Security Aspects of RSA Implementations

Christophe Clavier

University of Limoges

Master 2 Cryptis

RSA Description

The RSA public key cryptosystem has been invented in 1977 by Rivest, Shamir and Adleman. It is based on the integer factorisation problem.

RSA key elements

Public key $pk = \{n, e\}$:

- The modulus n is the product of two large secret primes p et q
- The public exponent e is used in the public operation: encryption or signature verification

Private key $pk = \{d\}$:

- The private exponent d is used in the private operation: decryption or signature computation

RSA Description

RSA use for encryption

Given an encoded confidential message $m < n$:

encryption $c = m^e \bmod n$

decryption $m = c^d \bmod n$

RSA use for signature

Given an message m to be signed and a hash function $\mathcal{H}$:

signature $s = \mathcal{H}(m)^d \bmod n$

verification $\mathcal{H}(m) \overset{?}{=} s^e \bmod n$

Consistency

So, why does it work?

Because of Euler's theorem that says:

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

so that

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

(notice 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 at least 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.

RSA Key Generation in Standard Mode

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

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

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 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 Mode 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.

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 $\mathbb{Z}_n$.)

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

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 :

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

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 Key Generation in CRT Mode

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

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

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

From the modulus factors to the private exponent

Question

How to compute d from p and q?

Answer

1. $\Phi(n) = (p-1)(q-1)$
2. $d = e^{-1} \bmod \Phi(n)$

This is why RSA security is said to be based on the difficulty to factorise large integers.

From d to $\Phi(n)$

Question

How to recover $\Phi(n)$ from d?

Answer

$$\begin{aligned} d &= e^{-1} \bmod \Phi(n) \\ e\,d &\equiv 1 \pmod{\Phi(n)} \\ e\,d - 1 &= k\,\Phi(n) \quad \text{(for some } k \in \{1,\ldots,e-1\}) \end{aligned}$$

Note that $\Phi(n)$ is extremely close to n. Indeed:

$$\frac{\Phi(n)}{n} = \frac{(p-1)(q-1)}{n} = \frac{pq-(p+q)+1}{n} = 1+\epsilon \quad \text{with } \epsilon \approx \frac{1}{\sqrt{n}}$$

When e is small, k is equal to $\lceil \frac{ed-1}{n} \rceil$. $\Phi(n)$ is then recovered as $\Phi(n) = \frac{ed-1}{k}$.

From $\Phi(n)$ to the prime factors

Question

How to recover p and q from $\Phi(n)$?

Answer

Notice that we know the sum S and the product P of the primes:

- $S = p + q = (n+1) - \Phi(n)$
- $P = p\,q = n$

Thus p and q are the solutions of the quadratic equation: $x^2 - S\,x + P = 0$ which is solved as:

$$\begin{aligned} \Delta &= S^2 - 4P \\ p &= \frac{S + \sqrt{\Delta}}{2} \\ q &= \frac{S - \sqrt{\Delta}}{2} \end{aligned}$$

From d_p to p (CRT mode)

Question

How to recover p from d_p?

Answer

$$\begin{aligned} d_p &= e^{-1} \bmod (p-1) \\ e\,d_p &\equiv 1 \pmod{p-1} \\ e\,d_p - 1 &= k\,(p-1) \quad (\text{for some } k \in \{1, \ldots, e-1\}) \end{aligned}$$

This seems like the standard case, except that we can not approximate $(p-1)$.

Assume that we can factorise $(e\,d_p - 1)$, at least partly. Then we exhaust all divisors δ which could be a candidate for k until $\frac{e\,d_p - 1}{\delta} + 1$ divides n.

If e is small, this is always easily feasible. If e is full size then a full factorisation of $(e\,d_p - 1)$ will be required in most cases which may be not possible.

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 typically ranges between k and $2k$. Time efficiency is simply understood as reducing the constant.

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.     $a \leftarrow a^2 \bmod n$
4.     if $d_i = 1$ then
5.         $a \leftarrow a \times m \bmod n$
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

Signed Digit Representation

Normal binary representation : $d = \Sigma_{i=0}^{k-1} d_i 2^i$, with $d_i \in \{0, 1\}$
Signed digit binary representation : $d = \Sigma_{i=0}^{k-1} d_i 2^i$, with $d_i \in \{-1, 0, 1\}$

Example

$d = 7975 = (1111100100111)_2 = (10000\bar{1}00101001\bar{1})_2$ ($\bar{1}$ stands for -1)

Non adjacent form (NAF)

