TIPE ENS : Cryptanalyse de RSA par la méthode de Coppersmith

Table des matières

1 Introduction **1**

2 Le cryptosystème RSA **2**
2.1 La cryptographie à clé publique . . . 2
2.2 RSA . . . 2
2.3 Attaquer RSA . . . 2

3 Réduction des réseaux **2**
3.1 Définitions et propriétés . . . 3
3.2 Algorithme LLL . . . 3

4 Attaque de Coppersmith **4**
4.1 Théorème de Howgrave-Graham et conséquence . . . 4
4.2 Méthode de Coppersmith . . . 5

5 Conclusion **6**

A Résultats de l'implémentation **7**

B Pseudo-code de l'algorithme LLL **7**

C Preuves **8**

D Implémentation **10**
D.1 RSA . . . 10
D.2 LLL . . . 13
D.3 Coppersmith . . . 14

E Bibliographie **15**

1 Introduction

La cryptographie, science des codes secrets, joue un rôle très important dans le développement des technologies actuelles et à venir. Des échanges de trames du protocole https, au stockage de données sur des serveurs, de nombreux algorithmes ont été développés pour rendre de l'information indéchiffrable. Un des algorithmes les plus répandus est l'algorithme RSA [1].
La cryptanalyse, quant à elle étudie les méthodes pour décrypter des messages chiffrés sans en avoir les clés. Dans cet exposé nous nous intéressons à une méthode de cryptanalyse de RSA. Cette méthode est proposée par D. Coppersmith [2] et réinterprétée par N. Howgrave-Graham [3]. Elle s'appuie notamment sur des techniques de réduction des réseaux [4]

Après avoir introduit le cryptosystème RSA en section 2, nous étudierons l'objet et la réduction des réseaux en section 3, avant de détailler la méthode de Coppersmith en section 4.

1. [1]
2. [3]
3. [4]
4. [2]

2 Le cryptosystème RSA

Dans cette section nous expliquons le principe du chiffrement par clé publique RSA, nommé d'après ses inventeurs : R.L Rivest, A. Shamir et L. Adleman.

2.1 La cryptographie à clé publique

La cryptographie à clé publique a été d'abord introduite par W. Diffie et M. Hellman. Elle consiste à construire, sur un ensemble de messages M et pour chaque utilisateur i, des applications E_i (clé de chiffrement publique) et D_i (clé de déchiffrement privée) inverses l'une de l'autre, faciles à calculer, et telles que, la connaissance de E_i ne permette pas de déduire D_i. Alors, tous les utilisateurs peuvent utiliser E_i mais seul i connaît D_i.
Ainsi, si un utilisateur j veut envoyer un message $m \in M$ à i, il calcule $c = E_i(m)$ [5], qu'il envoie à i. Alors seul i peut calculer $m = D_i(c)$ et lire le message.

Pour définir les E_i et D_i, on utilise généralement des procédures identiques qui dépendent d'une clé privée et d'une clé publique. C'est le cas dans RSA.

2.2 RSA

Les méthodes de chiffrement et de déchiffrement du cryptosystème RSA reposent sur le théorème suivant.

Théorème 2.1: RSA

Soient p et q des nombres premiers distincts, $N = pq$, $e \in \mathbb{N}$ premier avec $\phi(N) = (p-1)(q-1)$ et $d = e^{-1}[\phi(N)]$. On a

$$\forall m \in \mathbb{N}, (m^e)^d = m[N]$$

Ainsi, le fonctionnement de RSA s'organise autour de 3 étapes :

Premièrement la génération des clés. Elle consiste à générer deux nombres premiers [6] p et q, calculer $N = pq$ et $\phi(N) = (p-1)(q-1)$. Puis générer un nombre e premier à $\phi(N)$ à l'aide de l'algorithme d'Euclide, et son inverse d modulo $\phi(N)$, à l'aide de l'algorithme d'Euclide étendu. La clé publique est (e, N) et la clé privée est d. En suite, l'encodage d'un message $m \in \{0, ..., N-1\}$ se fait à l'aide de la clé publique (e, N) : l'envoyeur calcule $c = m^e[N]$. Enfin, le décryptage d'un message c se fait de manière analogue : le récepteur calcule $m = c^d[N]$.

