4. Public-key techniques

Reminder - Symmetric vs Asymmetric

Encryption #encryption

- *plaintext* $\Longrightarrow$ ciphertext
- under key $k_\epsilon \in K$

Decryption #decryption

- ciphertext $\Longrightarrow$ *plaintext*
- Under key $k_d \in K$

In #symmetric cryptography: $k_E = k_D$ is the **secret key**.
In #asymmetric cryptography: k_E is public and k_D is private.

Authentication #authentication

- message $=>$ (message, **tag**)
- Under key $k_A \in K$

Verification #verification

- (message, **tag**) $=>$ message
- Under key $k_V \in K$

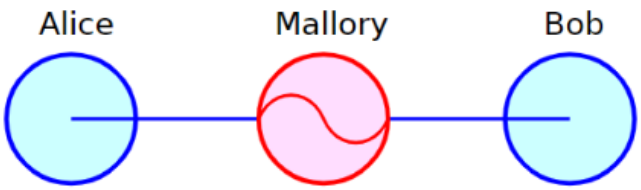
#Symmetric cryptography: $k_A = k_V$ is the secret key. The tag is called a *message authentication code* #MAC .

#Asymmetric cryptography: k_A is private and k_V is public. The tag is called a #signature .

#Symmetric is less versatile but faster. Indeed we can do much more with asymmetric.

4.1 Going public

When going public, several problems arrive. The world is not a goof place and people will try everything to go around the solutions imagined. An example of problem is the #man-in-the-middel-attack . Imagine That *Alice gets Mallory's public key*, thinking it's Bob's and *Bob gets Mallory's public key* thinking it's Alice's. Then **Mallory can decrypt and re-encrypt the traffic at will**!



[By Miraceti on Wikipedia]

The **binding of a public key to an identify** can be done through #certificates and **public key infrastructure** #PKI or via web of trust (GPG, GnuPG).

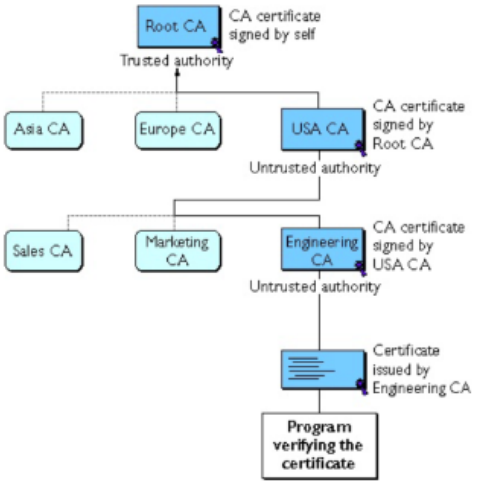
Public key certificate

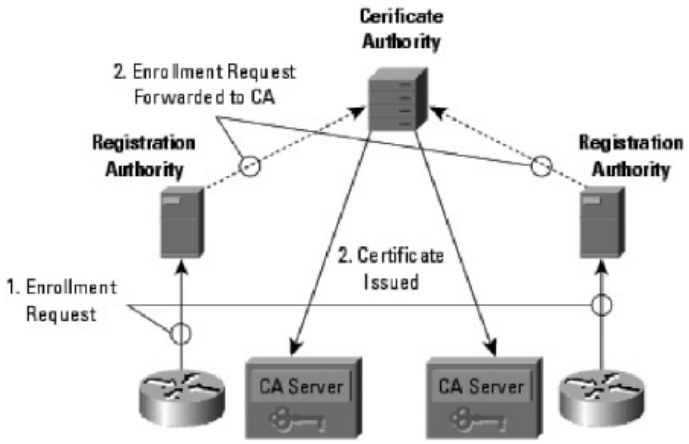
A #public-key **certificate**:

- *Allows to bind a public key with the identity of its owner*: this is good because it can prevent attacks such as #man-in-the-middel-attack .

- *Contains at least the public key*, the information *allowing to identify its owner* and a *digital signature on the key* and the information.
- Is signed by a **certification authority**.
  Example Slide 7-8.

Certificate hierarchy





Web of trust

Manual key verification over an authentic channel

- compare the fingerprint (hash function)
- if correct, sign it
  Public keys are distributed with their signatures.

A key is trusted

- if the key is signed by me, or
- if the key is signed by someone I trust.

Different public-key algorithms

There are multiple #public-key algorithms depending on the problem on which they are based:

- Factorisation
- #discrete-logarithm-problem : **modular exponentiation** or **elliptic curves** (see 4.4).
- Many more...

In practice: Hybrid encryption

In **practice**, to improve both **efficiency and bandwidth**, we can combine #asymmetric with #symmetric #encryption.

Alice computes the encryption c of the message m intended for Bob in the following way:

1. Alice chooses randomly the symmetric key k;
2. Alice computes $(c_k, c_m) = \left(\text{Enc}_{\text{Bob's public key}}(k), \text{Enc}_k(m)\right)$.

Bob decrypts as follows:

1. Bob recovers $k = \text{Dec}_{\text{Bob's private key}}(c_k)$
2. Bob recovers $m = \text{Dec}_k(c_m)$

Alice sends a **secret key** to Bob encrypted with **his public key**, and she **encrypts the plaintext with her secret key**.

4.2 Mathematical background

4.2.1 Primes

There are **infinitely many** #primes

- Suppose that $p_1 = 2 < p_2 = 3 < \ldots < p_r$ are all the existing primes.
- Let $P = p_1 p_2 \ldots p_r + 1 \implies P$ **cannot be a prime** and then let $p \neq 1$ be one of the existing primes that divides P.
- But p cannot be any of $p_1, p_2, \ldots, p_r$, because otherwise p would divide the difference $P - p_1 p_2 \ldots p_r = 1$, and p would be equal 1.
  $\rightarrow$ So this prime p is still another prime, and $p_1, p_2, \ldots, p_r$ would not be all of the existing prime.

