INF582 : Cryptologie
Attaque de clés RSA par la méthode de Wiener

Nicolas DOUZIECH - Thomas JANNAUD - X2005

19 mars 2008

Table des matières

1 Quelques rappels sur le cryptosystème RSA

Le principe de RSA repose sur le chiffrement asymétrique à clé publique / clé privée suivant : Pour e et d bien choisis, $\forall x \in \mathbb{Z}/n\mathbb{Z}, (x^e)^d = x \mod n$. On rend alors publique la clé e. Si l'on a un message $m \in \mathbb{Z}/n\mathbb{Z}$, on nous envoie m^e et connaissant d nous retrouvons le message initial.

La puissance du procédé tient dans le fait que connaissant m et e, il est très dur de retrouver d sans connaître la décomposition en facteurs premiers de n.

La méthode consiste donc à choisir 2 nombres premiers p et q aussi grands que nécessaire, et à poser $n = p \times q$. On choisit alors e plus ou moins "aléatoirement", on le rend publique, et il reste à calculer d.

Choix de e et d : à x fixé, $\{x^k, k \in \mathbb{Z}\}$ est un sous-groupe multiplicatif de $(\mathbb{Z}/n\mathbb{Z})^\star$, isomorphe d'après le théorème chinois à $(\mathbb{Z}/p\mathbb{Z})^\star \times (\mathbb{Z}/q\mathbb{Z})^\star$. Le cardinal du sous-groupe divise donc $ppcm(p-1, q-1)$. Ainsi, si e est premier avec $p-1$ et $q-1$,et si d est l'inverse de e dans $\mathbb{Z}/(p-1)(q-1)\mathbb{Z}$ (i.e tel que $ed = 1 \mod (p-1)(q-1)$), alors $(x^e)^d = x \mod n$.

2 Principe de l'attaque de Wiener

L'attaque de Wiener s'applique tout particulièrement lorsque l'exposant secret d est faible. Elle repose sur l'équation définissant d d'après l'exposant public e et la fonction de Carmichael appliquée au module $n = p \times q$:

$$e \times d = 1 \mod \lambda(n)$$

De cette équation, on en déduit l'existence d'un entier K tel que $e \times d = 1 + K \times \lambda(n)$. De plus, $\lambda(n) = ppcm(p-1, q-1) = \frac{(p-1)(q-1)}{G}$, avec $G = pgcd(p-1, q-1)$. Il vient alors $ed = 1 + \frac{K}{G}(p-1)(q-1) = 1 + \frac{k}{g}(p-1)(q-1)$, après réduction tel que $pgcd(k, g) = 1$.

On en déduit alors le pivot de l'algorithme des fractions continues appliquées à cette attaque :

$$\frac{e}{pq} = \frac{k}{dg}(1 - \frac{p+q-1-\frac{g}{k}}{pq}) \iff f' = f(1-\delta)$$

qui sera efficace si $\delta = \frac{p+q-1-\frac{g}{k}}{pq}$ est très faible.

Or, comme $pgcd(k, g) = 1$ par définition et $pgcd(k, d) = 1$ (d'après la formulation de Bézout : $ed = 1 + k\frac{(p-1)(q-1)}{g}$), si on détermine la fraction $\frac{k}{dg}$, on obtiendra aisément k et le produit dg.

3 L'algorithme des fractions continues

Le développement en fractions continues permet de calculer le numérateur et le dénominateur d'un nombre rationnel x (de manière exacte). Ainsi, sachant que $f' = f(1-\delta)$ et que l'on connait f', on va pouvoir remonter non pas à f, mais à

son numérateur et à son dénominateur, ce qui apporte considérablement plus d'informations. Cela permet en l'occurence de "casser" le système puisque connaissant ces informations on verra que l'on parvient à remonter à p, q, et d.

3.1 Principe

Tout réel x peut s'écrire de manière unique sous la forme

$$x = q_0 + \cfrac{1}{q_1 + \cfrac{1}{q_2 + \cfrac{1}{q_3 + \cfrac{1}{q_4 + \dots}}}}, \quad q_1, q_2, \dots \in \mathbb{N}$$

Par construction, $q_i = \lfloor \frac{1}{r_i - 1} \rfloor$ et $r_i = \frac{1}{r_i - 1} - q_i$.

On peut noter $(q_0, q_1, q_2, \dots)$ le développement d'un réel x. On peut intuiter qu'une certaine notion de "continuité" se dégage : Si x est proche de y, les premiers termes du développement de x seront égaux à ceux de y.

3.2 Quelques unes des propriétés fondamentales des fractions continues

D'une part, le développement converge rapidement, et il est fini si et seulement si x est rationnel.

