Public key cryptography

Robert Rolland

December, 2001

1 Introduction

For undertanding the starting point of **public key cryptography** (or **asymmetric cryptography**) and the difference between public key cryptography and secret key cryptography, let us consider first the ciphering problem.

In secret key cryptography or symmetric cryptography the sender and the receiver use the same secret key. The main problem is to exhange this secret key. You can think for example at what happens if a large amount of users have to communicate using a secret key cryptosystem.

In public key cryptography, each user X has a private key d_X (nobody except X knows this key), and a public key e_X which is published and can be known by everybody. These keys define a public ciphering function (denoted also by e_X) and a private deciphering function (denoted also by d_X) such that $d_X \circ e_X = Identity$.

When B wants to send a message M to A, B uses the public key of A to compute the ciphertext C:

$$C = e_A(M).$$

The ciphertext C is sent to A who is able to recover the plaintext

$$M = d_A(C) = d_A\left(e_A(M)\right).$$

So in a public key ciphering system, we have to build for each user X a function e_X which is easy to compute and hard to inverse, that is, it must be impossible **in practice** to compute d_X. Such a function is a **one-way function**. There is some paradoxe because the user X must know d_X. So the function X is not only a one-way function, but also a **trapdoor function**. A one-way trapdoor function is hard to inverse unless you know a secret (the trapdoor).

However, public key cryptosystems are more time consuming than secret key cryptosystems (there is a factor about 1000 between the two). So the solution is to use a public key cryptosystem to cipher the key of a secret key cryptosystem (which is a rather short plaintext) and once the secret key exchanged, to use a secret key cryptosystem to cipher the messages.

For example A and B agree on a public key cryptosystem and a secret key cryptosystem. If B wants to send some message M to A, first he generates by random a key k (the **session key**) for the secret key cryptosystem, and computes $k(M)$. Then B computes also $e_A(k)$, using the public key e_A of A. Now B sends $e_A(k)$ and $k(M)$ to A. The user A with his private key d_A recovers k, then using k recovers the plaintext M. Such a system has the advantage that the secret key is a session key and so is changed each time the system is used.

Public key is not only used for ciphering but also for other tasks like signature.

Let us remark that public key cryptosystems doesn't solve completely the key management problem. Public keys are kept on servers and it is necessary to verify that keys are not modified or forged.

We will describe in detail the RSA cryptosystem which is used for ciphering and signing. We will only give some indications on ElGamal cryptosystem.

2 Ciphering

2.1 RSA

2.1.1 The RSA function

Let $n = pq$ the product of two distinct large primes p and q. We denote by $\phi(n)$ the Euler's totient function.

$$\phi(n) = (p-1)(q-1).$$

Let e be relatively prime to $\phi(n)$. We can compute d such that

$$ed \equiv 1 \quad (\phi(n)).$$

- The **public key** is (n, e) where n is the **modulus**, e is the **ciphering exponent**.

- The **private key** is (n, d) where d is the **deciphering exponent**.

Let $0 \leq x < n$. We define the RSA fonction by

$$RSA_{n,e}(x) = x^e \mod n.$$

Theorem 2.1 *The RSA function is a* **permutation** *of* $\{0, \cdots, n-1\}$ *and it's inverse is given by*

$$RSA_{n,e}^{-1}(y) = y^d \mod n.$$

Proof. Let $y = x^e \mod n$ be in the image. Let us compute

$$y^d \mod n = x^{de} \mod n = x^{1+k\phi(n)} \mod n = x \mod n = x.$$

Hence the function is injective and since $\{0, \cdots, n-1\}$ is finite, the function is bijective. Moreover $y^d \mod n$ is the inverse function. □

We hope that RSA function is a **good candidate to be one-way**. RSA function has a **trapdoor**: everybody knowing the public key (n, e) and the factorization of n is able to compute d.