4.2.2 Groups

A #group is a **set** G along with a **binary operation** $\circ$ that *satisfy the following **properties***:

- #closure : $\forall g, h \in G, g \circ h \in G$
- #associativity : $\forall g_1, g_2, g_3 \in G, (g_1 \circ g_2) \circ g_3 = g_1 \circ (g_2 \circ g_3)$
- #identity (neutre): there exists an identity $e \in G$ such that $\forall g \in G, e \circ g = g \circ e = g$
- #inverse : $\forall g \in G$, there exists $h \in G$ such that $g \circ h = h \circ g = e$
  $\rightarrow$ In easier terms, it is a set in which we can combine elements in a precise way.

Some **examples** of group:

- $(\mathbb{Z}_n, +) \rightarrow$ the set $\mathbb{Z}_n = \{0, 1, 2, \ldots, n-1\}$
  The law is *addition mod n*
  The neutral/identity is 0
  The inverse of x is $-x \bmod n = n - x$
- $(\mathbb{Z}_p^*, \times) \rightarrow$ the set $\mathbb{Z}_p^* = \{0, 1, 2, \ldots, p-1\}$
  The law is *multiplication mod p*
  Its neutral is 1
  The inverse of x is $x^{-1} \bmod p$
- $(\{0,1\}, \oplus)$ is a group $= (\mathbb{Z}_2^*, +)$
- $(\mathbb{N}, +)$ is **not a group**

Order of a group

Let G be a group, the #order of G, denoted $|G|$ is **the number of elements in** G.
$\rightarrow |\mathbb{Z}_n^*| = n$ and $|\mathbb{Z}_p^*| = p - 1$

Exponentiation

Let $(G, \circ)$ be a group, we define the #exponentiation of an element g as

$$[x]g = g \circ \cdots \circ g$$

with x times g.
Example: In $(\mathbb{Z}_5^*, +) \rightarrow 3[g] = g + g + g$
With $g \in \{1, 2, 3, 4\}$

Order of an element in a finite group

The **order of an element a of a group** G is the smallest positive integer $m = ord(a)$ such that $[m]a = e$, where e denotes the **identity** element of the group.
Example: Let $G = \mathbb{Z}_{10}$ and $a = 2$. The condition is $m \circ a = 0 \bmod 10$. Thus $m = 5 = ord(2)$

#Theorem : **For any a its order $ord(a)$ divides the size $|G|$ of the group**.

An element g is said to be #generator of G if $ord(g) = |G|$. Thus (by using the theorem), it can be used to generate all the elements of the group.
Example: In $G = (\mathbb{Z}_5^*, +)$, the generator can be 2 as we can generate all elements of the group with it:
- $1[2] = 2$
- $2[2] = 4$
- $3[2] = 2 + 2 + 2 = 6 \bmod 5 = 1$
- $4[2] = 2 + 2 + 2 + 2 = 8 \bmod 5 = 3$

$\rightarrow$ All numbers of $\mathbb{Z}_5^*$ are generated by 2. So 2 is a #generator .

4.2.3 Operations

4.2.3.1 Modulo

A #modulo is :

- A **binary operation**: $a \bmod n = r$ iff $a = qn + r$ and $0 \leq r < n$
- An **equivalence relation**: $a \equiv b (\bmod n)$
  - iff there exists an integer k such that $(a - b) = kn$
  - iff $(a \bmod n) = (b \bmod n)$

Modular additions and subtractions

$\mathbb{Z}_n, +$ is the set $\{0, 1, \ldots, n-1\}$ together with addition modulo n.
$\mathbb{Z}_n, +$ is a #group with

- #identity is 0
- #inverse of x is $-x \bmod n$

1 is a #generator of the group because all the elements can be represented by adding 1 to itself an appropriate number of times.

4.2.3.2 Greatest common divisor

#gcd or **greatest common divisor** is noted $gcd(a; b)$

Bézout: For any positive integers a and b, there exist integers x and y such that $ax + by = \gcd(a, b)$.

Let $S = \{ax + by : x, y \in \mathbb{Z}\}$ and $d = \min(S \cap \mathbb{N}_{>0})$
Divide a by d: $a = qd + r$ with $0 \leq r < d$
But $r = a - qd = a - q(ax + by) = (1 - qx)a - (qy)b$ so $r \in S$ and $r < d$ hence $r = 0$ and d divides a
Similarly, d divides b and therefore d divides $\gcd(a, b)$
However $\gcd(a, b)$ divides all elements of S, so $d = \gcd(a, b)$

#corollary : a and b are relatively #primes (or equivalently $gcd(a, b) = 1$) *iff there exist integers x and y such that $ax + by = 1$*

Modular multiplications and multiplicative inversion

$\mathbb{Z}_n^*, \times$ is the set $\{x : 0 < x < n \text{ and } gcd(x, n) = 1\}$ together with multiplication #modulo n.

$\mathbb{Z}_n^*, \times$ is a #group with

- #identity is 1
- #inverse of x is denoted $x^{-1} \bmod n$ and is such that $x^{-1}x = 1 (\text{mod } n)$

The inverse x^{-1} mod n exists and is unique iff $\gcd(x, n) = 1$

- $xy \equiv 1 \pmod{n}$ means that $xy = 1 + kn$ for some integer k, so $xy + kn = 1$ and $\gcd(x, n) = 1$
- $\gcd(x, n) = 1$ implies that there exist integers y and z such that $yx + zn = 1$, so $xy = 1 - zn$ and $xy \equiv 1 \pmod{n}$

It is possible to show that the **inverse is unique** under certain conditions (if the number is coprime with the n).

4.2.4 Euler's $\Phi(n)$ function

We note $\Phi(n)$ the **number of integers smaller than n and that are relatively prime/coprime with n**. So $|\mathbb{Z}_n^*| = \Phi(n)$.
If $n = \Pi_{i=1}^{r} p_i^{e_i}$ for distinct #primes $p_1, p_2, \ldots, p_r$, then

$$\Phi(n) = n * \Pi_{i=1}^{r} \left(1 - \frac{1}{p_i}\right)$$

Some particular cases
- Prime p: $\Phi(p) = p - 1$
- Product of two distinct primes p and q : $\Phi(pq) = (p - 1)(q - 1)$

#Fermats-little-theorem : let p be a prime and a an integer not a multiple of p, then $a^{p-1} = 1 (\text{mod } p)$

#Eulers-theorem : Let a and n two relatively prime integers, then $a^{\Phi(n)} \bmod n = 1 \bmod n$

As a consequence: the **exponent can be reduced** #modulo $\Phi(n)$:

$$a^e = a^{e \bmod \Phi(n)} (\text{mod } n)$$

4.2.5 Generator in $\mathbb{Z}_n^*$

The group $\mathbb{Z}_n^*$ has size $\Phi(n)$, hence a #generator g is an element with $ord(g) = \Phi(n)$. Such a generator exists when n is $2, 4, p^a$ or $2p^a$, with p an odd prime.