D'autre part, pour i pair :

$$(q_0, q_1, \dots, q_i) < (q_0, q_1, \dots, q_{i+1}) < (q_0, q_1, \dots, q_i + 1)$$

et pour i impair :

$$(q_0, q_1, \dots, q_i + 1) < (q_0, q_1, \dots, q_{i+1}) < (q_0, q_1, \dots, q_i)$$

La suite $x - (q_0, q_1, \dots, q_i)$ est donc alternée, et les sous-suites d'indices pairs seulement ou impairs seulement sont monotones.

Enfin, en définissant n_i et d_i tels que $\forall i, \frac{ni}{di} = (q_0, \dots, q_i))$, on a $(q_0, q_1, \dots, q_i) = \frac{q_i n_{i-1} + n_{i-2}}{q_i d_{i-1} + d_{i-2}}$ et $n_i d_{i-1} - ni - 1di = -(-1)^i$.

3.3 L'idée de Wiener

Rappelons que $f' = f(1 - \delta)$, où δ semble être de petite taille, et l'on aimerait connaître le numérateur et le dénominateur de f, sachant ceux de f'. L'idée de Wiener consiste à effectuer le développement en fractions continues de f' : On note $(q_0, q_1, \dots, q_m)$ le développement de f et $(q'_0, q'_1, ; q'_{m'})$ celui de f'.

Si f et f' sont suffisamment proches, alors les premiers termes du développement de f et de celui de f' sont égaux. Et avec un peu de chance, $m' \geq m$, ce qui ferait que le développement de f serait un sous-développement de celui de f'. L'algorithme va fonctionner si et seulement si les m premières valeurs du développement de f' sont celles de f, c'est à dire si :

– m est pair et $(q_0, q_1, \dots, q_m - 1) < f' \leq (q_0, q_1, \dots, q_m) = f$

– m est impair et $(q_0, q_1, \dot{;} q_m + 1) < f' \leq (q_0, q_1, \dot{;} q_m) = f$
(ceci entrainerait que le développement de f' continue plus loin qu'au rang m)

Cette condition est remplie si $\delta < \frac{1}{\frac{3}{2} n_m d_m}$ où n_m et d_m sont le numérateur et le dénominateur de f.

Ce qui semble être de la "chance" n'en est en fait pas. La condition $m\prime \geq m$ qui semble être aléatoire est en fait vérifiée sur l'intervalle des rationnels compris entre $(q_0, q_1, \ldots, q_m - 1)$ et $(q_0, q_1, \ldots, q_m) = f$ (si m est pair), qui est en fait de largeur $f\delta$.

3.4 Démonstration

On prouve le résultat dans le cas où m est un entier pair ≥ 2. On a l'inéquation ci-dessus si et seulement si $\delta < 1 - \frac{(q_0, q_1, \ldots, q_m - 1)}{(q_0, q_1, \ldots, q_m)}$.

En utilisant les notations n_i et d_i vues dans la partie "propriétés fondamentales", l'inéquation se réécrit : $\delta < \frac{n_{m-1} d_{m-2} - n_{m-2} dm - 1}{(q_m n_{m-1} + n_{m-2})(q_m d_{m-1} + d_{m-2} - d_{m-1})}$ Et après réduction, en utisant la propriété fondamentale citée en dernier, on obtient $\delta < \frac{1}{n_m (d_m - d_{m-1})}$. Par voie de conséquence, $\delta < \frac{1}{n_m d_m}$ est suffisant pour obtenir la convergence de l'algorithme. (et donc $\delta < \frac{1}{\frac{3}{2} n_m d_m}$ aussi)

3.5 Estimation de la borne sur d

On utilise les notations de l'article. Il y est écrit que pour $d < n^{\frac{1}{4}}$ le "cassage" du code réussit grâce à l'algorithme des fractions continues.
Condition de réussite : $\delta < \frac{1}{n_m d_m} = \frac{1}{kdg}$.
Or $\delta = \frac{p+q-1-\frac{g}{k}}{pq}$ donc il suffit que $\frac{p+q}{pq} < \frac{1}{kdg}$ pour que l'algorithme marche. On peut estimer p et q par $\sqrt{n}$ puisque $n = pq$.

Une condition de réussite est donc $\frac{1}{\sqrt{n}} \leq \frac{1}{kdg}$ soit $d \leq \frac{\sqrt{n}}{kg}$.
Or $k = \frac{K}{PGCD(K,G)}$ et $g = \frac{G}{PGCD(K,G)}$ donc il suffit que $d \leq \frac{\sqrt{n}}{KG}$.

