Introduction to RSA

Modular Exponentiation, Key Generation (Standard and CRT)

Christophe Clavier

University of Limoges

Master 2 Cryptis

Description

The RSA cryptosystem was invented in 1977 by Rivest, Shamir and Adleman.
It is based on the factorisation problem.

Key: $\mathrm{pk} = \{n, e\}, \mathrm{sk} = d$, such that $ed \equiv 1 \pmod{\Phi(n)}$.

Encryption: $c = m^e \bmod n$

Decryption: $m = c^d \bmod n$

n is called the modulus.

e is the public exponent while d is the private exponent.

For classical 2-factor RSA, the modulus n is taken equal to pq, where p and q are primes of the same bitlength.

$\Phi(n) = (p-1)(q-1)$ is Euler's totient function.

Consistency

So, why does it work?

Because of Euler's theorem that says:

$$\forall m \in \mathbb{Z}_n^*, \quad m^{\Phi(n)} \equiv 1 \pmod{n}$$

so that

$$\forall m \in \mathbb{Z}_n^*, \quad c^d \equiv (m^e)^d \equiv m^{ed} \equiv m^{1+k\Phi(n)} \equiv m \pmod{n}$$

(remember that $ed \equiv 1 \pmod{\Phi(n)}$)

Typical Use Case

A classical use of RSA encryption scheme is as follows:

- The modulus is made of two prime factors of (almost) same bitlength, and the modulus is 1024 bits long.
- The public exponent is taken prime, and can be very small. Typical values are 3 or better $2^{16} + 1$. Taking a small public exponent is not less secure, and has some efficiency advantages.

Problem Statement

- Given a modulus n, a k-bit exponent $d = (d_{k-1}d_{k-2}\dots d_0)_2$ and an input $m < n$, we want to compute $m^d \bmod n$ in a secure and efficient way.
- Security should be considered with respect to side-channel analysis (SPA, DPA, CPA,...) and fault analysis.
- All methods proposed here are based on a sequence of $\mathcal{O}(k)$ modular multiplications. The average number of them is typically $c \cdot k$ $(1 \leqslant c \leqslant 2)$. Time efficiency is simply understood as reducing the constant c.

Modular computations

Usual integer multiplication methods have quadratic bit complexity. For efficiency purpose an implicit reduction modulo n will be performed after each multiplication of two k-bit integers. For example:

$$(a \times b \times c \times d) \bmod n$$

will be computed as

$$a \times (b \times (c \times d \bmod n) \bmod n) \bmod n$$

Left-to-right square-and-multiply

Principle

The left-to-right square-and-multiply exponentiation is based on the following expression:

$$d = d_0 + 2 \times (d_1 + 2 \times (\dots + 2 \times (d_{k-1}) \dots))$$

$$m^d = m^{d_0} \times \left(m^{d_1} \times \left(\dots \times \left(m^{d_{k-1}}\right)^2 \dots\right)^2\right)^2$$

Left-to-right square-and-multiply exponentiation

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1}d_{k-2}\dots d_0)_2$
Output: $m^d \bmod n$

1. $a \leftarrow 1$
2. **for** $i = k-1$ **to** 0 **do**
3. $\quad a \leftarrow a^2 \bmod n$
4. $\quad$ **if** $d_i = 1$ **then**
5. $\quad\quad a \leftarrow a \times m \bmod n$
6. **return** a

Left-to-right square-and-multiply

Exercise

Simulate the execution of the algorithm with $d = 3441 = (\mathrm{D71})_{16}$.
Write down the successive powers of m taken by register a.
Find how to express the value taken by a at the end of each loop.

Invariant property

At the end of each loop the following holds:

- $a = m^{(d_{k-1}d_{k-2}\ldots d_i)_2} \bmod n$

Complexity

- Time cost: $1S + 0.5M$ per exponent bit

Right-to-left square-and-multiply

Principle

The right-to-left square-and-multiply exponentiation is based on the following expression:

$$d = 2^{k-1} \times d_{k-1} + 2^{k-2} \times d_{k-2} + \ldots + 2 \times d_1 + d_0$$

$$m^d = \left(m^{2^{k-1}}\right)^{d_{k-1}} \times \left(m^{2^{k-2}}\right)^{d_{k-2}} \times \ldots \times \left(m^2\right)^{d_1} \times m^{d_0}$$