Example: consider $\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$.

- $\text{ord}(1) = 1 : 1^1 = 1$
- $\text{ord}(2) = 3 : 2^1 = 2, 2^2 = 4, 2^3 = 1$
- $\text{ord}(\mathbf{3}) = 6 : 3^1 = 3, 3^2 = 2, 3^3 = 6, 3^4 = 4, 3^5 = 5, 3^6 = 1$
- $\text{ord}(4) = 3 : 4^1 = 4, 4^2 = 2, 4^3 = 1$
- $\text{ord}(\mathbf{5}) = 6 : 5^1 = 5, 5^2 = 4, 5^3 = 6, 5^4 = 2, 5^5 = 3, 5^6 = 1$
- $\text{ord}(6) = 2 : 6^1 = 6, 6^2 = 1$

4.2.6 Chinese remainder theorem #CRT

The #CRT or **Chinese remainder theorem**:
Let $\{m_1, \ldots, m_k\}$ be a set of relatively prime integers, i.e., $\forall 1 \leq i \neq j \leq k, gcd(m_i, m_j) = 1$ and let m be their product $m = m_1 \times \cdots \times m_k$.
Then the following system of equations:

$$\begin{cases} x \equiv a_1 \pmod{m_1} \\ \vdots \\ x \equiv a_k \pmod{m_k} \end{cases}$$

has one and only one solution #modulo m.

To **solve a** #CRT **problem**:
Compute $M_i = \frac{m}{m_i}$, and notice that
- $\gcd(M_i, m_i) = 1$.
- $M_i \equiv 0 \pmod{m_j}$ for any $j \neq i$.

Compute $c_i = M_i^{-1} \pmod{m_i}$.
Let $x = \sum_{i=1}^{k} a_i c_i M_i$.

Indeed, $x \equiv a_i c_i M_i \equiv a_i \pmod{m_i}$.

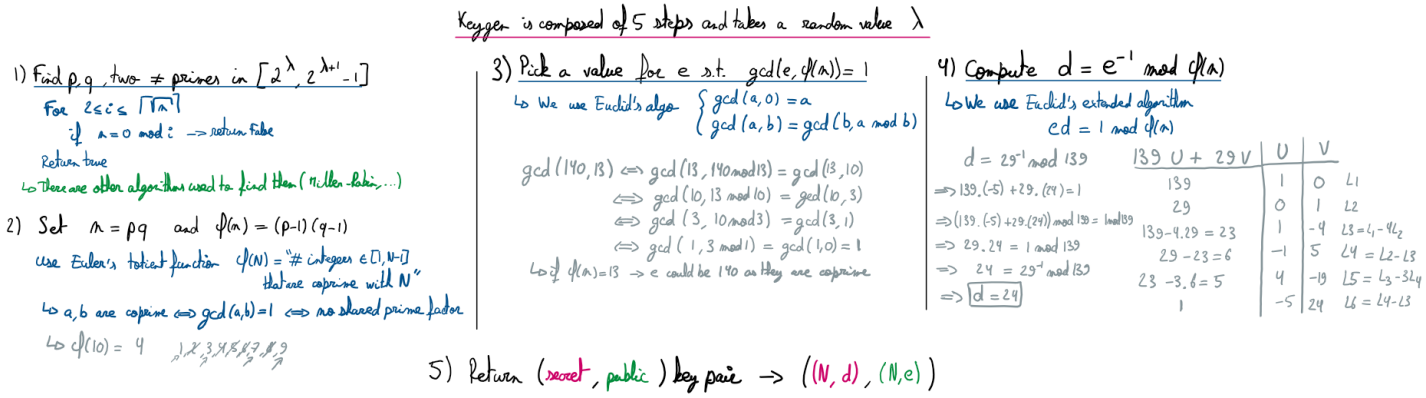
4.3 Rivest-Shamir-Adleman #RSA

4.3.1 RSA key generation

In #RSA #key-generation, the user **generates a public-private** key pair as follows:

- **Privately** generate two large distinct primes p and q
- Choose a **public** exponent $3 \leq e \leq (p-1)(q-1) - 3$
  - It must satisfy $gcd(e, (p-1)(q-1)) = 1$
  - often, one chooses $e \in \{2, 17, 2^{16} + 1\}$ then generates the primes
- Compute the **private** exponent $d = e^{-1} \bmod (p-1)(q-1)$
- Compute the **public modulus** $n = pq$ and discard p and q

The #public-key is: (n, e).
The #private-key is (n, d).



4.3.2 RSA textbook encryption - DONT USE

#RSA From **plaintext** $m \in \mathbb{Z}_n$ to **ciphertext** $c \in \mathbb{Z}_n$ and back:

- #encryption $Enc(m,(n,e)) \to c = m^e \bmod n$, return c
- #decryption $Dec(c,(n,d)) \to m' = c^d \bmod n$, return m'

Verification

Thanks to #Eulers-theorem, the fact that p and q are distinct #primes and by definition of e and d:

$$
\begin{aligned}
c^d &\equiv (m^e)^d \equiv m^{ed} \\
&\equiv m^{ed \bmod \Phi(n)} && \text{by Euler's theorem} \\
&\equiv m^{ed \bmod (p-1)(q-1)} && \text{since } p \text{ and } q \text{ are distinct primes} \\
&\equiv m^1 && \text{by definition of } e \text{ and } d \\
&\equiv m \pmod n
\end{aligned}
$$

RSA textbook signature

#RSA From **plaintext** $m \in \mathbb{Z}_n$ to **signatures** $s \in \mathbb{Z}_n$ and back:

- #signature Sends (m,s), with $s = m^d \bmod n$
- #verification Check whether $m = s^e \bmod n$

4.3.3 RSA Attacks

4.3.3.1 RSA and factorisation

#factorisation is an #attack on #RSA, and RSA textbook is not really resistant to it:
If one can factor n into $p \times q$, the *private exponent* d **follows immediately**.

