Attaque à texte chiffré choisi contre les protocoles basés sur PKCS#1

Thomas Baignères

2002, Winter Semester

Table des matières

1 Introduction

Ce rapport a été réalisé dans le cadre d'un projet de semestre au laboratoire de cryptographie et de sécurité (LASEC) de l'EPFL. Il étudie l'attaque à texte chiffré choisi réalisée en 1998 par Daniel Bleichenbacher contre le standard PKCS#1.
Dans un premier temps je rappelerai le fonctionnement de RSA, puis du standard PKCS#1. Je décrirai ensuite l'attaque, les problèmes auxquels j'ai été confrontés et les solutions que j'ai essayées d'apporter. Pour terminer j'exposerai la méthode que j'ai employée pour effectuer une réalisation de l'attaque.

2 RSA : algorithme de chiffrement à clé publique

2.1 Historique minimaliste

RSA est un algorithme de chiffrement à clé publique. Il a été baptisé d'après le nom de ses inventeurs : Ron Rivest, Adi Shamir et Leonard Adleman. Sa sécurité est assurée par le fait qu'il est difficile de décomposer de très grands nombres en facteurs premiers.

2.2 La génération des clés, le chiffrement et le déchiffrement

On choisi deux nombres premiers p et q. On calcule leur produit, que l'on appellera modulus de RSA :

$$n = p \cdot q$$

On choisi ensuite une clé de chiffrement aléatoire e de telle manière que e et $\varphi(n) = (p-1)(q-1)$ [1] soient premiers entre eux.
On utilise ensuite l'algorithme d'Euclide pour calculer la clé de déchiffrement d telle que :

$$d \equiv e^{-1} \pmod{(p-1)(q-1)}$$

On peut constater que d et n sont aussi premiers entre eux. Le nombre e sera la clé publique de RSA, et d sera la clé privée. Les nombres p et q ne seront plus utilisés mais ne doivent en aucun cas être révélés.
Soit m le message à chiffrer. Ce message m devra être plus petit que n. Le chiffrement se fait de la manière suivante :

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

Le déchiffrement s'effectue à l'aide de la clé privée :

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

On adoptera dans la suite de ce texte les notations suivantes :
- Pour le chifrement : $\mathcal{C}(m) = c$.
- Pour le déchiffrement : $\mathcal{D}(c) = m$.

[1] $\varphi(n)$ est appelé l'indicateur d'Euler de n

2.3 Quelques précisions

Les deux nombres premiers p et q sont de très grande taille, de l'ordre de 100 à 200 chiffres (et plus encore). C'est à dire que l'on choisira pour p et q des nombres de 512 bits. En les multipliant on obtient ainsi un nombre de 1024 bits pour n. Pour les générer on utilise des algorithmes probabilistes. On commence par générer un nombre de façon aléatoire et on vérifie à l'aide d'un test probabiliste (comme celui de Miller-Rabin) s'il est premier. Le resultat d'un tel test donne généralement la borne supérieure de la probabilité que ce nombre ne soit pas premier.
En pratique, le choix de e n'est pas laissé au hasard. PKCS#1 recommande les nombres 3 et $2^{16}+1 = 65537$. Les représentations binaires de ces deux derniers nombres ne présentent que deux 1, le calcul de leurs puissances en est grandement facilité.
L'indicateur d'Euler de n, $\varphi(n)$, est le nombre d'entiers inférieurs à n, premiers avec n.

$$\varphi(n) = \#\{i, i < n | \gcd(i, n) = 1\}$$

Finalement, pour générer la clé privée d de l'algorithme, on utilise l'algorithme d'Euclide étendu.