Right-to-left square-and-multiply exponentiation

Input: $m, n \in \mathbb{N}$, $m < n$, $d = (d_{k-1}d_{k-2}\ldots d_0)_2$
Output: $m^d \bmod n$

1. $a \leftarrow 1$
2. $b \leftarrow m$
3. **for** $i = 0$ **to** $k-1$ **do**
4.   **if** $d_i = 1$ **then**
5. $a \leftarrow a \times b \bmod n$
6. $b \leftarrow b^2 \bmod n$
7. **return** a

Right-to-left square-and-multiply

Exercise

Simulate the execution of the algorithm with $d = 9747 = (2613)_{16}$.
Write down the successive powers of m taken by registers a and b.
Find how to express the value taken by a at the end of each loop.

Invariant property

At the end of each loop the following holds:

- $a = m^{(d_i \dots d_1 d_0)_2} \bmod n$
- $b = m^{2^{i+1}} \bmod n$

Complexity

- Time cost: $1S + 0,5M$ per exponent bit

RSA Standard Key Generation

Usually the public exponent e is previously chosen as a small integer (typical values: 3, 5, 17, 257, $2^{16}+1$).

Choosing e small allows to encrypt efficiently.

Given e the following procedure generates a k-bit RSA key pair:

1. Generate at random a $k/2$-bit prime p such that $\gcd(e, p-1) = 1$
2. Generate at random a $k/2$-bit prime q such that $\gcd(e, q-1) = 1$
3. Compute $n = pq$, and $\Phi(n) = (p-1)(q-1)$
4. Compute $d = e^{-1} \bmod \Phi(n)$

Publish $\text{pk} = \{n, e\}$.

Keep secret $\text{sk} = \{d\}$. (and discard $\Phi(n), p, q$)

A Toy Example

Let's generate a 20-bit RSA key for the public exponent $e = 11$.

- $p = 547, \quad q = 797$
- $n = p \times q = 547 \times 797 = 435\,959$
- $\Phi(n) = (p-1) \times (q-1) = 546 \times 796 = 434\,616$
- $d = e^{-1} \bmod \Phi(n) = 11^{-1} \bmod 434\,616 = 355\,595$
  by means of extended Euclid's algorithm

Note that p and q are both 10-bit primes while n is only a 19-bit modulus !

Let's encrypt/decrypt the secret message $m = 123\,456$:

encryption $c = m^e \bmod n = 123\,456^{11} \bmod 435\,959 = 77\,111$

decryption $m = c^d \bmod n = 77\,111^{355\,595} \bmod 435\,959 = 123\,456$

Standard versus CRT Mode

RSA we just described is said in standard mode. This is the simplest way to use it.

Note that the encryption operation is efficient since e is small.

Conversely decryption is time consuming since d is always full size (about the size of n).

There is a computation trick that allows to make decryption faster. This is related to the Chinese Remainder Theorem, hence its name : CRT mode.

Chinese Remainder Theorem (CRT)

Chinese Remainder Theorem

Let $n_1, \dots, n_k$ be pairwise coprime positive integers, and $a_1, \dots, a_k$ be integers defined modulo $n_1, \dots, n_k$ respectively.

- There exists an integer a such that
$$a \equiv a_i \pmod{n_i} \quad (i = 1, \dots, k).$$
- Also, a' is also a solution to this equations system iff $a \equiv a' \pmod{N}$, where $N = \prod_{i=1}^{k} n_i$.

  Therefore, the solution to this congruences system is uniquely determined modulo the product of the moduli.

The Chinese remainder theorem expresses that there exists a one-to-one mapping between $\mathbb{Z}_N$ and $\mathbb{Z}_{n_1} \times \ldots \times \mathbb{Z}_{n_k}$.

In $\mathbb{Z}_N$, it is possible to represent the integer a by the tuple $(a_1, \dots, a_k)$ where for all i, $a_i = a \bmod n_i$.

Chinese Remainder Theorem (CRT)

Example

Example with $n_1 = 3$ and $n_2 = 5$ $(N = 15)$:

a	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
$a \bmod 3$	0	1	2	0	1	2	0	1	2	0	1	2	0	1	2
$a \bmod 5$	0	1	2	3	4	0	1	2	3	4	0	1	2	3	4

The integer 8 is represented by the couple $(2, 3)$.