Now concerning the RSA function, the building of n, e, d, the computation of the function, the security, many problems arise:

- **How long has to be** n**?** In practice n must have at least 700 bits (1024 bits is of course better).
- **Is there some other conditions on** n**?** The security of the RSA function depends for a large part upon the difficulty of factoring. Hence some n with particular properties are forbidden. For example by the Pollard's $p-1$-method we know that if $p-1$ is the product of low prime factors, we can factor n. But these particular cases occur with a very small probability. However we can include some test in the choice of p and q for discarding bad events.
- **How to find long prime numbers?** The quickest way to find large primes is to use a probabilistic test of primality (For instance the **Miller Rabin test**). With such a test we are not absolutly sure that p is prime, but the probability for claiming prime a composite number is very low.
- **How to find** e**?** One way to do that is to choose e at random and to test if $\gcd(e, \phi(n))) = 1$. If not, we do a new choice.

- **How to compute d?** Knowing the modulus n and the ciphering exponent e, the deciphering exponent d can be computed using the extended euclidian algorithm.

- **What about the semantic security?** Semantic security means that no information on the plaintext can be recovered in practice (that is in polynomial expected time) from the ciphertext. The RSA function is **not semantically secure**. For instance if e is odd, the Jacobi symbol of $y = x^e \mod n$ reveals the Jacobi symbol of x

$$\left(\frac{x^e}{n}\right) = \left(\frac{x^e}{p}\right)\left(\frac{x^e}{q}\right) = \left(\frac{x}{p}\right)\left(\frac{x}{q}\right) = \left(\frac{x}{n}\right).$$

  This is catastrophic if for instance the message sent means "yes" or "no" according to the value of the Jacobi symbol of x. So if we use a cryptosystem which is not semantically secure we have to choose an information coding adapted to the cryptosystem deficiency.

  On the other hand, some important partial informations do not leak from the ciphertext.

  Theorem 2.2 *Let f be the function defined by*

$$f(y) = \begin{cases} 0 & if\ RSA_{n,e}^{-1}(y)\ is\ even, \\ 1 & if\ RSA_{n,e}^{-1}(y)\ is\ odd. \end{cases}$$

  If f can be computed in pratice, it is possible to decipher any message.

  Theorem 2.3 *Let g be the function defined by*

$$g(y) = \begin{cases} 0 & if\ RSA_{n,e}^{-1}(y)\ is\ \leq n/2, \\ 1 & if\ RSA_{n,e}^{-1}(y)\ > n/2. \end{cases}$$

  If g can be computed in pratice, it is possible to decipher any message.

  Proof. n is odd, hence

$$f(2^e c \mod n) = g(c),$$

$$g(2^{-e}c \mod n) = f(c).$$

So we only have to prove the second theorem. A dichotomie gives us the result.

A bit wich is as hard to compute as the entire plaintext is called and **hard-core bit**.

- **Is it possible to break RSA function without factoring the modulus?** The answer at this question is not known. Let us give some results related to this problem.

Theorem 2.4 *It is equivalent to know $\phi(n)$ or the factorization of n.*

Proof. If p and q are known it is easy to compute in polynomial time $\phi(n) = (p-1)(q-1)$.

Conversely the following relations hold

$$(p-1)\left(\frac{n}{p} - 1\right) = \phi(n),$$

$$p^2 - (n + 1 - \phi(n))\, p + n = 0.$$

We can now solve this equation in polynomial time. □

Theorem 2.5 *There is a* **probalistic algorithm** *where the input is the modulus n and the ciphering exponent e, and the output the factorization pq of n, running in* **probabilistic polynomial time** *(Las Vegas algorithm) and* **calling an oracle** *which compute d.*

Proof. Let us remark first that

$$x^2 \equiv 1 \quad (n)$$

if and only if

$$\begin{cases} x^2 \equiv 1 \quad (p) \\ and \\ x^2 \equiv 1 \quad (q) \end{cases}$$

and these equations are verified if and only if

$$\begin{cases} x \equiv \pm 1 \quad (p) \\ and \\ x \equiv \pm 1 \quad (q) \end{cases}$$

There are four solutions, two of them are the trivial ones $x \equiv \pm 1 \quad (n)$.