2.4 Le déchiffrement

Si l'on effectue les opérations suivantes mod$n$, on obtient :

$$c^d \equiv (m^e)^d \equiv m^{ed} \pmod{n}$$

Comme $d \equiv e^{-1} \pmod{(p-1)(q-1)}$ alors $ed \equiv 1 \pmod{(p-1)(q-1)}$ ce qui revient à dire qu'il existe une constante k telle que $ed = 1 + k \cdot ((p-1)(q-1))$.
Donc :

$$c^d \equiv m^{k(p-1)(q-1)+1} \equiv m \cdot m^{k(p-1)(q-1)} \equiv m \cdot (m^{(p-1)(q-1)})^k \equiv m \cdot (m^{\varphi(n)})^k \pmod{n}$$

Or, d'après le petit théorème de Fermat, comme $pgcd(m, n) = 1$ on a :
$m^{\varphi(n)} \equiv 1 \pmod{n}$. On obtient finalement :

$$c^d \equiv m \cdot (1)^k \equiv m \pmod{n}$$

2.5 En bref

Modulus :

$n = p \cdot q$ avec p et q deux nombres premiers qui doivent rester secrets

Clé publique :

$$e \quad \text{premier avec} \quad \varphi(n) = (p-1)(q-1)$$

Clé privée :

$$d = e^{-1} \bmod ((p-1)(q-1))$$

Chiffrement :

$$c = m^e \pmod{n}$$

Déchiffrement :

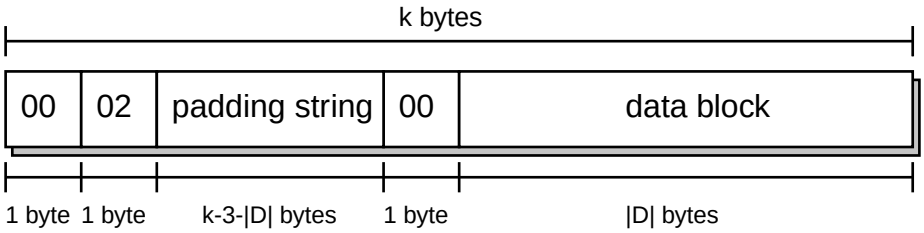
$$m = c^d \pmod{n}$$

3 Description de PKCS#1

Soit n le modulus, e la clé publique de RSA, soit d la clé privée correspondante. On notera k la longueur en bytes de n. On a donc :

$$2^{8(k-1)} \leq n < 2^{8k}$$

Le bloc utilisé pour les chiffrements conformes à PKCS#1 a la forme suivante [2] :



La longueur du bloc est la même que celle de la clé n.
D sera le bloc de données que l'on veux chiffrer. Le Padding String (PS) est formé de $k - 3 - |D|$ bytes non nuls, engendrés de façon pseudo-aléatoire. La longueur de PS doit être d'au moins 8 bytes, autrement dit, $|D|$ ne doit pas dépasser $k - 11$ bytes. Une fois le bloc constitué, on le chiffre à l'aide de la clé publique de RSA.
Le serveur PKCS#1 recoit le cryptogramme et le déchiffre avec sa clé privée. Il vérifie ensuite que le texte clair obtenu est bien conforme à PKCS#1.

Un bloc EB de k bytes :

$$EB = EB_1||\ldots||EB_k$$

est conforme à PKCS#1 s'il est de la forme du bloc précédemment décrit. En particulier, EB doit satisfaire les conditions suivantes :

- $EB_1 =$ `0x00`
- $EB_2 =$ `0x02`
- EB_3 jusqu'à EB_{10} sont différents de `0x00`
- Au moins un des bytes EB_{11} à EB_k est `0x00`. Il permet de repérer le début du message.

Par extension, un texte chiffré sera dit conforme si le texte clair correspondant est conforme à PKCS#1.

4 L'attaque à texte chiffré choisi contre PKCS#1 : Premier contact

4.1 Généralités sur l'attaque

Dans une attaque à texte chiffré choisi, l'ennemi choisi un texte chiffré, l'envoie à la victime et recoit en retour le texte chiffré correspondant ou une partie

[2] Il existe deux autres forme de blocs réservés pour les signatures digitales

de celui-ci. On dit que ce type d'attaque est adaptative si l'ennemi choisi un texte chiffré en fonction des précédents résultats de ses attaques.
On appelera Oracle le serveur PKCS#1 qui, lorsqu'il reçoit un message chiffré non conforme à PKCS#1, renvoie un message d'erreur. Dans notre cas, on suppose que l'ennemi a accès à un Oracle qui, pour chaque texte chiffré reçu, renvoie un message d'erreur chaque fois qu'un message reçu n'est pas conforme.

