Side Channel Attack On Smart Cards
Timing Attack On Rsa-Montgomery

CHAHI, Rabie Ala-Eddine et KESKES, Yasmine
Master : SeCReT
Enseignant : Mr Goubin Louis

3 Juin 2020

UNIVERSITÉ DE VERSAILLES ST-QUENTIN-EN-YVELINES
université PARIS-SACLAY

Sommaire

1 Avant-propos :

"Sometimes it is the people no one can imagine anything of who do the things no one can imagine." — Alan Turing

1.1 But :

Nous essayons dans ce document de vulgariser la timing attack sur RSA de telle sorte que toute personne ayant de bonnes notions en mathématiques/statistiques, une base solide en cryptographie et quelques connaissances en programmation puisse la comprendre et l'implémenter aisément. En effet, la plupart des articles détaillant l'attaque sur internet, sont soit très anciens, soit dédiées à des lecteurs très avancées en cryptographie. Nous commencerons donc par expliquer l'attaque d'un point de vue théorique; nous prendrons par la suite le temps de présenter pas à pas la manière dont une telle attaque peut être implementée en exploitant plusieurs failles de différentes manières. Finalement, une étude statistique reposant sur l'implémentation fournie avec ce docuement cloturera l'étude et démontrera les possiblités et les limites de cette attaque sur des machines référencées.

1.2 Notations :

Nous utiliserons tout au long de ce document, les notations suivantes :

p, q : Deux nombre premiers
N : Module RSA
ϕ : Indicatrice d'Euler
e : Clée publique
d : Clée privée
m : Message clair
c : Message chiffré
s : Message signé
h : Fonction de hachage
Ψ : Sample généré de messages
ω : Set de messages respectant un critère
κ : Partie de la clé déjà découverte

τ : Temps d'exécution
Δ : Différence de temps
$\tilde{\alpha}$: α dans la forme de Montgomery
$\star$: Multiplication de Montgomery

2 Cryptosystème RSA :

2.1 Histoire

RSA est un cryptosystème à clé publique nommée d'après Ron Rivest, Adi Shamir et Len Adleman, qui l'ont inventé en 1977. Comme dans tout algorithme cryptographique asymétrique, Alice et Bob possèdent chacun une paire de clés privée et publique pour communiquer en utilisant le protocole RSA. La clé publique comprend un très grand nombre N qui est le produit de deux nombres premiers p et q. C'est sur ce produit que repose la force du RSA. En effet, il est très facile de générer p et q mais très difficile de factoriser N. On dit donc que le RSA est basé sur le problème NP-Complet de factorisation de très grands nombres premiers.

2.2 Fonctionnement

RSA est utilisé pour chiffrer, déchiffrer et signer des messages. Ces opérations sont faites à l'aide de l'algorithme d'exponentiation rapide (Square and Multiply) que l'on décrira plus loin. L'exposant public e chiffrera les messages, tandis que l'exposant privé d les déchiffrera et signera les messages dans une certaine mesure. Pour calculer les nombres RSA, on génère dans un premier temps deux nombres pseudo-aléatoires p et q, de tailles approximativement égales de telle sorte que leur produit N soit de la longueur de bits souhaités, soit par exemple 2048 bits. L'exposant e est premier avec $\phi(n)$ et est choisi de manière arbitraire, mais très souvent dans une liste restreinte qui favorise l'efficacité en terme de calculs. Finalement pour trouver l'exposant d, il suffira d'inverser e modulo $\phi(n)$. Notons que toutes les opérations du RSA se font modulo N.