- The signed digit representation of an integer d is not unique
- There always exists a unique canonical representation of d with the property that two adjacent digits are not both non-zero
- This canonical representation is called the *Non Adjacent Form* (NAF) of d, and denoted $(d)_{\text{NAF}}$ or NAF(d)
- The length of NAF(d) is either the same of that of d, or just one more
- The average Hamming weight of the NAF of all k-bit integers is $\frac{k}{3}$

Computing a NAF Representation

NAF representation computation

Input: $d \in \mathbb{N}^*$
Output: $(d)_{\mathrm{NAF}}$

```
1.  $i \leftarrow 0$
2.  while $d \geq 1$ do
3.      if $d \bmod 2 = 1$ then
4.          $d_i \leftarrow 2 - (d \bmod 4)$
5.          $d \leftarrow d - d_i$
6.      else
7.          $d_i \leftarrow 0$
8.      $d \leftarrow d/2$
9.      $i \leftarrow i + 1$
10. return $(d_{k-1} d_{k-2} \dots d_0)$
```

Exercise

Compute the NAF of $7318 = (1C96)_{16}$

Signed Digit Exponentiation

Left-to-right square-and-multiply signed digit exponentiation

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

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

Complexity

- Time cost: $1S + \frac{1}{3}M$ per exponent bit
- Note that an inverse computation ($m^{-1} \bmod n$) is required. This costly operation makes the signed digit exponentiation useless for RSA.

Motivation

Simple Power Analysis can threaten exponentiations if no special attention has been paid to the implementation.

This is particularly true when the sequence of modular operations allows to infer exponent bits by distinguishing squarings from multiplications.

Regular exponentiations

Numerous exponentiation methods have been designed to present a regular sequence of operations to the adversary:

- left-to-right square-and-multiply-always
- right-to-left square-and-multiply-always
- the Montgomery ladder
- the Joye ladder
- the atomic exponentiation
- square-always methods

Left-to-right square-and-multiply-always

Left-to-right square-and-multiply-always exponentiation (version 1)

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

Complexity

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

Left-to-right square-and-multiply-always

Left-to-right square-and-multiply-always exponentiation (version 2)

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. $R_0 \leftarrow 1$
2. for $i = k-1$ to $0$ do
3.     $R_0 \leftarrow R_0^2 \bmod n$
4.     $R_{1-d_i} \leftarrow R_0 \times m \bmod n$
5. return $R_0$
```

Complexity

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

Left-to-right square-and-multiply-always

Question

- What is the difference between the two versions?
- Which one is more secure?

Answer

- Version 1 makes use of a conditional branching which may be noticeable on the side channel trace.
- Version 2 should be preferred as the selection of the destination register is based on a register index (bit) which is likely to be more difficult to detect than instruction addresses difference.

Left-to-right square-and-multiply-always

Common vulnerability

- Note that both versions are vulnerable to a fault attack named *safe error analysis*. In this attack the adversary induces a computation error during a multiplication operation (not a square) which results in a modification of the destination register value.
- Whether this data modification is propagated through the final output reveals the value d_i of the exponent bit at the fault iteration.

Right-to-left square-and-multiply-always

Both left-to-right versions of the square-and multiply-always can be adapted in a straightforward manner to a scan of the exponent from right to left.

Right-to-left square-and-multiply-always exponentiation (version 1)

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

Complexity

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

Right-to-left square-and-multiply-always

Right-to-left square-and-multiply-always exponentiation (version 2)

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. $R_0 \leftarrow 1$
2. $R_1 \leftarrow 1$
3. $R_2 \leftarrow m$
4. for $i = 0$ to $k-1$ do
5.     $R_{1-d_i} \leftarrow R_{1-d_i} \times R_2 \bmod n$
6.     $R_2 \leftarrow R_2^2 \bmod n$
7. return $R_0$
```

Complexity

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

Right-to-left square-and-multiply-always

Exercise

- Simulate the execution of the algorithm (either version) with the 10-bit exponent $d = 985 = (3D9)_{16}$.
  Write down the successive powers of m taken by the three registers.
- Repeat the same with $d = 623 = (26F)_{16}$.
- Can you notice something interesting?

Question

- Are these exponentiation methods susceptible to safe error analysis?

Answer

- Yes but. . . no!
- As noticed in the previous exercise, final values of the three registers a, b and c (or R_0, R_1, R_2) always verify : $c = (a \times b \times m) \bmod n$.
  This can be used as a consistency check to prevent safe error analysis.