4.2 L'idée fondamentale de l'attaque

L'ennemi choisi un texte chiffré c. Son objectif est de trouver $m \equiv c^d \pmod{n}$.
L'ennemi choisi des entiers s, et calcule :

$$c' \equiv cs^e \pmod{n}$$

L'ennemi envoie c' à l'Oracle. Si l'Oracle ne renvoie pas de message d'erreur, l'ennemi sait que les deux premiers bytes de ms sont `0x00` et `0x02`. En effet :

$$(c')^d \equiv (cs^e)^d \equiv c^d s^{ed} \equiv ms \pmod{n}$$

On peut chercher à quel intervalle appartient le message clair ms. On note $B = 2^{8(k-2)}$.
La plus petite valeur possible sera :

$$00||02||(k-2) \text{ bytes } \texttt{0x00} = 2 \cdot 2^{8(k-2)} = 2B$$

La plus grande valeur possible sera :

$$00||02||(k-2) \text{ bytes } \texttt{0xFF} = 3 \cdot 2^{8(k-2)} - 1 = 3B - 1$$

On a donc :

$$2B \leq ms \pmod{n} < 3B$$

Remarque : On n'a pas considéré ici qu'un message ne comportant que des 0 à partir du 3ème byte de poids fort n'est de toute façon pas conforme à PKCS#1. Grâce à ce genre d'information, il faut de l'ordre de 2^{20} messages chiffrés pour conclure l'attaque.

4.3 Le plan de l'attaque.

L'attaque peut être décomposée en trois phases :
- L'ennemi se donne un texte chiffré c_0 qui correspond à un message inconnu m_0. L'objectif ici est de trouver m_0.
- L'ennemi cherche de petites valeurs s_i pour lesquelles le texte chiffré $c_0(s_i)^e \pmod{n}$ est conforme. Pour chaque bonne valeur, l'ennemi calcule, en utilisant ce qu'il sait déjà sur m_0, un ensemble d'intervalles qui peuvent contenir m_0.
- Cette dernière phase n'intervient que quand il ne reste plus qu'un seul intervalle possible. L'ennemi a alors assez d'information pour choisir s_i tel que $c_0(s_i)^e \pmod{n}$ ait plus de chance d'être conforme qu'un texte chiffré choisi au hasard. On augmente graduellement la taille de s_i en se rapprochant de m_0 jusqu'à ce qu'il ne reste plus qu'une valeur possible.

5 L'attaque en détails

5.1 L'algorithme décomposé

Dans cette partie, M_i sera toujours un ensemble d'intervalles fermés qui sont calculés après qu'une bonne valeur s_i soit trouvée. m_0 sera toujours dans l'un des intervalles de M_i.

algorithme de l'attaque :

- 1er Pas :

  On choisi un entier c. On génère aléatoirement plusieurs s_0 jusqu'à ce qu'il y en ait un qui soit accepté par l'Oracle, c'est à dire que $c(s_0)^e \pmod n$ soit conforme. Dès qu'on le trouve :

$$c_0 \leftarrow c(s_0)^e \pmod n$$

$$M_0 \leftarrow \{[2B; 3B-1]\}$$

$$i \leftarrow 1$$

- 2ème Pas :

  – Si $i = 1$, on cherche le plus petit entier positif s_1 tel que le texte chiffré $c_0(s_1)^e \pmod n$ soit conforme. On cherche s_1 tel que :

$$s_1 \geq \left\lceil \frac{n}{3B} \right\rceil$$

  – Si $i \neq 1$:
  Si M_{i-1} contient au moins deux intervalles on cherche le plus petit $s_i > s_{i-1}$ tel que $c_0(s_i)^e \pmod n$ soit conforme.
  Si $M_{i-1} = \{[a, b]\}$ on choisis deux petits entiers r_i et s_i tels que :

$$r_i \geq 2 \cdot \frac{bs_{i-1} - 2B}{n}$$

$$\left\lceil \frac{2B + r_i n}{b} \right\rceil \leq s_i < \left\lceil \frac{3B + r_i n}{a} \right\rceil$$

  Jusqu'à ce que $c_0(s_i)^e \pmod n$ soit conforme.

