*.

9.3.5 PGP: Pretty Good Privacy

PGP is a *trust model* for *email certification*, in which there aren't any *trusted CA*, so it's like a **distributed approach**, in which each *user* acts like a *CA* and decides for himself. The *certificates* will contain *email addresses* and *public key*, and they are *signed* by *one* or *many users*, so if you *trust* a sufficient number of the *user* signing a *certificate* then you assume the *certificate* is good. Each *user* keeps info on *public keys* of other *users* and *signatures* of these *keys*, together with *trust value* of the *key*. The creator believed in a **Web of Trust**, that is something that is proving the *correctness* of the associations between a *public key* and an *identity* not by means of a *digital certificate* issued by a *CA* but by means the **trust** obtained from some other *users*.