The Montgomery Ladder

Montgomery ladder 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. $R_0 \leftarrow 1$
2. $R_1 \leftarrow m$
3. for $i = k-1$ to $0$ do
4.     $R_{1-d_i} \leftarrow R_0 \times R_1 \bmod n$
5.     $R_{d_i} \leftarrow {R_{d_i}}^2 \bmod n$
6. return $R_0$
```

Exercise

Simulate the execution of the algorithm with $d = 1419 = (58B)_{16}$.
Write down the successive powers of m taken by registers R_0 and R_1.
Find how to express the value taken by R_0 at the end of each loop.
Find a particular relation between R_0 and R_1 at the end of each loop.

The Montgomery Ladder

Invariant properties

At the end of each loop the following holds:

- $R_0 = m^{(d_{k-1}d_{k-2}\ldots d_i)_2} \bmod n$
- $R_1 = R_0 \times m \bmod n$

The check of this last property can be used as a countermeasure against fault attacks.

Complexity

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

The Joye Ladder

Joye ladder may be viewed as a simple and clever right-to-left counterpart of the Montgomery ladder.

Joye ladder 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. $R_0 \leftarrow 1$
2. $R_1 \leftarrow m$
3. for $i = 0$ to $k-1$ do
4.     $R_{1-d_i} \leftarrow R_{1-d_i}^2 \times R_{d_i} \bmod n$
5. return $R_0$
```

Exercise

Simulate the execution of the algorithm with $d = 839 = (347)_{16}$.
Write down the successive powers of m taken by registers R_0 and R_1.
Find how to express the value taken by R_0 at the end of each loop.
Find a particular relation between R_0 and R_1 at the end of each loop.

The Joye Ladder

Invariant properties

At the end of each loop the following holds:

- $R_0 = m^{(d_i \ldots d_1 d_0)_2} \bmod n$
- $R_0 \times R_1 \equiv m^{2^{i+1}} \pmod{n}$

Here, this last property can not be checked to detect fault attacks.
Can you see why?

Complexity

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

Atomic Principle

Atomic Principle

Replacing all squarings by multiplications allows an exponentiation method where a unique (atomic) instructions pattern is repeatedly executed

This leads to an exponentiation which mimics the basic *left-to-right square-and-multiply* but using only multiplication instructions.

Atomic Principle

Multiply-always atomic 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. $R_0 \leftarrow 1$
2. $R_1 \leftarrow m$
3. $i \leftarrow k-1$
4. $t \leftarrow 0$
5. while $i \geq 0$ do
6.     $R_0 \leftarrow R_0 \times R_t \bmod n$
7.     $t \leftarrow t \oplus d_i$
8.     $i \leftarrow i - 1 + t$
9. return $R_0$
```

Complexity

- Time cost: $1.5M$ per exponent bit

Left-to-right 2^t-ary square-and-multiply

Principle

Instead of scanning the exponent bitwise the 2^t-ary method scans it t bits at a time. It is thus based on the representation of the exponent in base 2^t:

$$d = (d_{k-1}d_{k-2}\ldots d_1d_0)_{2^t}$$

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

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

- Note that in this representation, the number k of digits in base 2^t is t times less than the bitlength of the exponent.
- The principle is quite similar to the binary case, except that one must pre-compute values $m^j \bmod n$ for all $j = 2, \ldots, 2^t - 1$.

Left-to-right 2^t-ary square-and-multiply

Left-to-right 2^t-ary square-and-multiply exponentiation

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