If x is a non-trivial solution then n divides $(x+1)(x-1)$ but don't divide $(x-1)$ nor $(x+1)$. Hence

$$\gcd(x+1, n) = p \; or \; q,$$
$$\gcd(x-1, n) = q \; or \; p.$$

So, if we know a non-trivial square root of 1 modulo n, we can compute in polynomial time the factorization of n.

Now the problem is: how to compute a non-trivial square root of 1 modulo n?

Choose at random a w such that $1 < w \leq n-1$. If we are lucky, $\gcd(w, n) > 1$, and we have a prime factor of n.

If not, we write (don't forget that d is known)

$$ed - 1 = 2^s r,$$

where $s \geq 1$ and r is odd. We know that

$$w^{2^s r} = w^{ed-1} = w^{k\phi(n)},$$

then

$$w^{2^s r} \equiv 1 \quad (n).$$

Hence, it exists a smallest t such that $t \leq s$ and

$$w^{2^t r} \equiv 1 \quad (n).$$

Let us denote by t_0 this integer. If $t_0 > 0$ let us define

$$v_0 = w^{2^{t_0-1}r},$$

so that

$$v_0 \not\equiv 1 \quad (n)$$

and

$$v_0^2 \equiv 1 \quad (n).$$

Now if

$$v_0 \not\equiv -1 \quad (n)$$

we have found a non trivial square root of 1 modulo n.

The algorithm fails an we must reapply it to another w in the two following cases

a) $t_0 = 0$, that is

$$w^r \equiv 1 \quad (n).$$

b) there is a t such that $0 \leq t \leq s-1$ and such that

$$w^{2^t r} \equiv -1 \quad (n).$$

The number of w for which the agorithm fails is by the Rabin's theorem $\leq \frac{\phi(n)}{4}$. Hence the probabilité to get the result with a given w is $\geq 3/4$. If we cannot obtain the result with this w we guess another one. The probability of an infinite number of iterations is 0 and moreover the expected number of iterations is $\leq 4/3$. Now we can conclude knowing that each iteration has a polynomial cost.□

On the other hand if e the factorization of n are known, clearly we can compute d. So when we have a RSA function $RSA_{n,e}$, computing d is **in pratice** as hard as computing the factors of n.

But this doesn't exclude the possibility of computing the inverse $RSA_{n,e}^{-1}$ without computing d.

2.1.2 RSA Encryption Decryption Protocol

When we have a plaintext, how to cipher it, using the RSA function? The hardness of RSA function is not sufficient to have at the end a secure system. For instance, if your plaintext is an ascii encoded text anf if you cipher one by one each of the ascii coded characters, the system can be easily broken. The **scheme** is bad.

We present here the RSAES-OAEP scheme recommended in the PKCS♯1v2.1 RSA document: **RSA Cryptography Standard** (January 5, 2001). (RSAES: **RSA E**ncryption **S**cheme. OAEP:**O**ptimal **A**symmetric **E**ncryption **P**adding).

Let us suppose we start from a block of plaintext M (the message to be encoded which is an octet string a length $mlen$ octets.

- **Encoding process**. This octet string M is transformed in an octet string EM of length $emlen$. This bloc EM is the **encoded message**. The transformation used will be discribed later. It is the EME-OAEP **encoding** process. (EME: **E**ncoding **M**ethod for **E**ncryption. OAEP: **O**ptimal **A**symmetric **E**ncryption **P**adding).
- **To integer translation**. Now EM is translated to an integer m by OS2IP (Octet String To Integer Primitive).
- **Ciphering process** The integer m is transformed in c by RSAEP (RSA Encryption Primitive). This transform is just the RSA function.
- **To octet string translation**. The integer c is translated in an octet string C by I2OSP (Integer To Octet String Primitive).

Now when the receiver get C he applies the decryption scheme:

- **To integer translation**. C is translated to an integer c by OS2IP (Octet String To Integer Primitive).
- **deciphering process** The integer c is transformed in m by RSADP (RSA Decryption Primitive). This transform is just the RSA^{-1} function.
- **To octet string translation**. The integer m is translated in an octet string EM by I2OSP (Integer To Octet String Primitive).
- **Decoding process**. This octet string EM is transformed in an octet string M. The transformation used is the EME-OAEP **decoding process**.