De manière formelle :

- $p, q \in P$ et $|p| \approx |q| \approx |N| \div 2$

- $N = p \times q$
- $\phi(n) = (p-1) \times (q-1)$
- $e \in \{3, 5, 17, 257, 65537\}$ avec $\text{pgcd}(e, \phi(n)) = 1$ et $e < \phi(n)$
- $d = e^{-1} \bmod N$

Opérations avec le RSA :

- $C = M^e \bmod N$
- $M = C^d \bmod N$
- $S = (hash(m))^d \bmod N$

2.3 Algorithmes de génération des facteurs RSA :

2.3.1 Generateur pseudo-aléatoire de nombres premiers :

Cet algorithme utilise un Triple-DES [1] pour générer des aléas et vérifie la primalité du nombre à l'aide du test de Miller-Rabin.

2.3.2 Generation de p et q

Les deux nombre premiers p et q sont générés à l'aide de l'algorithme précédent.

2.3.3 genration de la clé publique e

l'exposant e est choisi arbitrairement à partir de la liste décrite plus haut. Ceci est possible, notamment car l'exposant e est publique mais également. Les nombres constituants la liste possèdent uniquement deux bits à 1, ce qui nous amène à seulement deux multiplications lors de l'exécution du square and Multiply.

[1] publié dans l'article de Louis Goubin et Jacques Patarin : Génération d'aléas.

Algorithm 1 Random number generator

procedure RANDOMGENERATION(RSASize)
  $x \leftarrow AleaBitGen()$ ▷ Real alea geneartion : take 1 or 0 each time
  $k1 \leftarrow AleaBitGen()$
  $k2 \leftarrow AleaBitGen()$
  $k3 \leftarrow AleaBitGen()$
  $p \leftarrow TripleDES_k(x1, x2, x3)$

  if ($RSASize = 16$ or $RSASize = 32$) **then**
    $p \leftarrow p[0..RSASize]$ ▷ if the Rsa size $<$ 64bits we should resize th p
  else
    while $i \neq numberOfBlock$ **do**
      ▷ number of blocks = RSA size / 64
      $p \leftarrow Concatenate(p, TripleDES_{k1,k2,k3}(x))$
      $i \leftarrow i + 1$
      $x \leftarrow AleaBitGen()$

  $p \leftarrow Mix(p)$

  if $LowWeightBit(p) = 0$ **then**
    $LowWeightBit(p) \leftarrow 1$ ▷ This ensures the number is odd
  if $HightWeightBit(p) = 0$ **then**
    $HightWeightBit(p) \leftarrow 1$ ▷ This ensures n's last bit is 1

  while $MillerRabinTest(p) \neq true$ **do**
    $p \leftarrow p + 2$

  return p

Algorithm 2 p and q generation

procedure PANDQGENERATION(e)
  do
    $p \leftarrow RandomGeneration()$
  while $(p \bmod e) = 1$
  do
    $q \leftarrow RandomGeneration()$
  while $AbsoluteValue(p - q) \leq e$ **and** $(q \bmod e) = 1$ ▷ p, q must be equal size, but not too close in absolute value.
  $n \leftarrow p * q$
  return p, q, n

Algorithm 3 e generation

procedure EGENERATION
  $e \leftarrow RandValue(3, 5, 17, 257, 65537)$ ▷ take randomly one of the values
  return e

2.3.4 Calcul de l'indicatrice d'euler phi(n):

L'indicatrice d'Euler est essentielle au calcul de l'exposant d. On ne connaît pas encore d'algorithmes de complexité raisonnable pour calculer l'indicatrice d'Euler d'un entier n donné. En effet, il faut impérativement calculer les facteurs premiers de n en complexité sous-exponentielle. Le problème de factorisation est donc NP-Complet.

Algorithm 4 $\varphi(n)$ computation

1: **procedure** PHICOMUTATION
2: $\varphi(n) \leftarrow (p-1) \times (q-1)$
3: **return** $\varphi(n)$

2.3.5 La clé privée d:

On calcule l'exposant privé d en inversant e mod $\phi(n)$.

Algorithm 5 d computation

1: **procedure** DCOMPUTATION
2: $d \leftarrow e^{-1} \bmod \phi(n)$
3: **return** d

2.3.6 Generation des paramètres de RSA:

L'algorithme suivant est celui utilisé pour générer tous les paramètres.