- 3ème Pas :
  On a trouvé s_i. On calcule l'ensemble M_i tel que :

$$M_i = \bigcup_{(a,b,r)} \left\{ \left[\max\left(a, \left\lceil \frac{2B + rn}{s_i} \right\rceil \right), \min\left(b, \left\lfloor \frac{3B - 1 + rn}{s_i} \right\rfloor \right) \right] \right\}$$

  Pour tout $[a, b] \in M_i$ et $\left\lceil \frac{as_i - 3B + 1}{n} \right\rceil \leq r \leq \left\lfloor \frac{bs_i - 2B}{n} \right\rfloor$.

- 4ème Pas :

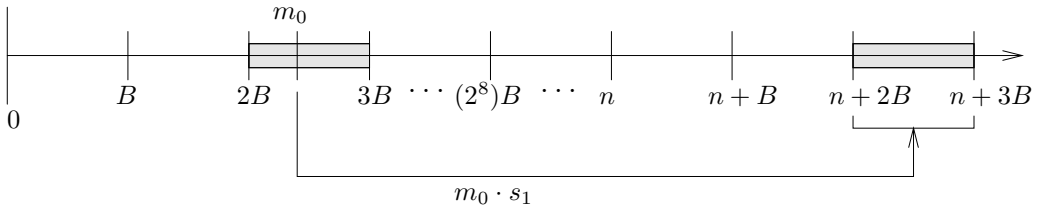
  – Si M_i contient un seul intervalle, de longueur 1 (i.e., $M_i = [a, a]$), alors :

  $$m \leftarrow a(s_0)^{-1} \pmod{n}$$

  Retourner m comme solution de $m \equiv c^d \pmod{n}$.
  – Sinon, effectuer $i \leftarrow i + 1$ et retourner au 2ème Pas.

5.2 Explications concernant le pas 2

– On considère le cas où $i = 1$.
On a $m_0 \equiv ms_0 \pmod{n}$. Dans ce cas m_0 est tel que $\mathcal{D}(m_0) = c_0$. On cherche s_1 tel que $s_1 \geq \lceil \frac{n}{3B} \rceil$. En effet :



Le fait que $c_0(s_1)^e \pmod{n}$ soit conforme à PKCS#1 est équivalent à $m_0 \cdot s_1 \pmod{n}$ conforme à PKCS#1. Dans ce cas, le nombre $m_0 \cdot s_1$ doit être compris entre $n + 2B$ et $n + 3B$. En effet, s_1 devant être le plus petit possible, le nombre $m_0 \cdot s_1$ ne peut être au delà de cet intervalle.
On a donc :

$$n + 2B \leq m_0 \cdot s_1$$

De plus on a vu que :

$$m_0 < 3B$$

On déduit de ces deux inégalités :

$$n + 2B < 3B \cdot s_1 \Rightarrow s_1 > \frac{n}{3B} + \frac{2}{3}$$

Comme s_1 doit être entier :

$$s_1 \geq \left\lceil \frac{n}{3B} + \frac{2}{3} \right\rceil \geq \left\lceil \frac{n}{3B} \right\rceil$$

Finalement :

$$s_1 \geq \left\lceil \frac{n}{3B} \right\rceil$$

On cherche s_1 le plus petit possible mais on voit qu'il ne sert à rien de partir de 0. Il faut commencer à $\lceil \frac{n}{3B} \rceil$.
– On considère le cas où $i \neq 1$ et où $M_{i-1} = \{[a, b]\}$.
On considère l'intervalle $[a, b]$ de M_{i-1}. On sait que m_0 lui appartient, par définition. On sait que $m_0 \cdot s_i$ est conforme, donc il existe un r_i tel que :

$$2B + r_i n \leq m_0 \cdot s_i < 3B + r_i n$$

C'est à dire :

$$\begin{cases} m_0 s_i \geq 2B + r_i n \\ m_0 s_i < 3B + r_i n \end{cases}$$

D'un autre côté, dans ce cas précis on a :

$$a \leq m_0 \leq b$$

On en déduit que :

$$\begin{cases} bs_i \geq 2B + r_i n \\ as_i < 3B + r_i n \end{cases} \Rightarrow \begin{cases} s_i \geq \frac{2B + r_i n}{b} \\ s_i < \frac{3B + r_i n}{a} \end{cases}$$