The OS2IP process is very simple. If $X_1, X_2, \cdots, X_l$ are the octets of the octet string, let x_{l-i} the integer represented by the octet X_i. Let

$$x = \sum_{j=0}^{l-1} x_j (256)^j.$$

x is the output of the OS2IP.
The description of the I2OSP primitive, which is the inverse of OS2IP, is straightforward.

The EME-OAEP coding and decoding primitives use a **hash function** h and a **mask generation function** g. The function h input is an octet string and the output is an octet string of fixed length $hlen$. The function g has for input an octet string and a length l and for output an octet string of length l.

For the encoding primitive we start from M (message to be encoded) and P an encoding parameter (which is an octet string). We generate an octet string PS consisting of $emlen - mlen - 2hlen - 1$ zero octets. We compute $pHash = h(P)$ an octet string of length $hlen$. Then we contatenate $pHash$, PS, the octet 01 and M to obtain DB

$$DB = pHash + PS + 01 + M$$

(where $+$ denotes the concatenation). Let us remark that DB's length is $emlen - hlen$.
Now we generate a random octet string s of length $hlen$ and we compute $dbmask = g(s, emlen - hlen)$. We successively compute

$$maskedDB = DB \oplus dbmask,$$

$$smask = g(maskedDB, hlen),$$

$$maskeds = s \oplus smask,$$

$$EM = maskeds + maskedDB.$$

For the decoding primitive we start from EM (the encoded message) and P the encoding parameter (which is known by everybody). The $mlen$ first octets of EM give us $maskeds$, the remaining octets give us $maskedDB$. Now we compute successively

$$smask = g(maskedDB, hlen),$$

$$s = smask \oplus maskeds,$$
$$dbmask = g(s, emlen - hlen),$$
$$DB = dbmask \oplus maskedDB.$$

Now it is easy to recover M.

2.1.3 Attacks of RSA

We will not give a complete lists of known attacks, nor the details of all the attacks with their variants against RSA. We will just give some general results on these attacks and some examples. We show that designing a secure protocol using the RSA function is not a staightforward task. In particular we point out that a naive ciphering of the plain text is not sufficient. We must preprocess the data, for example as described in the previous paragraph, before using the RSA function.

Let us remark that **any efficient algorithm for factoring** would give a means to break RSA. We refer to another part of the course for learning more about factoring methods.

- **Common modulus.** This is a very simple an very known attack. Two users must have different modulus. If not, let (n, e_A) the A's public key and (n, e_B) the B's public key. The first point is that A for example knowing e_A and d_A can factor n. Then, knowing e_B and the n's factorization , he is able to find d_B. For the second point let us suppose e_A and e_B be relatively prime. We can find x and y such that

  $$xe_A + ye_B = 1.$$

  If a user C sends a message m to A and B, he sends

  $$c_A = m^{e_A} \mod n,$$
  $$c_B = m^{e_B} \mod n.$$

  Then anybody is able to recover m by

  $$m = c_A^x c_B^y \mod n.$$

- **Low private exponent.** One can think to choose a small d for increasing the deciphering speed. But this is not a good choice. There is an attack due to Wiener based on continued fractions. We start from the relation

$$ed \equiv 1 \quad (\phi(n)),$$

where $\phi(n) = (p-1)(q-1) = n-(p+q)+1$. This relation can be rewrited

$$ed = 1 + k\left(\frac{n+1}{2} - \frac{p+q}{2}\right),$$

where k is an integer.

From the last relation we get

$$\left|\frac{2e}{n} - \frac{k}{d}\right| = \frac{|2 + k(1-(p+q))|}{nd},$$

so that if $\frac{|k(p+q-1)-2|}{n} < \frac{1}{2d}$ we obtain

$$\left|\frac{2e}{n} - \frac{k}{d}\right| < \frac{1}{2d^2}.$$

This inequality shows that k/d is a convergent of the continued fraction expansion of $2e/n$. If p and q are balanced, $p + q = \mathcal{O}(\sqrt{n})$. Moreover $k = \mathcal{O}(d)$, hence, if $d = \mathcal{O}(n^{0.25})$ the attack applies.