```
1.  $S \leftarrow 1$
2.  $R_0 \leftarrow 1$
3.  $R_1 \leftarrow m$
4.  for $j = 2$ to $2^t - 1$ do
5.      $R_j \leftarrow R_{j-1} \times m \bmod n$
6.  for $i = k - 1$ to $0$ do
7.      for $u = 1$ to $t$ do
8.          $S \leftarrow S^2 \bmod n$
9.      $S \leftarrow S \times R_{d_i} \bmod n$
10. return $S$
```

Complexity

- Time cost: $1S + \frac{1}{t}M$ per exponent bit
  (ignoring pre-computations and possibly saved first t squarings)

Left-to-right 2^t-ary square-and-multiply

Invariant property

At the end of each loop the following holds: $S = m^{(d_{k-1}d_{k-2}\ldots d_i)_{2^t}} \bmod n$

Computation tricks

- It is possible to save multiplications by $R_0 = 1$ but this is little gain and exposes the positions of zero digits in the exponent if squarings can be distinguished from multiplications by SPA.
- It is possible to save the first t squarings by initialising $S \leftarrow R_{d_{k-1}}$ and starting the loop at index $i = k - 2$.
- It is possible to divide by two the storage requirement. Indeed it is sufficient to pre-compute values m^j for odd j.
  At each iteration of the main loop, a non zero d_i can be expressed as $s \cdot 2^v$ with s odd and $0 \leq v \leq t - 1$. One just have to early multiply the accumulator S by m^s after only $(t - v)$ squarings and compute the v remaining ones after this multiplication.
  Note that the square-and-multiply sequence is then no more regular!

Left-to-right 2^t-ary square-and-multiply

Security tricks

- Even if present the multiplication by $R_0 = 1$ is possibly unsecure because of its very low Hamming weight.
  It is common practice to initialise $R_0 \leftarrow n + 1$.
- Values of other registers are also susceptible to the same kind of SPA in the case where the adversary can input an arbitrary small m.
  For that reason it may be a good idea to initialise $R_1 \leftarrow m + n$, and $R_j \leftarrow (R_{j-1} \times m \bmod n) + n$.

Right-to-left variant

There exists a right-to-left variant of the 2^t-ary square-and-multiply which requires post-computations instead of pre-computations.

Blinding Schemes

Principle

- Blinding schemes allow to prevent some side-channel attacks on the modular exponentiation.
- The basic principle is to randomly modify the operands of the exponentiation in a way that allows to easily recover to correct result from the output of the randomized exponentiation.

There are different ways to blind the exponentiation computations:

- Exponent blinding
- Modulus blinding
- Message blinding

Exponent Blinding in Standard Mode

From Euler's theorem we have:

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

The consequence is that adding $\Phi(n)$ (or any multiple thereof) to the exponent of a exponentiation modulo n do not modify the result.

Indeed we have:

$$\forall m \in \mathbb{Z}_n^\star, \quad m^{d+k\Phi(n)} \equiv m^d (m^{\Phi(n)})^k \equiv m^d \pmod{n}$$

Thus, blinding the exponent simply consists in replacing d by $d^\star = d + r \cdot \Phi(n)$ where the integer r is picked at random at each execution.

Exponent Blinding in Standard Mode

$$d^\star = d + r \cdot \Phi(n)$$

Efficiency

- Note that as $\Phi(n)$ has about the same bitlength than that of d, the blinded exponent $d^\star$ is expanded compared to d by the bitlength of the random r.
- This induces an linear extra cost due to the extra bitlength of the exponent.

Security

- The purpose of blinding the exponent is to thwart attacks which require several side-channel traces.
- As knowing $d^\star$ is equivalent to knowing d, exponent blinding is useless if an attacker is able to recover the exponent from a unique trace.

Exponent Blinding in CRT Mode

As a particular case of Euler's theorem when the modulus is prime, little Fermat's theorem states:

$$\forall m \not\equiv 0 \pmod{p}, \quad m^{p-1} \equiv 1 \pmod{p}$$

As in standard mode, the consequence is the opportunity to randomize exponents d_p and d_q used in CRT mode as:

$$\begin{aligned} d_p^\star &= d_p + r_p \cdot (p-1) \\ d_q^\star &= d_q + r_q \cdot (q-1) \end{aligned}$$

Here also there is an extra cost related to the expansion of the exponent.