Donc :

$$\frac{2B + r_i n}{b} \leq s_i < \frac{3B + r_i n}{a}$$

Comme s_i est un nombre entier, cette dernière inégalité implique :

$$\left\lceil \frac{2B + r_i n}{b} \right\rceil \leq s_i < \left\lceil \frac{3B + r_i n}{a} \right\rceil$$

5.3 Explications concernant le pas 3

– On considère un des intervalles $[a, b]$ de M_{i-1}. On suppose que m_0 lui appartient, ce qui est une possibilité puisque cet intervalle appartient à M_{i-1}. On sait que pour obtenir un $m_0 \cdot s_i$ conforme, il doit exister un r tel que :

$$2B + rn \leq m_0 \cdot s_i < 3B + rn \Rightarrow \begin{cases} rn \leq m_0 s_i - 2B \\ rn > m_0 s_i - 3B \end{cases}$$

Comme on a suposé que $m_0 \in [a, b]$, ceci implique :

$$\begin{cases} rn \leq bs_i - 2B \\ rn > as_i - 3B \end{cases}$$

Et donc :

$$as_i - 3B + 1 \leq rn \leq bs_i - 2B$$

Donc :

$$\frac{as_i - 3B + 1}{n} \leq r \leq \frac{bs_i - 2B}{n}$$

Comme r est entier, ceci implique :

$$\left\lceil \frac{as_i - 3B + 1}{n} \right\rceil \leq r \leq \left\lfloor \frac{bs_i - 2B}{n} \right\rfloor$$

– En ce qui concerne la construction de M_i :
Comme $m_0 \cdot s_i$ est valide :

$$2B + rn \leq m_0 \cdot s_i < 3B + rn \Rightarrow \frac{2B + rn}{s_i} \leq m_0 \leq \frac{3B + rn - 1}{s_i}$$

Comme m_0 est entier :

$$\left\lceil \frac{2B+rn}{s_i} \right\rceil \leq m_0 \leq \left\lfloor \frac{3B+rn-1}{s_i} \right\rfloor$$

Comme l'objectif est de réduire l'intervalle de départ $[a, b]$, on s'en assure en choisissant de remplacer ce dernier par :

$$\left[\max\left(a, \left\lceil \frac{2B+rn}{s_i} \right\rceil \right), \min\left(b, \left\lfloor \frac{3B-1+rn}{s_i} \right\rfloor \right) \right]$$

6 Les problèmes rencontrés, éléments de solution

6.1 Concernant le troisième pas

- Le problème :

Dans le rapport original de Daniel Bleichenbacher[1], la condition sur r est la suivante :

$$\frac{as_i - 3B + 1}{n} \leq r \leq \frac{bs_i - 2B}{n}$$

Ce qui suppose que :

$$\frac{as_i - 3B + 1}{n} \leq \frac{bs_i - 2B}{n} \tag{1}$$

alors que dans le présent rapport cette condition devient :

$$\left\lceil \frac{as_i - 3B + 1}{n} \right\rceil \leq r \leq \left\lfloor \frac{bs_i - 2B}{n} \right\rfloor$$

Ce qui suppose que :

$$\left\lceil \frac{as_i - 3B + 1}{n} \right\rceil \leq \left\lfloor \frac{bs_i - 2B}{n} \right\rfloor \tag{2}$$

Le nombre r étant entier, les parties entières sont implicites.
Le problème est le suivant :
L'équation (1) est toujours vérifiée. En effet :

$$\begin{cases} B \geq 1 \\ a < b \end{cases} \Rightarrow as_i - 3B + 1 \leq bs_i - 2B \Rightarrow \frac{as_i - 3B + 1}{n} \leq \frac{bs_i - 2B}{n}$$

En revanche l'équation (2) ne l'est pas toujours. Prenons par exemple le cas de figure suivant : $\frac{as_i-3B+1}{n} = 1.4$ et $\frac{bs_i-2B}{n} = 1.6$. L'équation (1) est vérifiée en revanche l'équation (2) ne l'est pas. Il sera donc impossible de trouver un r entier vérifiant l'équation (1).

- Une solution éventuelle :

Supposons qu'à un certain moment, le pas 2 trouve un s_i qui posera le problème dans le pas 3. Deux solutions sont envisageables si le cas se présente.