Conversely, **from the knowledge of** d, it is easy to factor n.

$\Longrightarrow$ This is one reason why we should NOT use textbook #RSA.

Proof: $ed \equiv 1 \pmod{\Phi(n)}$, so for any $a \in \mathbb{Z}_n^*$ we have $a^{ed-1} \equiv 1 \pmod{n}$. As $\Phi(n)$ is even, so is $ed - 1 = 2t$ and $a^{2t} \equiv 1 \pmod{n}$.

Define $z = a^t \bmod n$, so that $z^2 \equiv 1 \pmod{n}$. Assume $z \bmod n \neq \pm 1$ (otherwise change a). Hence n divides $z^2 - 1 = (z-1)(z+1)$.

Let $g_1 = \gcd(z-1, n)$ and $g_2 = \gcd(z+1, n)$. $g_1 \neq n \neq g_2$, as we assumed $z \bmod n \neq \pm 1$. Both g_1 and g_2 cannot be equal to 1 since $z^2 - 1$ is a multiple of n.

Hence g_1 or g_2 is p or q.

There is **no known polynomial time algorithm to factor an integer** (polynomial in the size of the integer).

But **there exist sub exponential algorithms**. The currently best known algorithm (general number field sieve) factors an integer n asymptotically in time:

$$\exp\left(\left(\sqrt[3]{\frac{64}{9}} + o(1)\right)(\ln n)^{\frac{1}{3}}(\ln \ln n)^{\frac{2}{3}}\right).$$

For 128-bit security, NIST recommends n to be at least 3072-bit long (hence p and q are at least 1536-bit long).
[NIST SP 800-57, see also https://www.keylength.com/]

4.3.3.2 Other attacks on RSA

There are other #attack such as **cyclic attack**, **message factorisation** or **short message attack**.

#short-message-attack consists in the following:
If e is small (e.g., $e = 3$), and if $m < n^{1/e}$ then $m^e \bmod n = m^e$ *without any modular reduction*. Therefore m **is retrieved** by simply computing $c^{1/e}$ over the integers.

Example: with e=3 and n on 3072 bits, the attack works until m is a 1024-bit integer.

4.3.4 RSA for confidentiality

4.3.4.1 RSA-OAEP

#RSA #OAEP or **Optimal Asymmetric Encryption Padding** allows to **counter** #factorisation and the #short-message-attack and to make RSA #IND-CPA-secure.

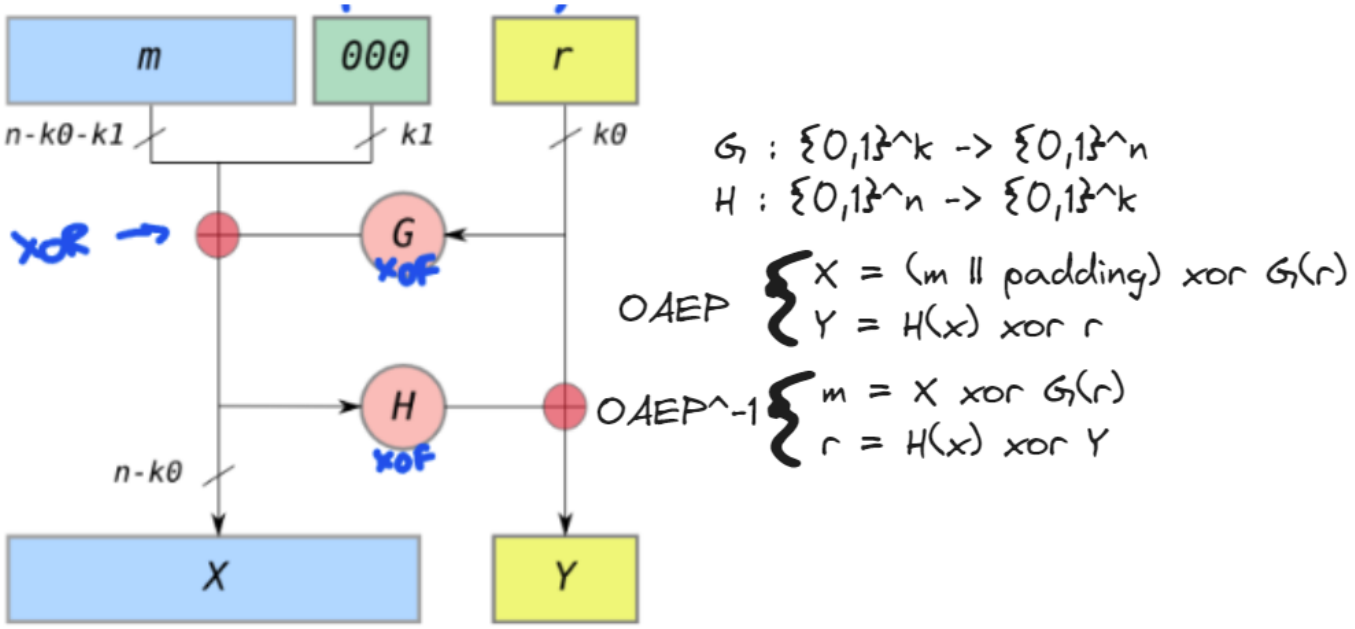
#OAEP works by **encrypting a message of small size**, **use randomness** in the encryption and then apply the output of #OAEP as #RSA input.

It allows to make the scheme **probabilistic**, **prevents** that **an adversary recovers any portion of the plaintext without being able to invert RSA** (the adversary can either have everything or nothing).