Exponent Blinding in CRT Mode

Security

- The purpose of blinding the CRT exponents is to thwart attacks which require several side-channel traces.
- Knowing some $d_p^\star$ and/or $d_q^\star$ value does not directly allow an adversary to compute a signature if he does not know other CRT key elements.
- However, only one randomized exponent (say $d_p^\star$) is sufficient to recover p. As in standard mode, CRT exponent blinding is useless when the exponent can be recovered with only one trace.

Recovering p from $d_p^\star$

For any m compute $\mu = (m^e \bmod n)^{d_p^\star} \bmod n$. We have:

$$\mu \equiv m^{ed_p^\star} \equiv m^{1+k(p-1)} \equiv m \pmod p$$

Just recover p as $\gcd(\mu - m, n)$.

Modulus Blinding in Standard Mode

- Modulus blinding consists in replacing n by $n^\star = r \cdot n$ for some random integer r.
- After computing $m^d \bmod n^\star$ it is sufficient to reduce the result modulo n to obtain the correct signature.

Efficiency

- For a given random bitlength λ, the extra cost due to the modulus expansion is greater than it is for exponent blinding.
- Exponent blinding implies an $\mathcal{O}((k+\lambda)k^2)$ exponentiation complexity while it is $\mathcal{O}(k(k+\lambda)^2)$ for modulus blinding.

Security

- The purpose of blinding the modulus is to thwart attacks where the adversary must predict intermediate values during the exponentiation.

Message Blinding in Standard Mode

- Together with modulus blinding, it is also possible to randomise the message. In that case the exponentiation is computed as:

$$(m + r_1 \cdot n)^d \bmod (r_2 \cdot n)$$

- The result must still be reduced modulo n at the end.
- Message blinding has no effect if not associated with modulus blinding.

Efficiency

- Message blinding does not imply any extra cost for the exponentiation compared to modulus blinding.

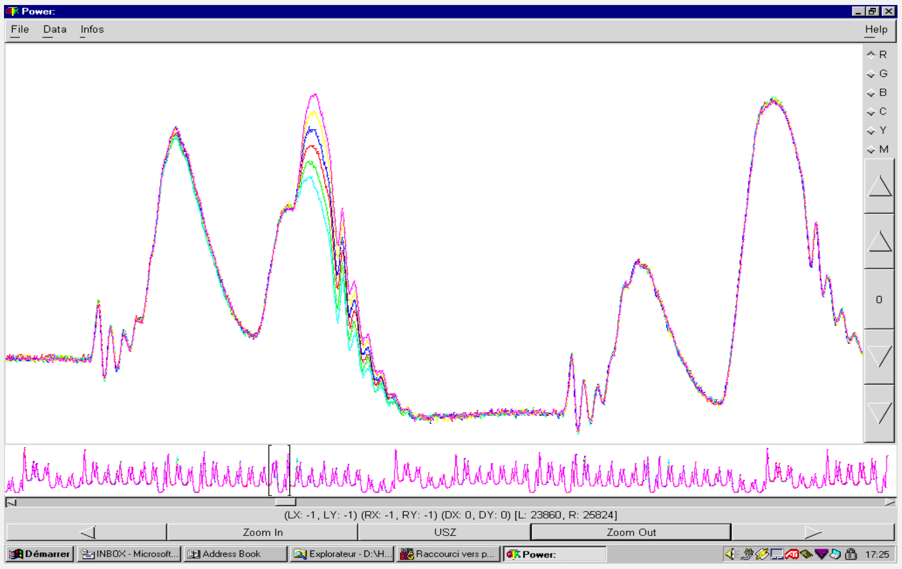
Security

- As for modulus blinding, message blinding prevents attacks where the adversary must predict intermediate values during the exponentiation.
- Message blinding strengthen modulus blinding in case of a leakage related to a multiplication operand rather than to its result.

Power Leakage

- The power consumption of a chip depends on:
  - The executed instruction
  - The manipulated data
- Leakage models
  - Hamming weight of whatever data put on the bus: data, address, operation code, . . .
    - $W = a \cdot \mathrm{HW}(data) + b$
  - Hamming distance (bus transition weight) w.r.t. a reference state
    - $W = a \cdot \mathrm{HD}(data_t, RF) + b \ = \ a \cdot \mathrm{HW}(data_t \oplus RF) + b$
    - RF : $data_{t-1}$ or $data_{t+1}$
  - Other models, chip & technologies, . . .

Example of Hamming Weight Leakage



(LX: -1, LY: -1) (RX: -1, RY: -1) (DX: 0, DY: 0) [L: 23860, R: 25824]

Simple Power Analysis

RSA computation requires arithmetic operations on large integer operands

On some crypto-coprocessors, the power consumption may depend on the type of arithmetic operation performed.

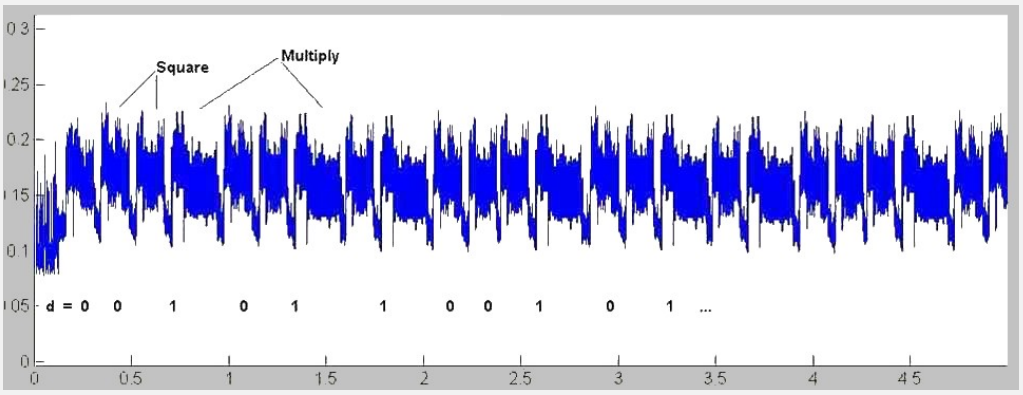
Simple Power Analysis on RSA exponentiation basically aims at distinguishing squarings from multiplications in the sequence of modular operations.

Example of the left-to-right square-and-multiply exponentiation:

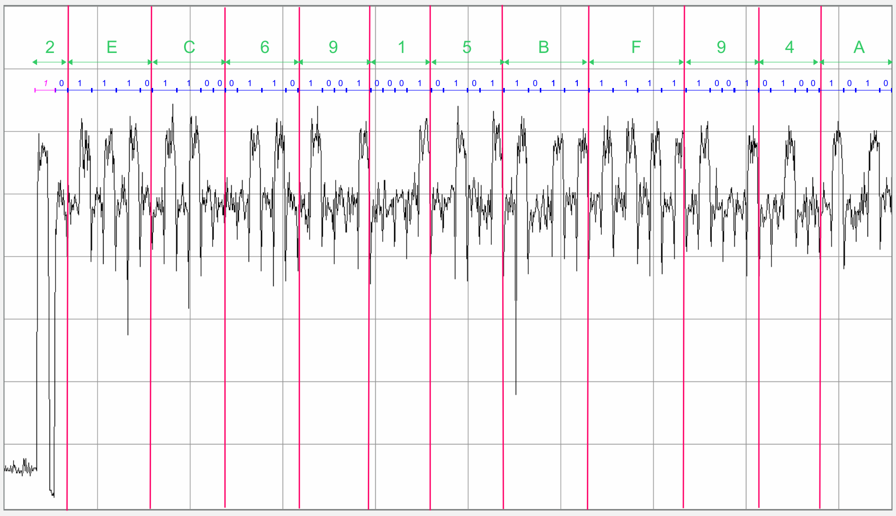
```
$a \leftarrow 1$
**for** $i = k - 1$ **to** 0 **do**
    $a \leftarrow a^2 \bmod n$
    **if** $d_i = 1$ **then**
        $a \leftarrow a \times m \bmod n$
**return** $a$
```

An SPA Example

The following power trace allows to readily recover successive exponent bits:



Try to Find the Private Key!



d = 0x 2E C6 91 5B F9 4A

Differential Power Analysis

What is it about?

Differential Power Analysis is a mean to isolate and enhance the tiny contribution of an arbitrary bit (belonging to an manipulated data) on a large set of power consumption traces.

For what usage?

Differential Power Analysis comes in two flavours:

- DPA on *known* data: allows to identify locations on the power trace where a known data is processed $\longrightarrow$ caracterisation, reverse engineering