Algorithm 6 keys computation

1: **procedure** *RSA* KEYS GENERATION(size)
2: **if** $Size > 2^6$ **and** $Size < 2^{12}$ **and** ($Size$ **and** $(Size-1)$) **then**
3: **do**
4: $e \leftarrow EGeneration()$
5: $(n, p, q) \leftarrow PAndQGeneration(e)$
6: $\varphi(n) \leftarrow PhiComutation()$
7: **while** $pgcd(\phi(n), e) \neq 1$
8: $d \leftarrow DComputation()$
9: **return** n, e, d

2.3.7 Chiffrement :

Pour chiffrer un message on l'élève puissance e le tout modulo N.

Algorithm 7 Encryption

```
1: procedure RSAEncrypt(m, e, N)
2:     c ← m^e mod N
3:     return c
```

2.3.8 Dechiffrement :

Pour déchiffrer un message on l'élève puissance d le tout modulo N.

Algorithm 8 Decryption

```
1: procedure RSADecrypt(c, d, N)
2:     m ← c^d mod N
3:     return m
```

2.3.9 Signature :

Pour signer un message, on le hash à l'aide de la fonction de hachage SHA-3[2] et on élève le résultat à la puissance d. La vérification de la signature se fait en comparant $h(m)$ et s^e.

Algorithm 9 Decryption

```
1: procedure RSADSign(m, d, N)
2:     s ← h(m)^d mod N
3:     return m
```

2.4 Arithmétique de Montgomery

Une multiplication modulaire classique est assez lente à calculer avec des algorithmes classiques. Une division est par construction une opération très couteuse et est toujours nécessaire dans une multiplication modulaire pour calculer le nombre de soustractions nécessaire du module N au produit AB. Étant donnée qu'en

[2] cryptographiquement sûr

cryptographie à clé publique on travaille souvent avec de très grands nombres, cette division est encore plus lente.

En 1985, le mathématicien américain Peter L.Montgomery introduit un système de calcul portant son nom, permettant entre autres de calculer une multiplication modulaire plus rapidement. Cette multiplication est très largement répandue dans les implémentations typiques du RSA.

Cette multiplication étant effectuée dans le système Montgomery, il nous faut préalablement convertir les opérandes de la multiplication dans ce système. Les calculs seront ensuite faits dans la notation de Montgomery, il faudra par conséquent effectuer une conversion inverse pour avoir le résultat final de la multiplication.

2.4.1 Fonctionnement

Soit N un entier impair, k la taille binaire de N et un quelconque entier T tel que $0 \leq T < N$.

Selon le théorème de Montgomery :

$$T.2^{-k} \equiv \frac{T+UN}{2^k} \pmod{N}$$

$$\text{Avec : } U = TN' \bmod 2^k \text{ et } N' = -N^{-1} \bmod 2^k$$

T vérifie les propriétés suivantes :

$$\begin{cases} T + UN \propto 2^k \\ 0 \leq \frac{T+UN}{2^k} < 2N \end{cases}$$

On en déduit que $R := T2^{-k} \bmod N$ peut être calculé en deux étapes :

1. $R \leftarrow \frac{T+UN}{2^k}$
2. $R \leftarrow R - N$

Les opérations arithmétique dans le système de Montgomery repose sur les calculs précédent et représente un entier α quelconque comme suit :

$$\tilde{\alpha} = \alpha 2^k \bmod N \text{ avec } 0 \leq \alpha < N \quad .$$

On peut donc formaliser la multiplication Montgomery comme tel :

$$\tilde{\alpha} \star \tilde{\beta} := (\tilde{\alpha} \cdot \tilde{\beta}) 2^{-k} \bmod N$$

$$\text{En posant : } \sigma = \alpha \cdot \beta \bmod N$$

$$\tilde{\sigma} \equiv (\alpha \cdot \beta) 2^k \equiv (\tilde{\alpha} \cdot \tilde{\beta}) 2^{-k} \equiv \tilde{\alpha} \star \tilde{\beta} \pmod{N}$$

