Chapter 1

Public-key cryptography

1.1 Going public

1.1.1 Symmetric crypto vs public-key crypto

Symmetric cryptography is the oldest kind of cryptography. Indeed, the idea of having a public key is quite disturbing at first glance. Just a quick reminder, here is how we settle a supposed secure way of communication between Ali and Bachar using public-key crypto.

1. Ali and Bachar both generate a key-pair : they keep one private for each, and they publish the other, making it public, so anyone has access to Ali and Bachar's public key.
2. If Bachar wants to send a message to Ali, he **encrypts with Ali's public key** and sends him
3. Ali receives messages that are encrypted using his public key. He decrypts them using his private key.

That's the kind of communication we have for public-key crypto. Symmetric crypto was based on the idea of having **only one key**, kept secret, between A and B.

1.1.2 Why going public ?

There are many problems to symmetric crypto :

- Key distribution problem : the key must be established between Ali and Bachar. How do they do ? The communication link between them is not secure.
- Number of keys : if there are n users, there are $n(n-1)/2$ keys. That's a lot ! 4 million keys for 2000 people.
- No protection against cheating : as Ali has the same secret key as Bachar, he can create a message from scratch and claim that Bachar created it.

In addition to all this, let me talk a little bit, because you guys talk too much,  . Symmetric crypto algorithms are meant to avoid any compact mathematical description. That is easy shit. Asymmetric crypto is **built on one-way number-theoric functions so there is no bullshitting in their security**, you get me ?

1.1.3 Actually going public

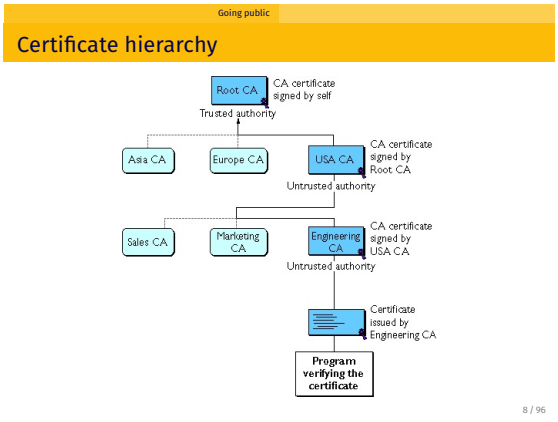
When going public, we are immediately subject to a particular kind of attack : the **Man-In-the-Middle attack** (MIM[1]). In public-key crypto, a crucial part is the exchange of keys : Ali gives his public key to Bachar, so that Bachar can send him messages, and inversely. But there can be an outsider Omar that acts as follows :

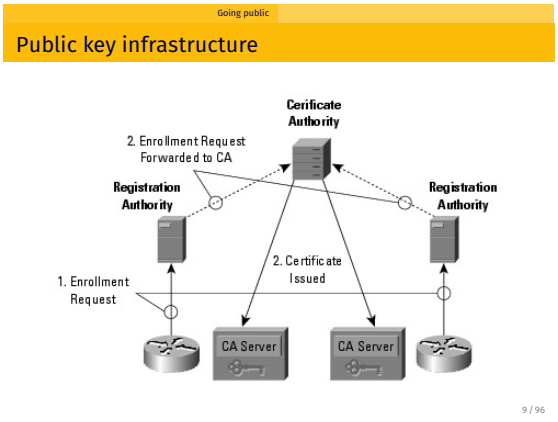
- He captures Ali's public key k_A before Bachar gets it
- He sends to Bachar a false public key, being his own, k_O
- Bachar sends his public key to Ali but it gets intercepted by Omar, that gives to Ali his k_O instead.
- Ali thinks he is receiving messages from Bachar, and inversely, but it is actually the no-life Omar that is intercepting everything and manipulating the conversation.

To avoid this, we make use of **certificates**, objects that bind a public key to an identity. A public key certificate is an object that contains :

- The public key
- Information allowing to identify its owner, such as the name, the ID, IP, etc.
- A digital signature on the key and the information : this is signed by a **certification authority**.

Thanks to this, at the end, Omar can not generate a fake certificate using his key and Ali's information because the signature will be different than Ali's public key certificate, and Bachar will notice that. The certification authority must of course be trustworthy. We can also build hierarchies between CAs as seen below.





Let us recall that we solve only a part of the problem, because Omar can still intercept the communication between Ali and the CA ! Hence, we must have a **secure authenticated channel with the CA**.

[1]Pas MYM, mécréant.

1.1.4 Hybric encryption

Note that in practice, we use **hybrid encryption** to improve both efficiency and bandwith (avoid transmitting a lot of data).