- DPA on *key dependant* data: results in an hypothesis test used to identify the value of an exponent bit $\longrightarrow$ key recovery

Differential Power Analysis

- DPA performs a statistical analysis of a large number of power traces with known inputs.
- DPA on RSA allows to recover exponent bits one at a time in the same order they are processed in the exponentiation loop.
- Assume a left-to-right square-and-multiply exponentiation. Knowing a first part $t = (d_{k-1}d_{k-2}\dots d_{i+1})_2$ of the exponent, the attacker knows that the intermediate values $v_0 = (m^{2t})^2 \bmod n$ and $v_1 = (m^{2t+1})^2 \bmod n$ are computed during the exponentiation if and only if next exponent bit d_i is respectively equal to 0 and 1.
- Consider an arbitrary bit of the two virtual intermediate values v_0 and v_1 (say w.l.o.g. the least significant bit). One can compute the two series of values taken by this *selection bit* for all known input messages.

Differential Power Analysis

- In the Hamming weight model, one expects that the power consumption of the intermediate value is higher if the selection bit is 1 than if it is 0.
- For the hypothesis $d_i = 0$, the attacker splits the set of traces into two subsets $T^{(0)}_{d_i=0}$ and $T^{(1)}_{d_i=0}$ according to the value of the selection bit of v_0. The average power consumption of the traces of $T^{(1)}_{d_i=0}$ should be higher than that of $T^{(0)}_{d_i=0}$ at the time where the intermediate value is supposed to be manipulated.
- The attacker computes the DPA trace $\Delta_{d_i=0}$ defined for any time sample as the difference of the average consumption of subsets $T^{(1)}_{d_i=0}$ and $T^{(0)}_{d_i=0}$. If the hypothesis $d_i = 0$ is correct then the DPA trace $\Delta_{d_i=0}$ should show a so-called DPA peak at the precise instant of manipulation of v_0.
- The attacker also computes the DPA trace $\Delta_{d_i=1}$ and recovers the value of d_i according to which DPA trace exhibits a peak.

Differential Power Analysis

Question

- Does this attack also apply on the left-to-right square-and-multiply-always exponentiation?

Questions

- Does exponent blinding prevent this attack?
- Does modulus blinding prevent this attack?
- Does message blinding prevent this attack?

Correlation Power Analysis

- With DPA the influence of only one bit is considered for statistically distinguish between the two hypothesis.
- The advantage is that the Hamming weight model is not strictly required. The drawback is a poor signal to noise ratio.
- If the Hamming weight model is assumed then it is possible for the attacker to compute the Hamming weight of the whole virtual intermediate values v_0 and v_1, that is k bits instead of only one.
- One then have two series of Hamming weights $\{\mathrm{HW}(v_0(m))\}_m$ and $\{\mathrm{HW}(v_1(m))\}_m$ and the attacker should decide which one is representative of the actual processing in the device.
- The statistical tool used is the Pearson correlation coefficient between the series of Hamming weights and the series of power leakages at some time sample.
- A CPA trace is computed for each exponent bit value, and the attacker selects the one presenting a CPA peak.

Average Hamming Weight Analysis

In 2008, Amiel et al. observed that the average Hamming weight of a multiplication result differs whether this is a $(x \times y)$ or a $(x \times x)$ operation.

This leads to a statistical analysis which threatens the *multiply-always atomic exponentiation* when many traces may be acquired with random (or at least varying) inputs and fixed exponent.

Multiply-always exponentiation at instruction level

d	1		1		0	0	1		0	...
m_1	M	M	M	M	M	M	M	M	M	...
m_2	M	M	M	M	M	M	M	M	M	...
m_3	M	M	M	M	M	M	M	M	M	...
...					...					

Average Hamming Weight Analysis

Multiply-always exponentiation at data level

d	1		1		0	0	1		0	...
m_1	S	M	S	M	S	S	S	M	S	...
m_2	S	M	S	M	S	S	S	M	S	...
m_3	S	M	S	M	S	S	S	M	S	...
...					...					

Assuming a Hamming weight leakage model, averaging many side-channel traces results in successive $\langle \mathrm{HW}(S) \rangle$ and $\langle \mathrm{HW}(M) \rangle$ leakages which are distinguishable from each others.

Questions

- Does exponent blinding prevent this attack?