- La première est de faire repartir le programme au pas 1. Un nouveau s_0 est généré. On peut espèrer qu'avec cette nouvelle valeur, le problème ne se reposera plus.
- La deuxième est de chercher une nouvelle valeur de s_i, plus grande que la précédente. Tout le problème est de ne pas passer trop de temps à chercher une nouvelle valeur $s_i^{'}$ qui ne posera pas problème à son tour. Le calcul de la difference minimale entre s_i et $s_i^{'}$ est exposé au paragraphe suivant.

6.2 Solution sans réinitialisation

Supposons qu'un système de vérification situé au niveau du 2ème pas permette de remarquer qu'une valeur s_i trouvée à ce même pas posera problème au pas suivant, c'est à dire que pour l'un des intervalles $[a, b]$ de M_i, on a :

$$\left\lceil \frac{as_i - 3B + 1}{n} \right\rceil > \left\lfloor \frac{bs_i - 2B}{n} \right\rfloor$$

Cette inégalité ne risque pas de s'inverser tant que le terme de droite n'augmente pas, puisque le terme de gauche croît quand s_i augmente. La nouvelle valeur $s_i^{'}$ de s_i devra donc vérifier l'inégalité suivante :

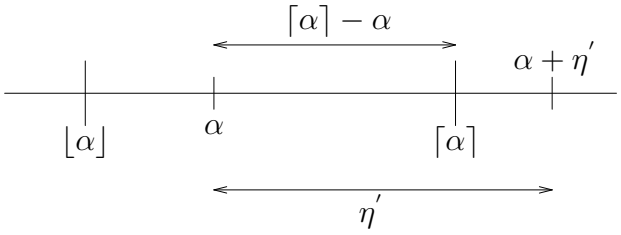
$$\left\lfloor \frac{bs_i^{'} - 2B}{n} \right\rfloor > \left\lfloor \frac{bs_i - 2B}{n} \right\rfloor$$

Notons $\eta = s_i^{'} - s_i$. L'inégalité précédente donne :

$$\left\lfloor \frac{b(s_i + \eta) - 2B}{n} \right\rfloor > \left\lfloor \frac{bs_i - 2B}{n} \right\rfloor \Rightarrow \left\lfloor \frac{bs_i - 2B}{n} + \frac{b}{n}\eta \right\rfloor > \left\lfloor \frac{bs_i - 2B}{n} \right\rfloor$$

Si on note $\alpha = \frac{bs_i - 2B}{n}$ et $\eta^{'} = \frac{b}{n}$, la dernière inégalité donne :

$$\left\lfloor \alpha + \eta^{'} \right\rfloor > \lfloor \alpha \rfloor$$



Cette dernière inégalité est équivalente à la suivante :

$$\eta^{'} > \lceil \alpha \rceil - \alpha$$

Si l'on remplace α et $\eta^{'}$ par leur valeur cela donne :

$$\frac{b}{n}\eta > \left\lceil \frac{bs_i - 2B}{n} \right\rceil - \frac{bs_i - 2B}{n} \Rightarrow \eta > \frac{n}{b}\left(\left\lceil \frac{bs_i - 2B}{n} \right\rceil - \frac{bs_i - 2B}{n} \right)$$

Et comme η est entier :

$$\eta \geq \left\lceil \frac{n}{b} \left(\left\lceil \frac{bs_i - 2B}{n} \right\rceil - \frac{bs_i - 2B}{n} \right) \right\rceil$$

En conclusion, on voit que si une valeur s_i trouvée au deuxième pas de l'algorithme pose problème, on cherche une nouvelle valeur $s_i^{'}$ telle que :

$$s_i^{'} \geq s_i + \left\lceil \frac{n}{b} \left(\left\lceil \frac{bs_i - 2B}{n} \right\rceil - \frac{bs_i - 2B}{n} \right) \right\rceil$$

6.3 Toutes les bonnes choses ont une fin

Malheureusement, après avoir implémenté la solution proposée ci-dessus, il faut se rendre à l'évidence. Le programme boucle sur l'erreur $r_{min} > r_{max}$ assez longtemps pour que la solution radicale soit préférable. J'entends par solution radicale celle de recommencer tout l'algorithme d'attaque à zero, en espérant que la nouvelle valeur de s_0 trouvée aléatoirement permette cette fois de mener l'algorithme à son terme.
Il est néanmoins possible de donner un semblant d'explication. Si le cas de figure suivant se présente :

$$\left\lceil \frac{as_i - 3B + 1}{n} \right\rceil > \left\lfloor \frac{bs_i - 2B}{n} \right\rfloor \tag{3}$$