Going public

In practice: hybrid encryption

To improve both efficiency and bandwidth we can combine asymmetric encryption 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 $(c_k, c_m) = \left(\mathsf{Enc}_{\text{Bob's public key}}(k), \mathsf{Enc}_k(m)\right)$.

Bob decrypts as follows:

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

12 / 96

There are many ways of doing public-key crypto, based on different mathematical problems. There is **large integer factorization** ($\rightarrow$ RSA), **discrete logarithm problem** ($\rightarrow$ ElGamal DSA, and DH), and **elliptic curves** ($\rightarrow$ ECDSA)

1.2 RSA : a factorization problem

RSA is mainly about key generation. It is an interessant way of generating keys but is subject to a lot of attacks.

1.2.1 Key generation

The key generation relies on the fact that factorizing large integers is complicated. Hence, it is based on two prime numbers p and q, kept private but **their product n is kept public**, as part of the public key.

The other part of the public key is an exponent that will come in play when encrypting. We note it e. It is public, and satisfies

$$3 \leq e \leq (p-1)(q-1) - 3 \qquad \gcd(e, (p-1)(q-1)) = 1\,.$$

Often, one chooses e and then generates the primes.

The private key is the inverse of e modulo $(p-1)(q-1)$.

Rivest-Shamir-Adleman

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 $e \in \{3, 17, 2^{16}+1\}$ 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) and the private key is (n, d).

26 / 96

1.2.2 RSA textbook encryption

Here is an encryption scheme that stinks a lot : we elevate the message to the power of e to get the ciphertext. To decrypt the ciphertext, we elevate it to the power of d, as e is the inverse of d.

1.2.3 RSA textbook signature

It is basically similar, but here we are **signing** a message with the private key, so $s = m^d \bmod n$. Again, to check the signature, we can elevate s to the power of e to verify if we indeed get the message back (valid signature) or not.

Rivest-Shamir-Adleman

RSA "textbook encryption" (don't use!)

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

- **Encryption:** $c = m^e \bmod n$
- **Decryption:** $m = c^d \bmod n$

Why is it correct?

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

27 / 96

Rivest-Shamir-Adleman

RSA "textbook signature" (don't use!)

From message $m \in \mathbb{Z}_n$ to signature $s \in \mathbb{Z}_n$ and back:

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

28 / 96

1.2.4 Attacks on RSA

RSA is subject to many attacks, one of the easiest ones being the **short message attack**, when we know that e is small. We can send short messages so the modular reduction by n does nothing in the encryption. So we can find back m by taking the e-th root of $c = m^e$. One just has to try the different cs.

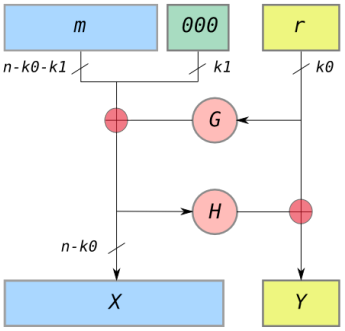
1.2.5 Powering RSA

RSA-OAEP : optimal asymmetric encryption padding

Padding is really interesting. We here add padding to the original message, and randomness to the pre-processing of the message before encryption. Encryption is done as always, exponentiating by e, moduling by n, but the input is a new one : the message after pre-processing.

Rivest-Shamir-Adleman | RSA for confidentiality

RSA-OAEP: Optimal asymmetric encryption padding



[Mihir Bellare and Phillip Rogaway, Eurocrypt 1994]

Encryption: m ... then $c = (X\|Y)^e \bmod n$
Decryption: $(X\|Y) = c^d \bmod n$, then ... m

RSA-OAEP

- makes the scheme probabilistic;
- prevents that an adversary recovers any portion of the plaintext without being able to invert RSA;
- is shown to be IND-CCA secure, assuming that G and H behave as random oracles.

32 / 96

This helps with the short message attack, makes the scheme probabilistic, and introduces diffusion. In addition, it is shown to be IND-CCA secure. Wow !

RSA-KEM : Key Encapsulation Method

Here, we present a method to do **hybrid encryption** using RSA. It is not an encryption scheme that we present ! Remember that in hybrid encryption, Ali somehow chooses the symmetric key and sends it to Bachar in an encrypted way. Here we will have something different :

- Ali chooses a random string m of size n, where n is part of his RSA public key.
- He hashes m : gives $k = \text{hash}(m)$ which will be the secret key
- Ali encrypts $c = m^e \mod n$, and sends c to Bachar

To decrypt (actually, *decapsulate*), Bachar does the following :

- Recovers $m = c^d \mod n$
- Computes $k = \text{hash}(m)$

We thus see that the key k is encapsulated in m, himself encrypted by RSA textbook encryption.