2.4.2 Algorithme de multiplication de Montgomery

Algorithm 10 Multiplication de montgomery

1: **procedure** MONTGOMERYMUL(a, b, R)
2: $T \leftarrow \tilde{\alpha}.\tilde{\beta}$ ▷ $\tilde{\alpha}$ désigne la forme de montgomery de a
3: $T' \leftarrow T \cdot -n^{-1} \bmod R$
4: $m \leftarrow T'.N + T$
5: $\tilde{\sigma} \leftarrow m/R$ ▷ $0 \leq \phi(c) < 2n$
6: **if** $\tilde{\sigma} > N$ **then**
7: $\tilde{\sigma} \leftarrow N - \tilde{\sigma}$ ▷ This is the extra Réduction of Montgomery
8: **return** $\tilde{\sigma}$

2.5 Square and Multiply :

2.5.1 Principe :

Les trois opérations possibles avec RSA à savoir le chiffrement le déchiffrement et la signature, nécessitent de calculer de très grandes puissances. L'algorithme naïf pour effectuer de tels calculs se montre très rapidement inefficace est extrêmement lent. C'est pourquoi une implémentation classique du RSA, comprend l'algorithme de square and Multiply sous ces différentes formes (Right To Left, left To Right, square and multipy Always etc...) pour procéder à ces élévations.

Ceci étant dit, une amélioration de l'algorithme consiste à utiliser la multiplication de Montgomery pour rendre le calcul encore plus rapide. De plus, le square étant une multiplication du nombre par lui-même on peut considérer que le square est

une multiplication. Les deux opérations du square and Multiply sont donc faites dans le système de Montgomery.

2.5.2 Algorithme de square and multiply left to right avec la multiplication de Montgomery

Algorithm 11 Square and Multiply

```
1: procedure SquareAndMultiply(m, e, N )
2:     x← m'                          ▷ m' est l'écriture de m sous forme de montgomery
3:     for (i = 1, i++, i < taille(d)) do
4:         x ← (x, x, R)
5:         if d[i] = 1 then
6:             x ← montgomeryMul(m', x, R)
7:     x ← x.R^{-1}                   ▷ Transformation inverse de montgomery
8:     return x
```

3 Timing Attack :

Une timing attaque est une cryptanalyse par canal auxiliaire d'algorithmes cryptographique reposant sur l'existence d'une variation de temps dans les exécutions d'un ou de plusieurs algorithmes qui composent le cryptosystème en utilisant un certain nombre d'entrées distinctes. Dans le cas du RSA, c'est l'algorithme d'exponentiation rapide qui comprend cette variation. Le but d'une timing attaque sur l'algorithme RSA est de récupérer la clé secrète d.

3.1 Histoire

En 1996, Paul Kocher présente une idée de timing attaque contre l'exponentiation modulaire. Seulement cette attaque est très théorique et nécessite que l'attaquant connaisse précisément les détails de l'implémentation des algorithmes à attaquer. Plus tard des attaques plus pratiques et plus développées ont permis de casser des exponentiations modulaires et donc des RSA, de tailles de modules plus ou moins grandes.

3.2 Principe

les Timing attaques s'applique sur des algorithmes ayant une implémentation dénotant un temps d'exécution non constant liées à des opérations conditionnelles. Cette différence conditionnelle nous donne des informations sur les conditions d'exécutions. En analysant à l'aide d'étude statistiques différents comportements liés à des interactions avec le système en question on obtient des informations sur le cryptosystème tel que la clé secrète de chiffrement.