- Does modulus blinding prevent this attack?
- Does message blinding prevent this attack?

The Doubling Attack

- This attack relies on the assumption that an adversary can detect collisions of side-channel trace segments if a same operation is performed twice by the device
- The attacker does not have to identify which values are manipulated
- He is supposed to distinguish a collision between the processing of X^2 and Y^2 if $X = Y$, but not to guess the value of X
- The attack needs only two traces with chosen messages m and m^2

- Assume an exponentiation with the left-to-right square-and-multiply-always method and the exponent $d = 151 = (10010111)_2$
- The next table shows the sequences of operations performed by the algorithm for both messages m and m^2
- When a 0 exponent bit is processed at step i, the squaring of the exponentiation $(m^2)^d \bmod n$ and that of $m^d \bmod n$ at the next step $(i-1)$ are the same operation

The Doubling Attack – Example

i	d_i	Computation of $m^d \bmod n$	Computation of $(m^2)^d \bmod n$
7	1	1×1 $1 \times m$	1×1 $1 \times m^2$
6	**0**	$m \times m$ $m^2 \times m$	$\mathbf{m^2} \times \mathbf{m^2}$ $m^4 \times m^2$
5	**0**	$\mathbf{m^2} \times \mathbf{m^2}$ $m^4 \times m$	$\mathbf{m^4} \times \mathbf{m^4}$ $m^8 \times m^2$
4	1	$\mathbf{m^4} \times \mathbf{m^4}$ $m^8 \times m$	$m^8 \times m^8$ $m^{16} \times m^2$
3	**0**	$m^9 \times m^9$ $m^{18} \times m$	$\mathbf{m^{18}} \times \mathbf{m^{18}}$ $m^{36} \times m^2$
2	1	$\mathbf{m^{18}} \times \mathbf{m^{18}}$ $m^{36} \times m$	$m^{36} \times m^{36}$ $m^{72} \times m^2$
1	1	$m^{37} \times m^{37}$ $m^{74} \times m$	$m^{74} \times m^{74}$ $m^{148} \times m^2$
0	1	$m^{75} \times m^{75}$ $m^{150} \times m$ $= m^{151}$	$m^{150} \times m^{150}$ $m^{300} \times m^2$ $= (m^2)^{151}$

The Doubling Attack

Exercise

- Simulate the exponentiation on m and m^2 with $d = 151$ in the case of the Montgomery ladder method
- Does the doubling attack still apply?

Questions

- Does exponent blinding prevent this attack?
- Does modulus blinding prevent this attack?
- Does message blinding prevent this attack?

CPA Attack on the CRT Recombination

The CRT recombination

Computing an RSA signature in CRT mode needs the following key elements:

- p and q (prime factors of the modulus)
- $d_p = d \bmod (p-1)$ and $d_q = d \bmod (q-1)$ (reduced private exponents)
- $I_p = p^{-1} \bmod q$

The signature is computed in three steps:

1. $s_p = \mathcal{H}(m)^{d_p} \bmod p$
2. $s_q = \mathcal{H}(m)^{d_q} \bmod q$
3. $s = ((s_q - s_p) \cdot I_p \bmod q) \times p + s_p$

Note that the recombination computation can be reformulated as: $s = x \cdot p + s_p$ where x is an intermediate value processed during the computation.

CPA Attack on the CRT Recombination

Principle of the attack

$$s = x \cdot p + s_p$$

- As s_p is half size, s and $x \cdot p$ have their $k/2$ leftmost bits in common.
- One can use the public signature s as a good approximation of $x \cdot p$.
- For each candidate g about p one is able to compute $\frac{s}{g}$ which is a candidate approximation of the intermediate x.
- Computing the series of $\mathrm{HW}(x)$ for many different signatures, it is possible to validate each guess g by means of CPA. (assuming a Hamming weight model)
- As the set of p candidates is too large to be exhausted one uses a divide-and-conquer approach where p is recovered t bits at a time.
  At each step of the attack there are only 2^t guesses to consider, each guess allowing to predict t more leftmost bits of x.

CPA Attack on the CRT Recombination

Countermeasure

- It is possible to prevent this CPA with modulus blinding of each exponentiation:
  - $s_p^\star = \mathcal{H}(m)^{d_p} \bmod p^\star$ (with $p^\star = r_p \cdot p$)
  - $s_q^\star = \mathcal{H}(m)^{d_q} \bmod q^\star$ (with $q^\star = r_q \cdot q$)
- Warning: care must be taken not to remove the blinding before the recombination.
- Instead, the recombination is computed on blinded "half" signatures $s_p^\star$ and $s_q^\star$ and then the overall blinding is removed by reducing modulo n.

CPA Attack on the CRT Recombination

Bad blinding implementation