1.2.6 Computational analysis of RSA

Encryption in RSA consists in two operations : exponentiating for encrypting, exponentiating for decrypting. If we choose a small e, it will be fast for encryption. However, there might still have some problems. Below, we list how to improve the computations for RSA :

- Encryption – computing a^e : fast encryption using **square and multiply** (short public exponents)
- Decryption – computing c^d : fast decryption using **Chinese Reminder Theorem** (see next 2 pages)

7.5.2 Fast Decryption with the Chinese Remainder Theorem

We cannot choose a short private key without compromising the security for RSA. If we were to select keys d as short as we did in the case of encryption in the section above, an attacker could simply brute-force all possible numbers up to a given bit length, i.e., 50 bit. But even if the numbers are larger, say 128 bit, there are key recovery attacks. In fact, it can be shown that the private key must have a length of at least $0.3t$ bit, where t is the bit length of the modulus n. In practice, e is often chosen short and d has full bit length. What one does instead is to apply a method which is based on the Chinese Remainder Theorem (CRT). We do not introduce the CRT itself here but merely how it applies to accelerate RSA decryption and signature generation.

Our goal is to perform the exponentiation $x^d \bmod n$ efficiently. First we note that the party who possesses the private key also knows the primes p and q. The basic idea of the CRT is that rather than doing arithmetic with one "long" modulus n, we do two individual exponentiations modulo the two "short" primes p and q. This is a type of transformation arithmetic. Like any transform, there are three steps: transforming into the CRT domain, computation in the CRT domain, and inverse transformation of the result. Those three steps are explained below.

Transformation of the Input into the CRT Domain

We simply reduce the base element x modulo the two factors p and q of the modulus n, and obtain what is called the modular representation of x.

$$x_p \equiv x \bmod p$$
$$x_q \equiv x \bmod q$$

Exponentiation in the CRT Domain

With the reduced versions of x we perform the following two exponentiations:

$$y_p = x_p^{d_p} \bmod p$$
$$y_q = x_q^{d_q} \bmod q$$

where the two new exponents are given by:

$$d_p \equiv d \bmod (p-1)$$
$$d_q \equiv d \bmod (q-1)$$

Note that both exponents in the transform domain, d_p and d_q, are bounded by p and q, respectively. The same holds for the transformed results y_p and y_q. Since the two

primes are in practice chosen to have roughly the same bit length, the two exponents as well as y_p and y_q have about half the bit length of n.

Inverse Transformation into the Problem Domain

The remaining step is now to assemble the final result y from its modular representation (y_p, y_q). This follows from the CRT and can be done as:

$$y \equiv [q\, c_p]\, y_p + [p\, c_q]\, y_q \bmod n \tag{7.7}$$

where the coefficients c_p and c_q are computed as:

$$c_p \equiv q^{-1} \bmod p, \qquad c_q \equiv p^{-1} \bmod q$$

Since the primes change very infrequently for a given RSA implementation, the two expressions in brackets in Eq. (7.7) can be precomputed. After the precomputations, the entire reverse transformation is achieved with merely two modular multiplications and one modular addition.

Before we consider the complexity of RSA with CRT, let's have a look at an example.

Example 7.6. Let the RSA parameters be given by:

$$\begin{aligned} p &= 11 & e &= 7 \\ q &= 13 & d &\equiv e^{-1} \equiv 103 \bmod 120 \\ n &= p \cdot q = 143 \end{aligned}$$

We now compute an RSA decryption for the ciphertext $y = 15$ using the CRT, i.e., the value $y^d = 15^{103} \bmod 143$. In the first step, we compute the modular representation of y:

$$\begin{aligned} y_p &\equiv 15 \equiv 4 \bmod 11 \\ y_p &\equiv 15 \equiv 2 \bmod 13 \end{aligned}$$

In the second step, we perform the exponentiation in the transform domain with the short exponents. These are:

$$\begin{aligned} d_p &\equiv 103 \equiv 3 \bmod 10 \\ d_q &\equiv 103 \equiv 7 \bmod 12 \end{aligned}$$

Here are the exponentiations:

$$\begin{aligned} x_p &\equiv y_p^{d_p} = 4^3 = 64 \equiv 9 \bmod 11 \\ x_q &\equiv y_q^{d_q} = 2^7 = 128 \equiv 11 \bmod 13 \end{aligned}$$

In the last step, we have to compute x from its modular representation (x_p, x_q). For this, we need the coefficients:

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

The discrete logarithm problem is also a problem hard to solve, like the factorization of large integers. Here, we focus on the definition of a **group**, that we dont recall in this document. To enter this section, we need ourselves a group :

- A set of items : it will be $\mathbb{Z}_p^*$. It is a set of strings where each character can take values from 0 to $p-1$.
- A composition relation : we choose modular addition
- Identity : 0
- Each item has an inverse

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

What is the discrete logarithm problem?