Sachant que l'on a toujours :

$$\frac{as_i - 3B + 1}{n} < \frac{bs_i - 2B}{n}$$

Cela signifie que :

$$\frac{as_i - 3B + 1}{n} \approx \frac{bs_i - 2B}{n}$$

Dans le meilleur des cas, la solution proposée au paragraphe précédent permettra à l'algorithme de continuer. Les deux entiers a et b ne pouront que se rapprocher. La quantité $\frac{as_i - 3B + 1}{n}$ va croître, la quantité $\frac{bs_i - 2B}{n}$ va décroître. Comme l'inégalité

$$\frac{as_i - 3B + 1}{n} < \frac{bs_i - 2B}{n}$$

est toujours vérifiée, ces deux quantités seront de plus en plus proches l'une de l'autre.
Cela n'implique pas qu'il sera impossible de mettre en défaut l'inégalité (3), il suffit que les deux fractions soient de part et d'autre d'un entier, mais cela signifie qu'il sera de plus en plus difficile de trouver un entier s à la fois acceptable au pas 2 et tel que :

$$\left\lceil \frac{as_i - 3B + 1}{n} \right\rceil \leq \left\lfloor \frac{bs_i - 2B}{n} \right\rfloor$$

En bref si le problème survient à un moment donné dans l'algorithme, la meilleure chose à faire semble être de le réinitialiser.

7 Description de l'implémentation

7.1 Les deux modes de fonctionnement

Le programme qui réalise concrètement l'attaque a été écrit en C. En dehors des librairies habituelles, j'ai utilisé la librairie gmp[3] de GNU, pour les calculs sur les grands nombres. Le programme est une application client-serveur, j'utilise des sockets pour les faires communiquer.
J'ai écrit deux versions du client, il y a donc deux modes de fonctionnement :

- Dans le premier, le client réalise l'attaque telle qu'elle est décrite dans l'algorithme. Le client envoie des messages $c_0 \cdot s^e$, le serveur déchiffre le message et renvoie un message d'erreur si le message obtenu n'est pas conforme.
- Dans le deuxième, on passe un paramètre supplémentaire au client : la clé privée d. L'objectif est de simuler la vérification de la conformité au sein même du client. Plutot que de former $c_0 \cdot s^e$ et de l'envoyer au serveur, on forme $m_0 \cdot s$ et on vérifie sa conformité. m_0 est déterminé grace à la clé publique d. Cette opération permet de simuler une vraie attaque, mais elle est beaucoup plus rapide. En effet dans la première version il faut effectuer un calcul de puissance pour chaque message envoyé, $c_0 \cdot s^e$, alors que ce n'est pas nécessaire dans le deuxième mode de fonctionnement.

7.2 Les commandes et leur fonctionnement

Les commandes sont situées dans trois sous-répertoires du répertoire `./projet`.
L'application serveur se trouve dans le répertoire `./projet/SERVEUR`. La commande à executer est `./serveur_pkcs`. Aucun argument n'est nécessaire.
Le client fonctionnant en mode normal est situé dans le répertoire `./projet/CLIENT`.
La commande à executer est `./client_pkcs <server>`, où `<server>` est l'adresse IP ou le nom de la machine sur laquelle tourne le serveur.
Le client fonctionnant en mode local est situé dans le répertoire `./projet/LOCAL_CLIENT`.
La commande à executer est `./client_pkcs <server> <clé d>`, où `<server>` est l'adresse IP ou le nom de la machine sur laquelle tourne le serveur et `<clé d>` est la clé privée d à passer en paramètre.
La valeur du message chiffré que l'on veut déchiffrer est modifiable dans le fichier `client_pkcs.h`.