3.3 Timing attack sur RSA

Rappelons que notre implémentation du RSA comprend un square and Multiply left to right utilisant la multiplication de Montgomery. Cette multiplication comme décrite plus haut, peut selon la taille du résultat de la multiplication nécessiter une soustraction supplémentaire du module N au résultat; on appelle cette opération extra réduction de Montgomery. C'est cette réduction conditionnelle qui divise naturellement les temps d'exécution d'une multiplication modulaire en deux temps différents :

$$t_1 \text{ et } t_2 \text{ avec } t_1 = t_2 + \epsilon \ (\epsilon \text{ étant le temps d'une extra-réduction}) \quad .$$

Nous utiliserons cette faille pour attaquer le RSA et nous supposerons dans ce qui suit que nous sommes en possession d'une Smart Card, implémentant un RSA Montgomery et qu'on peut envoyer autant de messages qu'on veut à la carte. À chaque réception de message la carte signera[3] le message et nous le retournera. Nous serons également capables de calculer le temps nécessaire à chaque signature par la carte.

L'attaque se fera bit par bit, on attaquera à chaque étape le bit i a l'aide de la partie de la clé $(d_0 d_1 d_2 ... d_{i-1})_2$ déjà trouvée au tour précédent.(avec $(d_0 ... d_{l-1})_2$ l'écriture binaire de d). Ceci est effectivement possible car on connaît déjà le premier bit de la clé soit d_0 qui vaut toujours 1.

[3] La signature n'implique pas de hachage dans notre cas $S = C^d$

3.4 Attaque classique

3.4.1 Principe :

L'attaque classique consiste à attaquer l'étape de la multiplication du square and Multiply. Nous générerons un sample de message Ψ. Pour chaque bit, nous déterminerons si l'étape de multiplication qu'il implique dans le square and Multiply nécessite une réduction supplémentaire ou pas. C'est ce critère de réduction qui nous permettra de diviser Ψ en deux sets et de déterminer ω_1 et ω_2. Les messages appartenant à ω_1 seront ceux qui pour le bit actuel i qu'on cherche à déterminer feront une extra-réduction; les autres messages seront naturellement mis dans ω_2. Nous profiterons de cette étape pour calculer les deux temps d'exécution de la multiplication modulaire liée à ce bit t_1 et t_2 (ex : $x^2 \star x \bmod N$ pour le bit 1).

Notons que cette séparation en deux sets est possible car nous sommes en possession d'un square and Multiply qu'on peut faire tourner pour déterminer les messages engendrant une extra-réduction. Nous calculerons ensuite le temps nécessaire à une signature de chacun des messages de ω_1 et de ω_2 et calculerons pour chaque set la moyenne de temps d'exécution Mt_i. Nous calculerons par la suite :

$$\Delta = |Mt_1 - Mt_2|$$

Si $k_i = 1$:

La multiplication a réelement lieu pour le bit i notre séparation en deux sets prendra donc sens et Δ vaudra le temps d'une extra-réduction $t_r = t_1 - t_2$.

Si $k_i = 0$:

La multiplication n'a pas lieu et notre critère de séparation n'a pas de sens, Δ devra donc être très proche de 0.

Ainsi nous avons trouvé le bit i nous suivrons pour les bits suivants les mêmes étapes jusqu'à trouver le clé finale. Nous vérifions la réussite de l'attaque en signant un message avec la clé trouvée.

3.4.2 Notions importantes : le bruit

Évidemment la carte et plus particulièrement le RSA n'exécute pas seulement le square and Multiply. Ces autres opérations brouillent très fortement les résultats; c'est ce qu'on appelle le bruit. Le théorème Central limite nous indique que plus un échantillon est grand et plus il tend vers une loi normale. Concrètement ceci veut dire qu'il faut envoyer plusieurs messages (de l'ordre de milliers) pour réussir à atténuer ce bruit et garantir la véracité de Δ et donc la découverte du bit i.