Domain parameters:

- Let p be a large prime.
- Let g be a generator of $\mathbb{Z}_p^*$, i.e., $\{g^i \bmod p \ : \ i \in \mathbb{N}\} = \mathbb{Z}_p^*$.

Discrete logarithm problem (DLP)

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

There are currently no known polynomial time algorithm to solve this problem.

For 128-bit security, NIST recommends p to be at least 3072-bit long.
[NIST SP 800-57, see also https://www.keylength.com/]

1.3.1 Key generation

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

Key generation

Domain parameters (public, common to all users):

- Let p be a large prime.
- Let g be a generator of $\mathbb{Z}_p^*$, i.e., $\{g^i \bmod p \ : \ i \in \mathbb{N}\} = \mathbb{Z}_p^*$.

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

- Privately choose a random integer $a \in [1, p-2]$
- Compute $A = g^a \bmod p$

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

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

Key generation (group notation)

Domain parameters (public, common to all users):

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

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

- Privately choose a random integer $a \in [1, q-1]$
- Compute $A = [a]g$

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

1.3.2 ElGamal encryption

This is an encryption scheme, as stated in the name. We come armed with the key generated like at previous section. We are now ready to fight. We have a message $m \in \mathbb{Z}_p^*$ that we want to send to Ali. We need his public key, he has generated it before, it is $A = g^a \mod p$.

- **Encryption** : he uses the generated key as a multiplicative mask with an exponent. Typically :

$$c = mA^k \mod p$$

  where k is a random integer $\in [1, p-2]$ [2]. k is not shared. However, Ali sends $K = g^k \mod p$ alongside c.

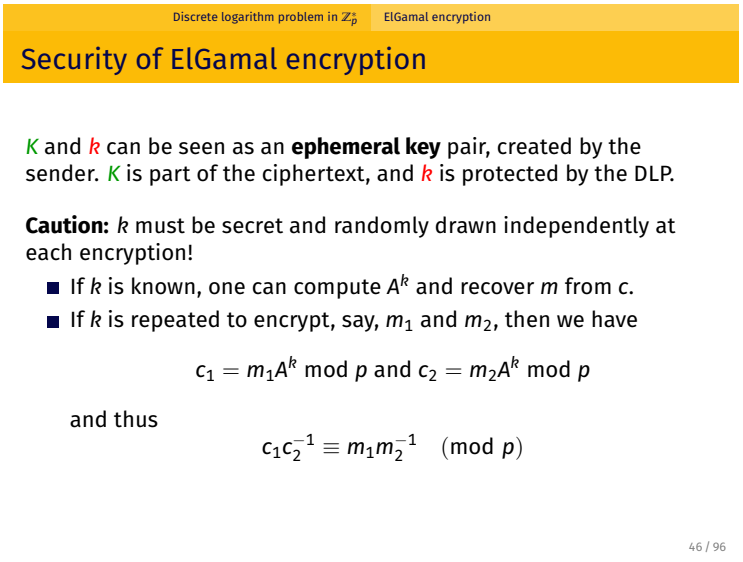
- **Decryption** : Ali receives c and K, and computes

$$m = cK^{-a} \mod p$$

Just a quick reminder here :

- Are public : K, g, A, p
- Are private : a, k

k needs to be ephemeral



Now, let's have a talk about the security of ElGamal encryption scheme.

[2] Note here the bounds : for exponents, it is often $[1, p-2]$

1.3.3 The Diffie-Hellman problem

The Diffie-Hellman problem

Domain parameters:

- Let p be a large prime.
- Let g be a generator of $\mathbb{Z}_p^*$, i.e., $\{g^i \bmod p \ : \ i \in \mathbb{N}\} = \mathbb{Z}_p^*$.

Diffie-Hellman problem

Given $X = g^x \bmod p$ and $Y = g^y \bmod p$, find $X^y = Y^x = g^{xy} \bmod p$.

It is an easier problem than DLP (breaking DHP does not give you the exponents). However, there are currently no known polynomial time algorithm to solve the DHP either.

The DHP does not give x and y, so it does not solve the DLP. We can see two things here :

- Retrieving the secret key a or k from A or K is DLP
- Breaking ElGamal requires to compute g^{ak}, so this is DHP.