There exist improvements of the Wiener's attack, in particular the Boneh-Durfee attack which permits to recover d as long as $d < n^{0.292}$.

- **Low public exponent, Broadcast attack.** Let us give first a very simple situation. Suppose that there are at least three participants P_1, P_2, P_3, using for public keys $(n_1, 3)$, $(n_2, 3)$, $(n_3, 3)$. We will suppose that (n_1, n_2, n_3) are pairwise relatively prime (if not, we can factorize some of the n_i). If somebody sends the message m to P_1, P_2, P_3, and if $m < \inf(n_1, n_2, n_3)$ by eavesdropping one obtains

$$c_1 = m^3 \quad (n_1),$$

$$c_2 = m^3 \quad (n_2),$$

$$c_3 = m^3 \quad (n_3),$$

and by the Chinese remainder algorithm

$$c = m^3 \quad (n_1 n_2 n_3).$$

But $m^3 < n_1 n_2 n_3$, so that $c = m^3$ in $\mathbb{N}$. To recover m one have just to extract the cubic root of c in $\mathbb{N}$.

Suppose now k parties $P_1, P_2, \cdots, P_k$, having public keys $(n_1, e_1), (n_2, e_2), \cdots, (n_k, e_k)$ where the n_i are pairwise relatively prime. Suppose that somebody wants to send a message $m < n_{min} = \inf n_i$ to the k parties P_i. To havoid the pevious attack he pads or broadcasts m before sending it. More precisely he sends to each participant P_i

$$c_i = (f_i(m))^{e_i} \mod n_i,$$

where for each i, f_i is a monic polynomial function of degree $deg(f_i)$, such that the application giving c_i from m be injective. Now let us set

$$g_i = f_i^{e_i} - c_i.$$

If $k > \max e_i deg(f_i)$, an eavesdropper is able to recover m. To do that, using the chinese remainder algorithm he builds the polynomial function

$$g(x) = \sum_{i=1}^{k} u_i g_i(x),$$

where

$$u_i \equiv \begin{cases} 1 & (n_i), \\ 0 & (n_j) \quad if\ j \neq i. \end{cases}$$

Let us remark that g is monic.

Let $n = n_1 n_2 \cdots n_k$. The message m is the unique solution of

$$g(x) \equiv 0 \quad (n)$$

such that

$$m < n_{min} < n^{\frac{1}{k}} \leq n^{\frac{1}{deg(g)}}.$$

Then he uses the following Coppersmith's theorem:

Theorem 2.6 *Let n be an integer, and $g \in \mathbb{Z}[x]$ a monic polynomial of degree $deg(g)$. Set $X = n^{\frac{1}{deg(g)} - \epsilon}$ for some $\epsilon \geq 0$. Then it is possible to efficiently find all the integers x such that $g(x) = 0$ and $|x| < X$. The running time is dominated by the time it takes to run the LLL algorithm on a lattice of dimention $\mathcal{O}\left(\min\left(\frac{1}{\epsilon}, \log_2(n)\right)\right)$.*

- **Timing attacks** For smart card implementations of RSA, an attack due to Kocher by precisely mesuring the time it takes to perform an RSA decryption exposes the decryption key. The attack is based on the fact that depending upon the value of the bit i of d, the i^{th} loop of the computation implies one or two multiplications.

  Let us suppose that the algorithm used to compute c^d is the the following

  $C := c; D := d; R := 1;$
  $while\ D > 1\ do$
  $\quad begin$
  $\quad\quad if\ odd(D)\ then$
  $\quad\quad\quad begin$
  $\quad\quad\quad\quad R = R * C \mod n;$
  $\quad\quad\quad\quad D := D - 1;$
  $\quad\quad\quad end;$
  $\quad\quad C := C * C \mod n;$
  $\quad\quad N := N/2;$
  $\quad end;$
  $R = R * C \mod n;$
  $output\ R.$