Comme $G = PGCD(p-1, q-1)$ et $K = \frac{ed-1}{PPCM(p-1,q-1)} \sim \frac{ed}{PPCM(p-1,q-1)}$, cela équivaut $d \leq \frac{\sqrt{n}}{\frac{G^2 ed}{(p-1)(q-1)}}$ avec $(p-1)(q-1) \sim n$.

Une condition est donc $d \leq \frac{\sqrt{n}}{\frac{G^2 ed}{n}}$ soit $d^2 \leq \frac{n\sqrt{n}}{G^2 e}$. Comme $e < n$, une condition plus stricte est $d^2 \leq \frac{\sqrt{n}}{G^2}$.

p et q étant premiers, avec une forte probabilité les facteurs de $(p-1)$ et $(q-1)$ n'ont que de petits facteurs premiers et $G(= PGCD(p-1, q-1))$ est très petit devant n (soit $O(1)$). On retrouve donc la condition $d < n^{\frac{1}{4}}$.

4 Pratique

4.1 Algorithme complet

Voici l'algorithme complet de l'attaque de Wiener contre une clé publique (n, e).

CasserRSA(n, e)

$m \leftarrow 0$, $q_0 \leftarrow \lfloor e/n \rfloor$, $r_0 \leftarrow e/n - q_0$, $n_0 \leftarrow q_0$ et $d_0 \leftarrow 1$
while $(r_m \neq 0)$
  do if m est pair
    then $k/dg \leftarrow < q_0, \ldots, q_m + 1 >$
    else $k/dg \leftarrow < q_0, \ldots, q_m >$
  ▷ On vérifie si le développement partiel convient pour k/dg
  $edg \leftarrow e \times dg$
  $(p-1)(q-1) \leftarrow \lfloor edg/k \rfloor$
  $g \leftarrow edg \mod n$
  if $(g = 0)$
    then goto FinTest ▷ le développement ne convient pas
  $p + q \leftarrow n - (p-1)(q-1)$
  if $p + q$ impair
    then goto FinTest ▷ le développement ne convient pas
  $(\frac{p-q}{2})^2 = (\frac{p+q}{2})^2 - n$
  $\frac{p-q}{2} = \sqrt{((\frac{p-q}{2})^2)}$ ▷ racine carré entière
  if $(\frac{p-q}{2})^2$ n'est pas un carré parfait
    then goto FinTest ▷ le développement ne convient pas
  $p \leftarrow \frac{p+q}{2} + \frac{p-q}{2}$, $q \leftarrow \frac{p+q}{2} - \frac{p-q}{2}$
  $d \leftarrow dg/g$
  ▷ Fin des vérifications, retour de d
  return d
FinTest :
  ▷ Calcul de l'élement suivant du développement de e/n
  $q_{m+1} = \lfloor \frac{1}{r_m} \rfloor$ et $r_{m+1} = \frac{1}{r_m} - q_{m+1}$

  if (m=0)
    then $n_1 = q_0 q_1 + 1$ et $d - 1 = q_1$
    else $n_{m+1} = q_{m+1} n_m + n_{m-1}$ et $d_{m+1} = q_{m+1} d_m + d_{m-1}$
return "Clé incassable"

4.2 Implémentation

Pour implémenter cet algorithme, nous avons opté pour développer une classe dont les membres seront nos variables n, e, m_{etap}, la liste des q_i, celle des n_i et d_i et r_i est stocké sous le format numérateur (variable r_i) et dénominateur (variable r_{i-1}).

Pour ce qui est des fonctions de calcul, nous avons :

- Une fonction *casser* qui prend en argument le mode de calcul choisi. Nous verrons qu'il s'agit en fait des améliorations que nous proposerons plus bas.
- Une fonction *breaker* qui est l'implémentation de l'algorithme de base mais dans lequel nous entrons le numérateur et le dénominateur de f' qui approche, de manière inférieure $f = \frac{k}{dg}$.
- Une fonction *calcul_next* qui calcule l'élément $m + 1$ du développement en fraction continue et met à jour les listes q_i, n_i et d_i.

– Une fonction *Verifier* qui, connaissant le développement (partiel) de f' vérifie s'il convient pour f. Cette fonction prend en argument une structure (qui contient n, p, q, e, d) qui sera remplie si la vérification s'avère positive et retournera VRAI, sinon retournera FAUX.

Parallèlement, nous avons développé une classe principale qui permet de choisir différents formats d'entrées (lancement en ligne de commande). On peut spécifier la valeur de n, soit directement, soit via un fichier, celle de e de la même manière. On peut aussi demander la génération aléatoire d'une clé dont on spécifie la taille (la taille de p et de q). Un mode permet aussi de lancer k tests générant aléatoirement des clés et les testant. Enfin, on peut spécifier le mode de calcul qui précise si on travaille avec la version de base (par défaut) ou avec une amélioration.