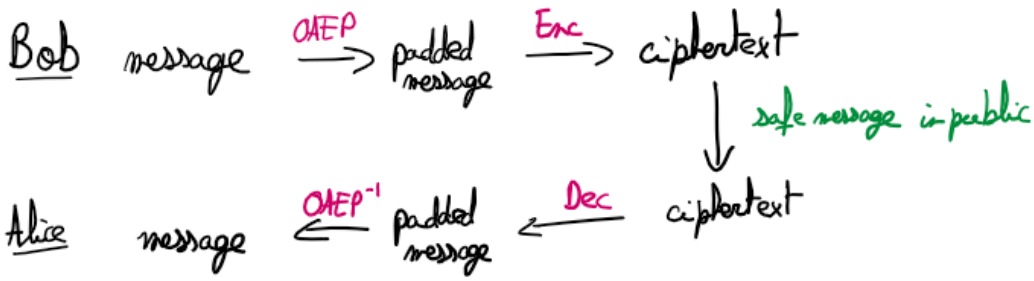
It is shown to be #IND-CPA-secure assuming that G and H behave as #random-oracle.

#encryption : $Enc(m, r, (N, e)) = (X||Y)^e \bmod n$ with $(X||Y) = OAEP(m, r)$
- Add a padding full of 0 to the message to have a fix length $m||padding$
- Message m of size n max

- #nonce r of size k
#decryption : $(X||Y) = c^d \bmod n$, then $\dots m$



Note that it is similar to #feistel-network.



It is applied like this to RSA.

4.3.4.2 RSA-KEM

#RSA-KEM
In hybrid encryption, **Alice does not have to choose the secret key**, but **she can let it be derived from some random bits** by encapsulating the symmetric key using RSA. It enables **shared key creation**.

To #encapsulate, Alice does the following

1. Alice chooses a random m of the same bit size as n
2. Alice encrypts $c = m^e \bmod n$, and she sends c to Bob
3. Alice computes $k = hash(m)$

To #decapsulate, Bob does the following

1. Bob recovers $m = c^d \bmod n$
2. Bob computes $k = hash(m)$

4.3.5 RSA for authenticity

The different options always uses #RSA but there are some modifications made on the message m that is encrypted/signed by RSA.

RSA: How NOT to sign

For #authentication, let's recall the #RSA textbook signature.

- **Signature**: Send (m, s), with $s = m^d \bmod n$
- **Verification**: Check whether $m = s^e \bmod n$

But as said earlier, the textbook signature is **not efficient** as a #forgery attack that exploit the multiplicative structure allows to crack this:

Forgery attack: (exploiting the multiplicative structure)

- Ask for the signature of m_1: $s_1 = m_1^d \bmod n$
- Ask for the signature of m_2: $s_2 = m_2^d \bmod n$
- Compute $m_3 = m_1 \times m_2 \bmod n$ and $s_3 = s_1 \times s_2 \bmod n$
- Submit (m_3, s_3) for verification:

$$\begin{aligned} m_3 &\equiv m_1 m_2 \\ &\equiv s_1^e s_2^e \\ &\equiv s_3^e \pmod n \end{aligned}$$

RSA with message recovery

If the **message** m is **short enough**, we can **embed it in the signature**.
Let $R(m) \to m'$ be a redundancy function from message space M to range $\mathcal{R} \subset \mathbb{Z}_n$.

- **Signature**: Compute $m' = R(m)$ then send $s = (m')^d \bmod n$
- **Verification**:
  - Recover $m' = s^e \bmod n$
  - If $m' \in \mathcal{R}$, return $m = R^{-1}(m')$ and accept it.
  - Otherwise, reject it.

But this is not used much in practice.

RSA with full-domain hashing

Let H be an extendable output function (or take an old-style hash function and use MGF1).

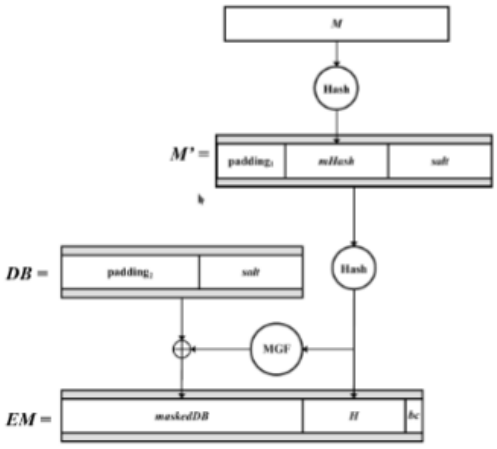
- **Signature**:
  - Compute $h = H(m)$ so that h has the same bit size as n
  - Send (m, s), with $s = h^d \bmod n$
- **Verification**:
  - Compute $h = H(m)$ like above
  - Check whether $h = s^e \bmod n$

It is also not used a lot but it is real nice and easy to achieve.

RSA with probabilistic signature scheme

#RSA-PSS is a #signature scheme used with #RSA, it utilises a probabilistic padding scheme to provide a better resistance against certain types of attacks.

It allows the use of variable *salt* length, and it is very much used (standardised for PKCS #1 v2.1).



[PKCS #1 v2.2] [Bellare and Rogaway]

Signature:

- From m and random *salt*, compute *EM*
- Send (m, s), with $s = (EM)^d \bmod n$

Verification:

- Recover $EM = s^e \bmod n$
- Recover *salt*
- Check $H \stackrel{?}{=} \text{hash}(M')$

4.3.6 RSA implementation

This section explains how to compute the RSA (manually or with a computer).

How to find large primes

#RSA needs **large primes** to be efficient. Indeed, the security of RSA relies on the difficulty of factoring the product of two large prime numbers.

The number of primes up to n is given by $\pi(n) \sim \frac{n}{\ln n}$ for large n.

$\rightarrow$The average prime gap for numbers of b bits is about $b \ln 2$.

The **recipe**:

- Draw a random number n
- Test co-primality with the first few primes $2, 3, 5, \ldots$
- Test **pseudo**-primality with (e.g. Miller-Rabin)
- Otherwise, increment n and repeat
  This method has been done and used in the exercise sessions.

Exponentiation using square and multiply

To compute $a^e \bmod n$, write the exponent in binary and apply the **square and multiply** *algorithm*. Reduce the numbers modulo n as you compute.
Here is an example for $e = 26 = 11010_2$

a	initialization 1
$a \rightarrow a^2 \times a = a^3$	square and multiply 1
$a^3 \rightarrow (a^3)^2 = a^6$	square 0
$a^6 \rightarrow (a^6)^2 \times a = a^{13}$	square and multiply 1
$a^{13} \rightarrow (a^{13})^2 = a^{26}$	square 0

The 1's $\rightarrow$ square and multiply
The 0's $\rightarrow$ just square.