  Let us set $d = \sum_{i=0}^{k} d_i 2^i$. We know that d is odd so that $d_0 = 1$ and after the first loop, $R = c$ and $C = c^2 \mod n$. If $d_1 = 1$ the second loop computes first $R * C = c * c^2 \mod n$ and if $d_1 = 0$ this product is not done. Let $t(c)$ the expected value of the time it takes for computing $c * c^2 \mod n$. Let $T(c)$ the expected value of the time it takes for computing $c^d \mod n$. If $d_1 = 1$, $t(c)$ and $T(c)$ are correlated. If $d_1 = 0$, there is no correlation

between $t(c)$ and $T(c)$. So by performing the computation of $t(c)$ and $T(c)$ for many different values of c we can determine if d_1 is 0 or 1. Now we can do the same thing for $d_2, \cdots, d_k$.

Kocher has also designed an attack based on mesuring power consumption (**power cryptanalysis**).

2.2 ElGamal

2.2.1 The exponential function and the index function

Let p a prime number and $(\mathbb{Z}/p\mathbb{Z})^*$ the multiplicative group of non-zero integers modulo$p$. We know that this group is cyclic. Let α be a primitive element. Let E_α be the function from $(\mathbb{Z}/p\mathbb{Z})^*$ onto $(\mathbb{Z}/p\mathbb{Z})^*$ defined by

$$E_\alpha(x) = \alpha^x \mod p.$$

The inverse of E_α is the **index** function or the **discrete logarithm** i_α.

Computing the discrete logarithm of a given element of $(\mathbb{Z}/p\mathbb{Z})^*$, namely the **discrete logarithm problem**, is known to be hard. More precisely, the problem is NP.

2.2.2 The Diffie-Helman problem

Let us suppose that we know

$$\alpha^a \mod p,$$

$$\alpha^b \mod p.$$

Compute

$$\alpha^{ab} \mod p.$$

This problem is the **Diffie-Hellman problem**. If we can solve the discrete logarithm problem, clearly we can solve the Diffie-Hellman problem. The converse is not known.

Let us define also the **decision Diffie-Hellman problem**: Let us suppose that $\alpha^a \mod p$, $\alpha^b \mod p$, and y are given. Is $y = \alpha^{ab}$?

2.2.3 The ElGamal function

Let p a large prime, q a prime factor of $p-1$ and α an order q element of $(\mathbb{Z}/p\mathbb{Z})^*$. We denote by G the multiplicative subgroup of $(\mathbb{Z}/p\mathbb{Z})^*$ generated by α. Let a be in $\mathbb{Z}/q\mathbb{Z}$ and $\beta = \alpha^a \mod p$ (which is an element of G). We define the ElGamal function from $G \times (\mathbb{Z}/q\mathbb{Z})$ onto $G \times G$ by

$$ELG_{p,q,\alpha,\beta}(x,k) = (y_1, y_2),$$

with

$$y_1 = \alpha^k \mod p,$$

and

$$y_2 = x\beta^k \mod p.$$

If we find x knowing y_1 and y_2, we can compute also $\beta^k = \alpha^{ak}$ knowing α^a and αk. Hence recovering x is solving an instance of the Diffie-Hellman problem. Let us remark that it is not a general instance of the Diffie-Hellman problem because in fact all the element y_1, y_2, x, are in the G subgroup of $(\mathbb{Z}/p\mathbb{Z})^*$. But in pratice this seems to be no easier than solving the problem for a random instance in $(\mathbb{Z}/p\mathbb{Z})^*$.

We hope that this function be one-way. It is a trapdoor like function: if somebody knows the index a of β, he is able to recover x from the relation

$$y_2 = xy_1^a.$$

Let us remark that we don't recover k.

- **The public key** of A is (p, q, α, β)
- **The private key** of A is (p, q, α, a)
- **The Random number** k is guessed by the sender for each message. The number k could not be used more than one time.
- **The integer message** x is an element of G to be computed from the plaintext.

Hence, the ciphering process is not deterministic. The ciphertext depends upon the plaintext and a random k.

As for RSA function we can ask many questions about ElGamal function.