```
1. $p^\star \leftarrow r_p \cdot p$
2. $s_p^\star \leftarrow \mathcal{H}(m)^{d_p} \bmod p^\star$
3. $s_p \leftarrow s_p^\star \bmod p$
4. $q^\star \leftarrow r_q \cdot q$
5. $s_q^\star \leftarrow \mathcal{H}(m)^{d_q} \bmod q^\star$
6. $s_q \leftarrow s_q^\star \bmod q$
7. $s \leftarrow ((s_q - s_p) \cdot I_p \bmod q) \times p + s_p$
8. return $s$
```

Correct blinding implementation

```
1. $p^\star \leftarrow r_p \cdot p$
2. $s_p^\star \leftarrow \mathcal{H}(m)^{d_p} \bmod p^\star$
3. $q^\star \leftarrow r_q \cdot q$
4. $s_q^\star \leftarrow \mathcal{H}(m)^{d_q} \bmod q^\star$
5. $s^\star \leftarrow \left((s_q^\star - s_p^\star) \cdot I_p \bmod q^\star\right) \times p^\star + s_p^\star$
6. $s \leftarrow s^\star \bmod n$
7. return $s$
```

Some other known side-channel attacks on RSA

Other attacks

There exist some other known attacks on RSA, amongst which:

- The Big Mac attack where the multiplications by m are identified by means of a kind of template analysis. It applies also on 2^t-ary exponentiations. Message and modulus blinding prevents this attack.
- The horizontal correlation power analysis where correlations are computed on trace segments of a single curve to distinguish between squarings and multiplications.

Useful sources of further information:

- The side-channel cryptanalysis lounge: `http://imperia.rz.rub.de:9085/en_sclounge.html`
- Cryptology ePrint archive: `http://eprint.iacr.org/`