3.4.3 Algorithme :

Algorithm 12 Timing Attack

$M \leftarrow generateSet(sizeOfSet)$

while $m_i \in M$ **do**
    if isReduction(m_i) **then**
        $M1 \leftarrow M1 \cup \{m_i\}$
    else
        $M2 \leftarrow M2 \cup \{m_i\}$

$T_r \leftarrow 0$
while $m_i \in M1$ **do**
    $T_r \leftarrow T_r + ExecutionTime(RSASign(m_i))$
$T_r \leftarrow T_r / M_1.size$

$T \leftarrow 0$
while $m_i \in M1$ **do**
    $T \leftarrow T + ExecutionTime(RSASign(m_i))$
$T \leftarrow T/M2.size$

if $(|T_r - T| = t_{reduction} + Bruit)$ **then**
    $d_i = 1$
else$(|T_r - T| = Bruit)$
    $d_i = 0$

    return d

3.5 Attaque améliorée :

La première attaque présente deux principaux défauts :

1. La détermination d'ϵ est difficile.

2. Le nombre de réductions de multiplication de l'étape Multiply du square and Multiply Montgomery est fallacieux et beaucoup trop changeant. En effet il s'est avéré après plusieurs tests, que le nombre de réductions diverge trop quand l'un des opérandes de la multiplication est fixe. Cette conclusion est purement statistique et n'a pas été prouvée mathématiquement.

C'est pourquoi nous avons pensé à une autre attaque qui nous permettrait d'avoir un critère de séparation plus fin et qui permettrait de simplifier la distinction finale du bit.

3.5.1 Principe :

Dans cette attaque nous nous intéresserons à l'étape du square de l'algorithme square and Multiply et non pas à la partie de la multiplication. En effet rappelons que les deux opérations du square and Multiply se font en Montgomery et que le square est une multiplication du nombre par lui-même.

Nous cherchons à trouver le bit i et nous avons déjà la partie $\kappa = (d_0 d_1 d_2 ... d_{i-1})_2$ de la clé. Nous exécuterons pour chacun des messages de Ψ le square and multiply des bits connues en calculant :

$$M^{\kappa} \ (M \text{ étant un message de } \Psi)$$

Les prochaines opérations seront le square et le multiply liée au bit i. Nous calculerons l'étape du square :

$$(M^{\kappa})^2 \text{ que nous appellerons } \aleph \quad .$$

Si le bit i est à 1 la prochaine étape est la multiplication de $\aleph$ par M et si le bit est à 0 alors il n'y aura pas de multiplication. La prochaine étape concerne le bit $i+1$ et quelque soit la valeur du bit dans le square and multiply le square est forcément calculé. Nous partagerons donc Ψ en 4 sets différents comme suit :

1. le bit i est à 1
   - $M \in \omega_1$ si $(\aleph \times M)^2$ nécessite une réduction.
   - $M \in \omega_2$ si $(\aleph \times M)^2$ ne nécessite pas de réduction.

2. le bit i est à 0
    - $M \in \omega_3$ si $(\aleph)^2$ nécessite une réduction.
    - $M \in \omega_4$ si $(\aleph)^2$ ne nécessite pas de réduction.

Nous calculerons ensuite les temps d'exécutions moyens pour tout le RSA et ce, pour chacun des 4 sets. Seulement, cette fois ci en utilisant le médiane qui donne selon plusieurs test un meilleur résultat que la moyenne. Nous calculerons ensuite la différence des médianes des deux premiers sets comme suit :

$$diff_1 = Med_1 - Med_2$$

idem pour les deux autre sets :

$$diff_2 = Med_3 - Med_4$$

Finalement :

$$\text{Si } diff_1 > diff_2 \text{ alors } i = 1$$

$$\text{Sinon si } diff_2 > diff_1 \text{ alors } i = 0$$

$$\text{Sinon si } diff_2 \simeq diff_1 \text{ alors l'attaque est erronnée pour ce bit}$$

3.5.2 Avantage de cette attaque par rapport à l'autre :

1. Cette attaque ne dépend plus d'ϵ et la différence entre $diff_1$ et $diff_2$ est très visible car seulement l'une des deux séparations à un sens.
2. Nous pouvons considérer qu'une détection d'erreur est possible par construction dans le cas ou $diff_2 \simeq diff_1$
3. Cette attaque necessite jusqu'à 75% moins de messages que la première attaque
4. Attaquer le square implique que la multiplication de Montgomery n'est pas falacieuse car les opérandes ne sont pas constantes.