- **How long has to be p?** It is about the same length as the RSA modulus length. Namely, n must have at least 700 bits (1024 bits is of course better).
- **Is there some other conditions on n?** The security of the ElGamal function depends for a large part upon the hardness of the discrete logarithm problem. Hence some p with particular properties are forbidden. For example $p-1$ must have a large prime factor to avoid Pohlig-Hellman's attack of the discrete logarithm problem.
- **What about the semantic security?** The semantic security of ElGamal encryption is equivalent to the decision Diffie-Hellman problem.

We point out that for the multiplicative group $(\mathbb{Z}/pZ)^*$ we have **subexponential algorithms** to solve the discrete logarithm problem. Working over the multiplicative group of a general finite field doesn't increase this complexity. On the other hand, at the moment we don't know any subexponential algorithm to solve the discrete logarithm problem on the group of an elliptic curve. With elliptic cryptography the size of the key can be shorter (about 200 bits).

3 Signature

Signature uses the same tools than public key ciphering. Each user X has a private key d_X (nobody except X knows this function), and a public key e_X which is published and can be known by everybody. The private key defines a **signature function** s_{d_X}. The public key defines a **verifier** v_{e_X} such that

$$v_{e_X}(m,t) = \begin{cases} 0 & if \quad t \neq s_{d_X}(m), \\ 1 & if \quad t = s_{d_X}(m). \end{cases}$$

We will describe here **signature schemes with appendix** and not **signature scheme with message recovery**.

Suppose that B wants to send a signed message M to A.

- First B uses a public hash function h to compute the digest $m = h(M)$.
- Now B uses its private key d_B to compute the signature part $s = s_{d_B}(m)$.
- The signed message (M, s) is transmitted to A.

Let us remark that the signed message is **the message itself** M **with an appendix** s.

The receiver A knows M and s.

- First he computes $m = h(M)$
- Now A uses the public key e_B to verify the signature by computing $v_{e_B}(m, s)$.

The **NIST** (**N**ational **I**nstitute of **S**tandards and **T**echnology) defines in the **FIPS 186-2** (**F**ederal **I**nformation **P**rocessing **S**tandard) the standard **DSS** (**D**igital **S**ignature **S**tandard).

The standard allowes two signature algorithms: **DSA** (**D**igital **S**ignature **A**lgorithm) and **RSA**.

3.1 DSA

The DSA signature is based upon the discrete logarithm problem. It is nearly related to ElGamal cryptosystem. Let us describe now the signature function and the verifier.

Let p a prime modulus where

$$2^{k-1} < p < 2^k,$$

with k a multiple of 64 such that

$$512 < k < 1024.$$

Let us remark that p has k bits.

Let q be a prime divisor of $p - 1$ where

$$2^{159} < q < 2^{160}.$$

The prime q has 160 bits.

Letα be an order q element of $(\mathbb{Z}/p\mathbb{Z})^*$. Let a be in $(\mathbb{Z}/q\mathbb{Z})^*$ and $\beta = \alpha^a \mod p$.

The public key of A is (p, q, α, β), its private key is (p, q, α, a). If A wants to send a signed message to B he compute the digest m of the message M and chooses a random k (one for each signature) in $(\mathbb{Z}/q\mathbb{Z})^*$. Next he computes

$$r = (\alpha^k \mod p) \mod q$$

and

$$s = (k^{-1}(m + ar)) \mod q.$$

Then A transmits $(M, (r, s))$.

To verify the received signed message $(M, (r, s))$, B computes the digest m of M and

$$\begin{aligned} w &= s^{-1} \mod q, \\ u_1 &= mw \mod q, \\ u_2 &= rw \mod q, \\ v &= (\alpha^{u_1}\beta^{u_2} \mod p) \mod q. \end{aligned}$$

If $v = r$ the signature is verified.

Proof. If the signature (r, s) is correct then

$$\begin{aligned} w &= k(m + ar)^{-1} \mod q, \\ u_1 &= km(m + ar)^{-1} \mod q, \\ u_2 &= kr(m + ar)^{-1} \mod q, \\ v &= \left(\alpha^{km(m+ar)^{-1}}\alpha^{akr(m+ar)^{-1}} \mod p\right) \mod q, \\ v &= \left(\alpha^{km(m+ar)^{-1}+akr(m+ar)^{-1}} \mod p\right) \mod q, \\ v &= \left(\alpha^k \mod p\right) \mod q, \\ v &= r. \end{aligned}$$

Let us remark that p, q, α can be common to a group of users.