Optimisation using the CRT

To speed up the decryption or signature generation, keep p and q and use the #CRT Chinese Remainder Theorem. It allows computing to a smaller power, and thus faster.

Instead of computing $m = c^d \bmod n$, compute:

- $m_p = c^d \bmod p = c^{d \bmod (p-1)} \bmod p$
- $m_q = c^d \bmod q = c^{d \bmod (q-1)} \bmod q$

Then recombine:

$$m = (m_p - m_q)(p^{-1} \bmod q)p + m_p$$

4.4 Discrete logarithm problem in $\mathbb{Z}_p^*$

#discrete-logarithm-problem #DLP

The #DLP is a problem that has currently **no known polynomial time algorithm to solve it**. It is therefore used for encryption and signatures as we will see here bellow.

First, let's **fix two domain parameters**:

- Let p be a *large* #prime
- Let g be a #generator of $\mathbb{Z}_p^*$ (for example: $\{g^i \bmod p : i \in \mathbb{N}\} = \mathbb{Z}_p^*$)

> **Discrete logarithm problem (DLP)**
>
> Given $A = g^a \bmod p$, find a.

For 128-bit security, NIST recommends p to be at least 3072-bit long.

4.4.1 Key generation

#key-generation #DLP

The **domain parameters**

- Let p be a *large* #prime
- Let g be a #generator of $\mathbb{Z}_p^*$ (for example: $\{g^i \bmod p : i \in \mathbb{N}\} = \mathbb{Z}_p^*$)

The user generates a #public #private #key-pair as follows:

- The **private key** a is made by privately choose a random integer $a \in [1, p-2]$
- Compute the **public key** $A = g^a \bmod p$

In group notation

The **domain parameters**

- Let G be #group
- Let $g \in G$ be a #generator of G of order $q = |G|$

The user generates a #public #private #key-pair as follows:

- The **private key** a is made by privately choose a random integer $a \in [1, q-1]$
- Compute the **public key** $A = [a]g$

The **public key** is A and the **private key** is a.

4.4.2 El Gamal encryption

Taher ElGamal

The objective of #ElGamal is to use the #DLP to secure its methods.

#encryption of $m \in \mathbb{Z}_p^*$ with Alice's public key A.

- Choose randomly an integer $k \in [1, p-2]$
- Compute:
  - $K = g^k \bmod p$
  - $c = mA^k \bmod p$

#decryption of (K, c) by Alice with her private key $a \rightarrow m = K^{-a}c \bmod p$

A, B and K are all *public*, the **rest** is *private*.

This scheme is #correct because we can recover m by applying encryption and decryption when we fix the parameters.

$$K^{-a} \equiv (g^k)^{-a} \equiv (g^a)^{-k} \equiv (A^k)^{-1} \pmod{p}$$
$$(A^k)^{-1}c \equiv (A^k)^{-1}m(A^k) \equiv m \pmod{p}$$

4.4.2.1 Security of El Gamal encryption

(K, k) can be seen as an #ephemeral #key-pair, created by the sender.

- K is part of the ciphertext;
- k is protected by the #DLP.
  In order to have a **secure ElGamal** encryption, k must be **secret** and **randomly drawn independently at each encryption**!
  Indeed, if
- k is known, one can compute A^k and recover m from c
- k is repeated to encrypt, say m_1 and m_2 then we have

  $$c_1 = m_1A^k \bmod p \text{ and } c_2 = m_2A^k \bmod p$$

  And thus

  $$c_1c_2^{-1} = m_1m_2^{-1} \pmod{\text{p}}$$

The Diffie-Hellman problem

The Diffie-Hellman Problem #DHP is easier than the #DLP (breaking #DHP does not give us the exponents). However, there are currently **no known polynomial time algorithm** to solve the #DHP either.

First, let's **fix two domain parameters**:

- Let p be a *large* #prime
- Let g be a #generator of $\mathbb{Z}_p^*$ (for example: $\{g^i \bmod p : i \in \mathbb{N}\} = \mathbb{Z}_p^*$)

> **Diffie-Hellman problem**
>
> Given $X = g^x \bmod p$ and $Y = g^y \bmod p$, find $X^y = Y^x = g^{xy} \bmod p$.

To **recover** k or a from K or A $\implies$ #DLP

But to break ElGamal, it is sufficient to recover $K^a = A^k = g^{ak} \bmod p$. Hence, #ElGamal #encryption relies on the #DHP.

4.4.3 Diffie-Hellman key agreement

#Diffie-Hellman
This is **hybrid encryption**, we *do not need to choose the secret key*.
First, let's **fix two domain parameters** p and g and:

- Alice's key pair $A = g^a \bmod p$
- Bob's key pair $B = g^b \bmod p$

Alice computes	Bob computes
$K_{AB} = B^a \bmod p$	$K_{AB} = A^b \bmod p$
$k_{AB} = \text{hash}(K_{AB})$	$k_{AB} = \text{hash}(K_{AB})$

$\leftrightarrow$

Secure channel using k_{AB} in a symmetric cipher

This is #correct because: $A^b = g^{ab} = B^a \pmod{p}$.

Note that the hashing is to have the same size and to avoid it to be uniformly distributed.

In group notation

#Diffie-Hellman in #group notation.
First, let's **fix two domain parameters**: a #group G and a #generator $g \in G$

- Alice's key pair $A = [a]g$
- Bob's key pair $B = [b]g$

Alice computes	Bob computes
$K_{AB} = [a]B$	$K_{AB} = [b]A$
$k_{AB} = \text{hash}(K_{AB})$	$k_{AB} = \text{hash}(K_{AB})$

$\leftrightarrow$

Secure channel using k_{AB} in a symmetric cipher

This is #correct because $[b]A = [ab]g = [a]B$

The **problem** with this is that **when they communicate once**, both **Alice and Bob have the same secret key**. Therefore, a more secure encryption scheme would counter that.

Ephemeral Diffie-Hellman key agreement

#ephemeral #Diffie-Hellman key agreement has for goal to **avoid using the same long-term keys** for all communications.