7.3 Les autres répertoires

Dans le répertoire principal du projet, on trouvera aussi les répertoires suivants :

- `./DOC` : Ce répertoire inclut la documentation utilisée pour exploiter la librairie `gmp` de GNU, le rapport de Daniel Bleichenbacher[1], ainsi que d'autres documents qui m'ont été utiles pour ce projet.
- `./PKCS` : Contient les fichiers relatifs à l'implémentation du standard PKCS#1.
- `./RAPPORT` : Contient le code source du présent rapport, ainsi qu'un Makefile pour le compiler. Il contient aussi trois sous répertoires dans lesquels se

trouvent le résumé pour la page web, le poster au format `.ppt` compressé au format `.zip` et les transparents utilisés pour la présentation.
- `./RSA` : Contient les fichiers relatifs à l'implémentation de RSA.
- `./VERIFICATION` : Contient un programme de test qui permer de vérifier la validité des résultats obtenus grace à l'attaque. Pour cela il faut éditer les paramètres du fichier `verif.c`.

7.4 Principe de dévelopement

J'ai divisé le programme en cinq sous-programmes :
- `rsa.c` : implémente les fonctions nécessaires à la génération d'un couple de clés, au chiffrement et déchiffrement de messages.
- `pkcs.c` : utilise `rsa.c`. Il permet de verifier si un message chiffré (ou clair) est conforme à PKCS#1. Pour programmer cette partie j'ai suivi à la lettre les indications données dans la documentation sur PKCS#1[7].
- `serveur_pkcs.c` : utilise `pkcs.c`. Lorsqu'il est executé, il génère un couple de clé en prévision d'une future connection. Contrairement à ce qu'il se passe dans la réalité, une paire de clé est générée par connection. Ceci pour s'assurer que l'algorithme ne fonctionne pas uniquement avec une paire de clé en particulier. `pkcs.c` attend ensuite une connection. Il envoie ensuite au client le modulus n et la clé publique e. Il déchiffre alors tous les messages qu'il reçoit pendant cette session et renvoie un message d'erreur pour chaque message non conforme reçu.
- `client_pkcs.c` (normal) : Effectue une demande de clé publique. Ensuite il execute l'algorithme tel qu'il a été décrit. L'algorithme est réinitialisé lorsque $r_{min} > r_{max}$.
- `client_pkcs.c` (local) : Effectue une demande de clé publique. Déchiffre c_0 pour connaitre m_0. Ensuite il execute l'algorithme tel qu'il a été décrit, à la différence près qu'il n'envoie aucun message au serveur. Ici on vérifie que $m_0 \cdot s$ est conforme plutot que $c_0 \cdot s^e$. L'algorithme est réinitialisé lorsque $r_{min} > r_{max}$.

8 Conclusion

Telle que je l'ai comprise, l'attaque proposée par Daniel Bleichenbacher[1] fonctionne, moyennant quelques petites modifications. Le serveur est ici vulnérable car il laisse échapper de l'information lorsqu'il prévient le client qu'un message est non conforme à PKCS#1. D'autres attaques ont elles aussi profité du même type de vulnérabilité de la part du serveur. James Manger a réalisé une attaque contre PKCS#1 v2.0 en exploitant une faille de sécurité similaire[5].

Le principe qu'il faudra sans doute retenir de l'attaque est que le serveur ne doit en aucun cas laisser filtrer de l'information, et ce pour éviter une *side channel attack*.

Remerciements
J'aimerais remercier le Professeur Serge Vaudenay de m'avoir proposé ce projet et Pascal Junod pour son aide inestimable.

Références

[1] Daniel Bleichenbacher. Choosen Ciphertext Attacks Against Protocols Based on the *RSA* Encryption Standard PKCS#1, 1998.

[2] Jean-Yves Le Boudec. Réseaux Informatiques I. ICA-EPFL, 1999.

[3] GNU. gmp. http ://www.swox.com/gmp/.

[4] Brian W. Kernighan and Denis M. Ritchie. Le langage C - Norme ANSI. Dunod, second edition, 1990.

[5] James Manger. A Chosen Ciphertext Attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as Standardized in PKCS#1 v2.0, 2001.

[6] Neil Matthew and Richard Stones. Programmation Linux. Eyrolles, 2000.

[7] RSA Laboratories. PKCS#1 : RSA Encryption standard, novembre 1993.

[8] Bruce Schneier. Cryptographie appliquée. Wiley, second edition, 1997.