The DSA algorithm needs two primes p and q such that

$$\begin{aligned} 2^{159} < q < 2^{160}, \\ 2^{k-1} < p < 2^k, \\ q|(p-1), \\ 512 \leq k = 512 + 64j \leq 1024. \end{aligned}$$

Let us give a sketch of a method to do that.

- **Getting q:** We first build at random a prime q between 2^{159} and 2^{160} (using a primality test).
- **Getting p:** Now we try to build at random a multiple $p-1$ of q such that p be a prime between 2^{k-1} and 2^k (we eventually do many tries). If this part fails, we restart with another q.

We refer to **FIPS 186-2** for going into detail.

3.2 RSA

The RSA function can be used also for signing. The system is the same as a RSA ciphersystem. But now a user wanting to sign a message **signs the digest** by enciphering it with **his private key**. This is the appendix part. The **verification** is done by deciphering the appendix part with **his public key**.

We don't insist more on the necessity of getting a careful protocol to generate and verify signatures. We refer to RSA document: **PKCS# 1 v2.1: RSA Cryptography Standard** for a complete description of **RSASSA-PSS** (**RSA S**ignature **S**cheme with **A**ppendix-**P**robabilistic **S**ignature **S**cheme). We just give the sequence of operations to do.

- **Encoding process**. The message (octet string) M is transformed in an octet string EM of length $emlen$. This bloc EM is the **encoded message**. The transformation is done using the EMSA-PSS **encoding** process. (EMSA: **E**ncoding **M**ethod for **S**ignature with **A**ppendix. PSS: **P**robabilistic **S**ignature **S**cheme).
- **To integer translation**. Now EM is translated to an integer m by OS2IP (Octet String To Integer Primitive).
- **Signature process** The integer m is transformed in s by RSASP1 (RSA Signature Primitive).
- **To octet string translation**. The integer s is translated in an octet string S by I2OSP (Integer To Octet String Primitive).

Now when the receiver get (M, S) he applies the verification scheme:

- **To integer translation**. S is translated to an integer s by OS2IP (Octet String To Integer Primitive).

- **Inverse signature part** The integer s is transformed in m by RSAVP1 (RSA Verification Primitive).
- **To octet string translation**. The integer m is translated in an octet string EM by I2OSP (Integer To Octet String Primitive).
- **Verification**. The EMSA-PSS **verification** function is applied to M and EM to determine whether they are consistent.

4 Key management

Key management deals with the keys generation (pairs public-private keys), the private key distribution (if not locally generated), the private key storage, the public keys selection storage and protection, keys revocation. The key management level is probably the weakest part of public key cryptosystems.

Theses problems are within the scope of a **PKI** (**P**ublic **K**ey **I**nfrastructure).

The first problem is to generate a pair of keys. Depending upon the PKI organization, this can be done locally or by a **KDC** (**K**ey **D**istribution **C**enter) which is a central server.

Then the public key must be registered with a **CA** (**C**ertifying **A**uthority). The CA gives to the user a **certificate**. The certificate attests to the binding of the key to the valid user. It contains the public key and the name of the user, the name of the **certifying authority** that issued the certificate, an expiration date and it is signed by the certificate issuer. Other users know the public key of the authority and so they can control the validity of the key. The CA's public key must be trustworthy. In a hierarchic organization, this can be done by a **certification hierarchy**. But in an open network the problem is harder.

The private key must be locally stored. The best is to store it in encrypted form on a media not accessible through a computer network.

The public key can in general be stored using the directory service of the CA.

Sometimes, a certificate has to be revoked before its scheduled expiration date (for example if the key is compromise). A **CRL** (**C**ertificate **R**evocation **L**ist) is maintened by the CA. Each user wanting a party's public key must control before using the key that the relevant certificate is not revoked.