Changing the Representation

From $\mathbb{Z}_N$ to $\mathbb{Z}_{n_1} \times \ldots \times \mathbb{Z}_{n_k}$

$$\forall i = 1, \ldots, k \quad a_i = a \bmod n_i$$

From $\mathbb{Z}_{n_1} \times \ldots \times \mathbb{Z}_{n_k}$ to $\mathbb{Z}_N$ (Gauss's method)

For all $i = 1, \ldots, k$, let $N_i := N/n_i$. Then:

$$a = \left(a_1 \cdot N_1 \cdot (N_1^{-1} \bmod n_1) + \ldots + a_k \cdot N_k \cdot (N_k^{-1} \bmod n_k)\right) \bmod N$$

From $\mathbb{Z}_{n_1} \times \ldots \times \mathbb{Z}_{n_k}$ to $\mathbb{Z}_N$ (Garner's method)

Particular case $k = 2$:

$$a = a_1 + n_1 \cdot \left((a_2 - a_1) \cdot (n_1^{-1} \bmod n_2) \bmod n_2\right)$$

This method can be generalised for $k > 2$. It consists in computing the successive digits of a in *multi-radix* base $(n_1, n_2, \ldots, n_k)$.

CRT Mode

Remember that $n = pq$. From the Chinese Remainder Theorem, m can be evaluated as $(m_p, m_q) \in \mathbb{Z}_p \times \mathbb{Z}_q$. (Instead of being computed directly in $\mathbb{Z}_n$.)

$$\begin{aligned}
m_p &= m \bmod p \\
&= (c^d \bmod n) \bmod p \\
&= c^d \bmod p \\
&= c^{d \bmod (p-1)} \bmod p \quad \text{since } c^{p-1} \equiv 1 \pmod p \quad \text{(Fermat)} \\
m_p &= c^{d_p} \bmod p \qquad (\text{with } d_p := d \bmod (p-1))
\end{aligned}$$

A similar computation leads to $m_q = c^{d_q} \bmod q$ (with $d_q := d \bmod (q-1)$)

From Garner's formula, $m \in \mathbb{Z}_n$ is recovered as :

$$m = m_p + p \cdot ((m_q - m_p) \cdot I_p \bmod q)$$

where $I_p = p^{-1} \bmod q$.

CRT Mode

This is a lot more computations ! How would it be more efficient ?

- Modular exponentiation on k-bit operands is $\mathcal{O}(k^3)$.
- There are two modular exponentiations (modulo p and q) in CRT mode, but each one computes on $k/2$-bit operands.
- These exponentiations are thus eight times faster than $c^d \bmod n$!
- Assuming the final Garner's recombination time is negligible, CRT mode decryption achieves a four times speed-up compared to standard mode.

RSA CRT Key Generation

Given e the following procedure generates a k-bit RSA key pair usable in CRT mode:

1. Generate at random a $k/2$-bit prime p such that $\gcd(e, p-1) = 1$
2. Generate at random a $k/2$-bit prime q such that $\gcd(e, q-1) = 1$
3. Compute $n = pq$
4. Compute $d_p = e^{-1} \bmod (p-1)$
5. Compute $d_q = e^{-1} \bmod (q-1)$
6. Compute $I_p = p^{-1} \bmod q$

Publish $\text{pk} = \{n, e\}$.

Keep secret $\text{sk} = \{p, q, d_p, d_q, I_p\}$.

A Toy Example

Let's generate a 20-bit RSA CRT key for the public exponent $e = 11$.

- $p = 547, \quad q = 797$
- $n = p \times q = 547 \times 797 = 435\,959$
- $d_p = e^{-1} \bmod (p-1) = 11^{-1} \bmod 546 = 149$
- $d_q = e^{-1} \bmod (q-1) = 11^{-1} \bmod 796 = 579$
- $I_p = p^{-1} \bmod q = 547^{-1} \bmod 797 = 424$

Let's encrypt and CRT decrypt the secret message $m = 123\,456$:

encryption $c = m^e \bmod n = 123\,456^{11} \bmod 435\,959 = 77\,111$

decryption
$m_p = c^{d_p} \bmod p = 77\,111^{149} \bmod 547 = 381$
$m_q = c^{d_q} \bmod q = 77\,111^{579} \bmod 797 = 718$
$m = 381 + 547 \cdot ((718 - 381) \cdot 424 \bmod 797) = 123\,456$

Université de Limoges