First, let's **fix two domain parameters** p and g and:

- Alice's key pair $A = g^a \bmod p$
- Bob's key pair $B = g^b \bmod p$

Alice		Bob
randomly chooses e		randomly chooses f
sends $E = g^e \bmod p$	$\leftrightarrow$	sends $F = g^f \bmod p$
along with $\text{sign}_a(E)$	$\leftrightarrow$	along with $\text{sign}_b(F)$
checks Bob's signature with B		checks Alice's signature with A
$K_{AB} = F^e \bmod p$		$K_{AB} = E^f \bmod p$
$k_{AB} = \text{hash}(K_{AB})$		$k_{AB} = \text{hash}(K_{AB})$
	$\leftrightarrow$	
Secure channel using k_{AB} in a symmetric cipher		

> Note that the signature exists to make sure that the keys E and F that are sent (public) comes from Alice and Bob. Otherwise anyone could send a public key and this would ruin the key agreement.

4.4.4 El Gamal signature

The #ElGamal #signature of message $m \in \mathbb{Z}_2^*$ by Alice with her private key a:

- Compute $h = hash(m)$
- Choose randomly an integer $k \in [1, p-2]$
- Compute $r = g^k \bmod p$
- Compute $s = k^{-1}(h - ar) \bmod (p-1)$
  - If $s = 0$, restart with a new k
- Send (r, s) along with m.

The #verification of signature (r, s) on m with Alice's public key A.

- Compute $h = hash(m)$
- Check whether $A^r r^s = g^h \pmod{\text{p}}$

This is #correct because:

$$\begin{aligned} A^r r^s &\equiv (g^a)^r (g^k)^s \\ &\equiv g^{ar+ks \bmod (p-1)} \\ &\equiv g^{ar+kk^{-1}(h-ar) \bmod (p-1)} \\ &\equiv g^{ar+h-ar \bmod (p-1)} \\ &\equiv g^h \pmod p \end{aligned}$$

This is done thanks to #Eulers-theorem.

Security of ElGamal signature

r and k can be seen as an #ephemeral #key-pair, created by the signer.

- r is part of the signature
- k is protected by the #DLP

But we must be careful because k **must be secret and randomly drawn independently at each encryption** (Even more than with ElGamal Encryption):

- If k is known, one can recover a from s.
- If k is repeated to sign messages with hashes $h_1 \neq h_2$ then

$$s_1 - s_2 = k^{-1}(h_1 - ar) - k^{-1}(h_2 - ar) = k^{-1}(h_1 - h_2) \bmod (\text{p-1})$$

  and we can recover k, then a from s_1 or s_2!

A famous case is SONY that used the same k to sign all their games, so one user recovered k and K and sold a lot of games with it.

4.4.5 Schnorr signature

#Schnorr #signature of a message $m \in \mathbb{Z}_2^*$ by Alice with her private key a:

- Choose randomly an integer $k \in [1, p-2]$
- Compute $r = g^k \bmod p$
- Compute $e = hash(r||m)$
- Compute $s = k - ea \bmod (p-1)$
- Send (s, e) along with m.

#verification of signature (s, e) on m with Alice's public key A:

- Compute $r' = g^s A^e \bmod p$
- Compute $e' = hash(r'||m)$
- Check whether $e' = e$

It is different than #ElGamal as here, e is computed with m and r.

4.5 Security of the discrete logarithm problem

Discrete logarithm problem (DLP) in $\mathbb{Z}_p^*$

Given $A = g^a \bmod p$, find a.

The #DLP is not easy to solve but let's take a look at how to solve it.

Solving DLP generically: baby-step giant-step

Let $N = |\mathbb{Z}_p^*|$ and g be a generator of $\mathbb{Z}_p^*$.

- Let $m \approx \sqrt{N}$ and suppose that $a = a_0 + a_1 m$ with $a_0, a_1 < m$
- $A \equiv g^{a_0 + a_1 m} \pmod{p} \quad \Leftrightarrow \quad A g^{-a_1 m} \equiv g^{a_0} \pmod{p}$
- For $i = 0$ to $m - 1$ sequentially (baby steps)
  - Compute and store (i, g^i), i.e., multiply by g at each step
- For $j = 0$ to $m - 1$ sequentially (giant steps)
  - Compute $A g^{-jm}$, i.e., multiply by g^{-m} at each step
  - If $A g^{-jm} = g^i$ then $a = i + jm$ and **exit**

The **baby steps** is to compute and store all exponential values of g.

The **giant steps** is to compute Ag^{-jm} and compare it with the previously stored values.

$\Longrightarrow$ This takes a complexity of $O(\sqrt{N})$ in **time and memory**

In group notation

Discrete logarithm problem (DLP) for any group G

Given $A = [a]g$, find a.

Let $N = |G|$ and g be a generator of G.

- Let $m \approx \sqrt{N}$ and suppose that $a = a_0 + a_1 m$ with $a_0, a_1 < m$
- $A = [a_0 + a_1 m]g \quad \Leftrightarrow \quad A \circ [-a_1 m]g = [a_0]g$
- For $i = 0$ to $m - 1$ sequentially (baby steps)
  - Compute and store $(i, [i]g)$, i.e., apply $\circ g$ at each step
- For $j = 0$ to $m - 1$ sequentially (giant steps)
  - Compute $A \circ [-jm]g$, i.e., apply $\circ [-m]g$ at each step
  - If $A \circ [-jm]g = [i]g$ then $a = i + jm$ and **exit**

$\Longrightarrow$ This takes a complexity of $O(\sqrt{N})$ in **time and memory**

Pohlig-Hellman

Let the size of the group be decomposed in prime factors:

$$N = |G| = \Pi_i p_i^{e_i}$$

Then we can solve the #DLP in time

$$O\left(\Sigma_i e_i (\log N + \sqrt{p_i})\right)$$

Conclusions

In conclusion, for the #DLP to be hard to break. The following conditions are **necessary** (but **not sufficient**):

- The **size of the group should be** $\approx 2^{2s}$ for security strength s.
- The size of the group should be prime.

Note: we focus on the actual group used. In the case of DSA, it is the sub-group of size q generated by f.