Nous avons donc trouver le bit i en utlisant κ et le square lié au bit $i+1$. Les autres bits seront trouvés de la même manière hormis le dernier, qu'on trouvera en essayant les deux cas possibles (0 ou 1).

4 Statistiques

4.1 Attaque classique :

- Nombre de messages par bit : 15000
- Temps d'execution par bit : 6 secondes.
- Taux d'erreur par bit : 1 bit / 64 environs.

4.2 Attaque améliorée :

- Nombre de messages par bit : 3500
- Temps d'execution par bit : 2 secondes.
- Taux d'erreur par bit : 1% d'erreurs dans le pire cas.

4.2.1 Conclusion statistique :

L'attaque améliorée est nettement plus efficace car elle réussit dans 99% des cas, nécessite 75% de messages en moins et surtout, elle est 3 fois plus rapide. Notons tout de même que nous prenons en considération le bruit dû aux différentes autres étapes ainsi que le bruit de transport entre la machine et la carte à puce.

5 Implémentation :

Une implémentation fonctionnelle de la deuxième attaque en C++ est prposée afin de mettre en évidene la cohérence des résultats théoriques obtenues. Nous avons simulé une carte bancaire dans un premier programme, placée sur une machine.

Une deuxième machine enverra à l'aide du protocle SFTP le programme d'attaque qui l'exercera à distance et remplira au fur et à mesure un fichier contenant la clée privée d. À la fin de l'exécution la clé est vérifiée deux fois en signant un message, car le dernier bit ne peut pas être découvert à l'aide de cette attaque.

6 Contre mesures :

1. Une première contre mesure possible consiste à faire en sorte que la réduction supplémentaire dans l'algorithme de Montgomery est toujours effectué même si le résultat n'est pas utilisé. Cette modification est très facile à mettre en place et n'affecte pas réellement les performances de la carte.
2. Borné le temps d'éxecution et obtenir T_{max} d'une signature. La contre mesure consiste à faire en sorte cette fois ci que les temps de signature de tous les messages quelque soient leurs tailles se fasse en T_{max}. Un désavantage génant liée a cette contre mesure est évidement que les performances sont cette fois ci largement moins bons.
3. Faire en sorte d'avoir des signatures aveugles pour empêcher l'attaquant d'effectuer l'attaque. Avant de signer le message, ce dernier est manipulé avec une valeur aléatoire σ qui le modifie. Le fait que ce message nécessite ou non une réduction ne donne aucune information sur la réduction du message d'origine.

7 Conclusion :

La timing attaque sur RSA, prouve qu'on peut facilement venir à bout de cryptosystèmes extrêmement perfectionnés et ce à l'aide de détails qui parraissent de prime abord superflus. L'élement le plus délicat dans cette attaque a été la manipulation des temps d'éxecutions liés à une unique réduction.Cette contrainte nous oblige à produire une implémentation rigoureuse et chrirurgicale. Aujourd'hui cette attaque n'est plus possible car les contre-mesures ont évidement été mises en places.

8 Remerciements :

Nous tenons particulièrement à remercier Mr Louis Goubin, directeur du Master sécurité des Contenus, des Réseaux, des Télécommunications et des Systèmes (SeCReTS) qui nous a encadré dans ce projet, et appris énormément de choses tout au long de l'année universitaire et de ce travail d'étude et de recherche.

9 Références :

1. Cours de monsieur Louis Goubin M1 Informatique - Calcul sécurisé.
2. Algorithme de Montgomery pour la multiplication modulaire.
3. Une approche de la timing attaque.
4. Lois binomiale.
5. Thérème central limite.