Pour générer une clé, nous choisissons "aléatoirement" p et q premiers de la taille spécifiée. n est alors le produit des deux et $\varphi(n) = (p-1) \times (q-1)$. On choisit alors d aléatoirement de taille le quart de celle de n (`BigInteger(n.bitLength()/4, rand)`) et on essaie d'inverser d pour obtenir e ($\mod \varphi(n)$). Cependant, on sait que d n'est pas toujours inversible, dans ce cas, on rejette simplement la clé et on en essaie une autre.

Notons en passant que pour calculer la racine carrée de n, nous avons développé une fonction *sqrt* sur les *BigInteger* qui calcule le plus grand entier (format *BigInteger*) m tel que $m^2 \leq n$ par la méthode de Newton.

4.3 Améliorations implémentées

Nous avons implémenté deux améliorations proposées dans l'article. Les deux reposent sur le principe d'approcher mieux f par f' (toujours inférieurement). Cependant, elles peuvent ne pas fonctionner dans des cas où l'attaque classique fonctionne (car on peut se retrouver avec $f' > f$)

Pour la première (mode 1), il s'agit de remplacer $f = \frac{e}{n}$ par $\frac{e}{(\sqrt{n}-1)^2}$, ce qui améliore la borne maximale de d.

Pour la deuxième (mode 2), on souhaite toujours s'approcher inférieurement de f. Aussi, en supposant e et n grands, on peut remplacer $f = \frac{e}{n}$ par $f = \frac{e+i}{n+i}$ avec i faible. Aussi, on essaie avec i variant de 1 à 100.

4.4 Quelques résultats

A titre d'illustration, nous avons testé le programme.

Concernant l'algorithme de base, nous avons testé la création de 10000 clés de taille 512 bits. On constate que seulement 3106 clés ont pu être générées jusqu'au bout (l'inversion de d échoue donc à 70%) Parmi ces 3106 clés, seulement 186 n'ont pas pu être cassées par l'attaque de Wiener, ce qui est un taux d'échec de seulement 6%. Sur un ordinateur normal, le test de ces 10000 clés n'ont pris que sept minutes. Notons cependant qu'un même test de 10000 clés mais en choisissant d par `BigInteger(n.bitLength()/4-1, rand)` ne génère statistiquement aucune clé incassable !

Concernant les améliorations, nous avons simulé 100000 clés comme précédemment mais lorsqu'une clé n'est pas cassable par la méthode de base, alors nous tentons de la

casser par la version améliorée 1 et par la version améliorée 2. Comme précédemment, seules 30% des clés ont pu être générées jusqu'au bout (30740 sur 100000), parmi celles-ci, 2528 n'ont pu être cassées par l'attaque de base (soit environ 8%). Cependant, parmi ces 2528 clés, l'amélioration 1 a permis d'en casser 35, ce qui est certes faible mais montre l'intérêt de cette dernière. Malheureusement, la deuxième amélioration n'a pu en casser aucune.

A titre d'exemple, nous donnons deux clés non cassées par l'attaque de base mais où la première amélioration a réussi. (l'algorithme prend n et e en entrée et retourne p, q et d)

$n =$	709904940300015996400809032464414641688420991851679798308042375843328957917310881789630059149200956779199993473959300456887100950313145997889135492431229 3
$e =$	480059080341509900306277570379221116391331618825871519987688985231317060628605573847702454638691726642897862500375290391345850232673591996072707565187255 1
$p =$	115088288511471823122445937999426539141123926888949808211472535165876275496561
$q =$	616835083292818075653514360382553007889032110216523677843744384562666874137013
$d =$	337422556629262957727680320823671598551

$n =$	794782504508886730212551662182667440533262191889040011137395319268637265528381778003206900612345981323780883572060384739975071367605608012581874798634944 7
$e =$	652509910762586208578281549850204039537003837222879764692634600121079876063953428168627076972839030752078735606393771287552089560943521184810433447493745 7
$p =$	102075790301626293298652731383506434185015370950620441944026800672878142881511
$q =$	778619986345796712927961683512415746907434378541560478797691994197496803021 77
$d =$	3019666174218337808957239456030181237 93

5 Conclusion

Comme le montre les résultats, l'attaque de Wiener est efficace pour peu que $d < n^{\frac{1}{4}}$. De plus, nous constatons que l'amélioration qui consiste à remplacer $\frac{e}{n}$ par $\frac{e}{(\sqrt{n}-1)^2}$ s'avère utile mais ne peut pas être généralisée car on se retrouve fréquemment au dessus de $\frac{k}{dg}$ et l'algorithme des fractions continues n'est donc plus efficace.