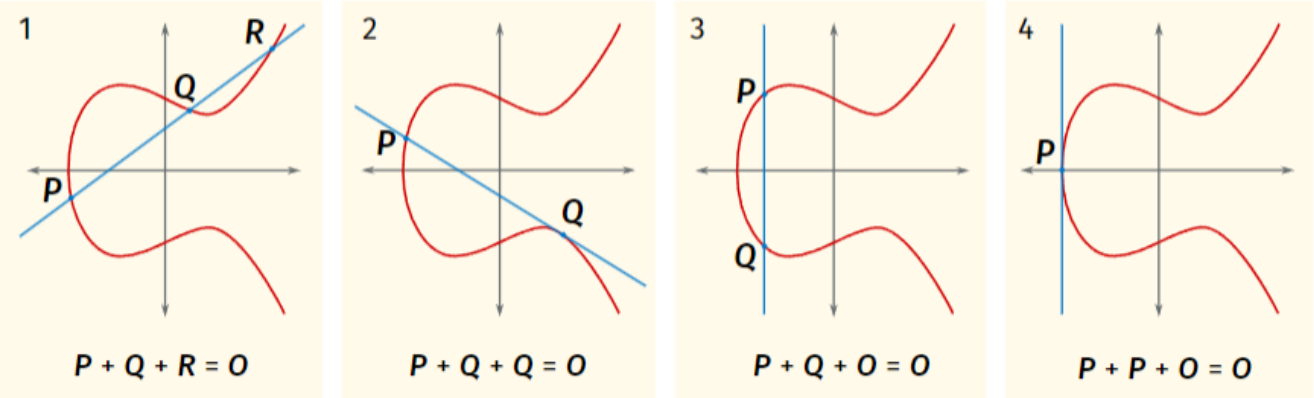
4.6 Elliptic curve cryptography

Given constants a and b, an #elliptic-curve #EC is the **set of points** $(x, y) \in \mathbb{R}^2$ that **satisfy the Weierstrass equation**:

$$y^2 = x^3 + ax + b$$

together with the **point at infinity** denoted O (one can view the point at infinity as $()0, \pm\infty)$).

An important #property of the #EC :
If a straight line meets an elliptic curve in two points, then it must cross a third point.
A point on a tangent counts for 2
Don't forget O.



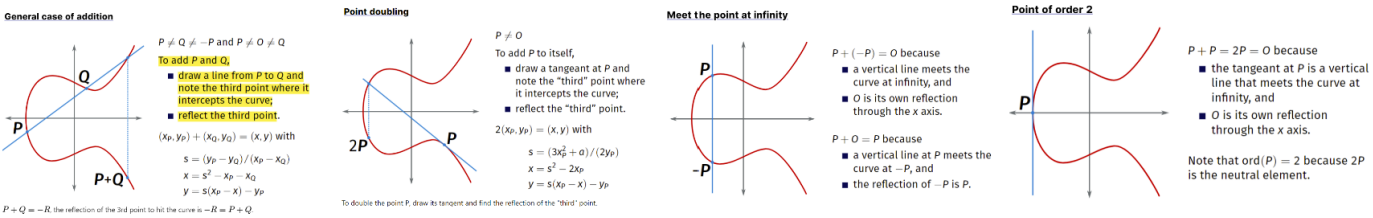
4.6.1 Group law

In #group law, let's build a **rule to add points on the curve**.

- Three aligned points must sum to O
- The point at infinity O is the neutral element.

So:

- If P and P' are each other's reflections over the x axis, then P, P', O are aligned. So $P + P\prime + O = O$ and $-P = P'$.
- If P, Q, R are aligned, then $P + Q + R = O$ and $P + Q = -R$.



4.6.2 Elliptic curves over (prime) finite fields

Fix a #prime $p \geq 3$ and constants a and b. An #EC is the set of points $(x, y) \in \mathbb{Z}_p^2$ that satisfy the Weierstrass equation

$$y^2 = x^3 + ax + b$$

together with the point at infinity denoted O.

Fact: the *formulas for point addition and doubling also work here*.
We can thus define a finite group comprising the points on the curve (including O, the neutral element), together with the point addition as operation.

Number of points in a curve over a finite field

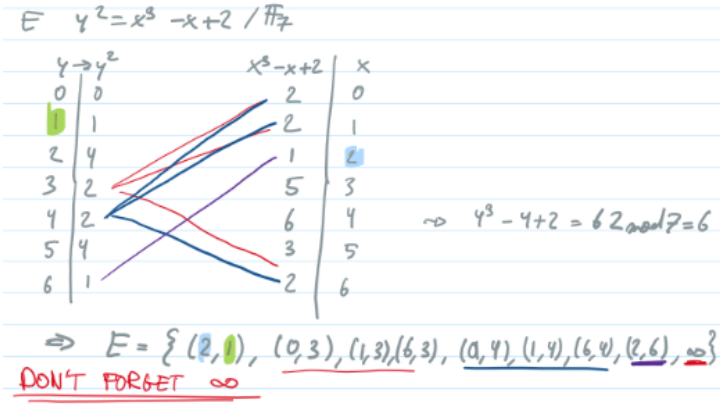
The **number of points on a curve** E is denoted $\#E$. This number depends on the parameters a and b.

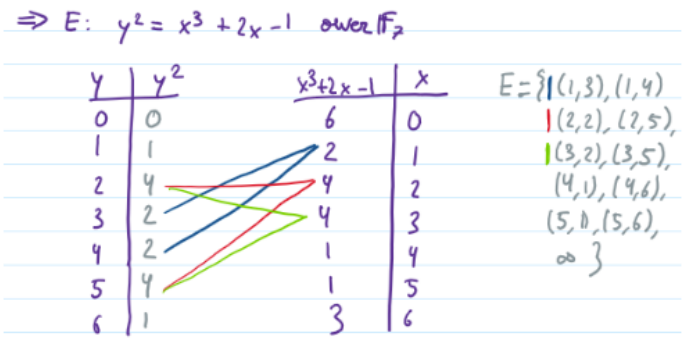
Hasse's theorem

In GF(p), the number of points of an elliptic curve E satisfies

$$p + 1 - 2\sqrt{p} \leq \#E \leq p + 1 + 2\sqrt{p}.$$

To find points in a curve over a finite field





4.7 Protocols

#skip 4-Public

That's it folks!