Notons que les procédures d'encodage et de décodage peuvent se calculer de manière efficace [7] mais que la simple connaissance de (e, N) ne permet pas de déduire facilement d. En effet cela nécessite le calcul $\phi(N) = (p-1)(q-1)$ et donc la factorisation de N, problème pour lequel il n'y a pas d'algorithmes efficaces dans le cas général.

2.3 Attaquer RSA

Dans le cas général, avec les technologies actuelles [8], il semble compliqué d'attaquer RSA de par la difficulté à factoriser de grands nombres. Nous n'avons donc pas d'autre choix que de chercher à exploiter certaines failles d'implémentation.
Le cadre que l'on développe dans cet exposé est le suivant : l'attaquant connaît la clé publique (e, N), ainsi qu'une partie b du message envoyé ($m = b + x$ avec x inconnu). L'objectif de l'attaquant est de trouver x, à partir de $c = m^e = (b+x)^e[N]$, ce qui est équivalent à résoudre $p(x) = (b+x)^e - c = 0[N]$.
On se retrouve à devoir étudier les racines, modulo N, d'un polynôme à coefficients entiers. C'est sur ce problème que porte les travaux de D. Coppersmith puis N. Howgrave-Graham . Leurs méthodes utilisent ce qu'on appelle la réduction des réseaux. Continuons par l'étude de cette théorie.

3 Réduction des réseaux

Dans cette partie nous introduisons la notion de réseau ainsi qu'une méthode algorithmique permettant l'obtention d'un certain type de base de vecteurs courts.

5. En envoyant $c = E_i(D_j(m))$ cette méthode permet aussi une signature du message. i calcule $m = E_j(D_i(c))$.
6. Dans mon implémentation j'utilise le test de primalité probabiliste de Miller Rabin.
7. Par expodentiation rapide.
8. P. Shore trouve en 1994 un algorithme quantique de factorisation en $O(\log(N)^3)$.

3.1 Définitions et propriétés

Définition 3.1 (Réseau, dimension, base)**.** *Soit $n \in \mathbb{N}$, et L un sous-ensemble de $\mathbb{R}^n$. On dit que L est un réseau s'il existe $m \in \mathbb{N}$ et une famille libre $b_1, ..., b_m$ de $\mathbb{R}^n$ telle que*

$$L = \sum_{i=1}^{m} \mathbb{Z} b_i = \{\sum_{i=1}^{m} a_i b_i | a_1, ..., a_m \in \mathbb{Z}\}$$

On dit alors que m est la dimension du réseau et que $b_1, ..., b_m$ en est une base.

On aura souvent tendance à utiliser des réseaux de dimension n. On a aussi la définition suivante.

Définition 3.2 (Déterminant)**.** *Soit L un réseau de $\mathbb{R}^n$ et $b_1, ...b_n$ une base de ce dernier. On appelle déterminant de L la grandeur*

$$\det(L) = |\det(b_1, ..., b_n)|$$

Cette définition est bien licite car elle ne dépend pas de la base choisie. En effet le passage entre deux bases d'un réseau se fait par une matrice de $GL_n(\mathbb{Z})$ qui a donc son déterminant dans $\{-1, 1\}$.
Nous introduisons aussi les notations suivantes pour l'orthogonalisée de Gram-Schmidt, qui sera particulièrement utile dans notre étude.

Définition 3.3 (Orthogonalisation de Gram-Schmidt)**.** *Soit $b_1, ..., b_n$ une base de $\mathbb{R}^n$. On pose $b_1^*, ..., b_n^*$ la base orthogonale définie par $b_1^* = b_1$ et $\forall 2 \leq i \leq n$ $b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^*$ avec $\forall\ 1 \leq j < i \leq n$, $\mu_{i,j} = \frac{<b_i, b_j^*>}{<b_j^*, b_j^*>}$.*

Un problème fondamental dans l'étude des réseaux est la recherche des vecteurs non nuls les plus courts du réseau. En grande dimension ce problème semble difficile [9] à traiter de manière algorithmique. On introduit alors la notion suivante de base réduite, qu'on peut interpréter comme une base de vecteurs relativement courts [10].

Définition 3.4 (Base réduite)**.** *Soit $\mathcal{B} = (b_1, ..., b_n)$ une base d'un réseau L. On dit que $\mathcal{B}$ est réduite si*

$$\forall 1 \leq j < i \leq n, |\mu_{i,j}| \leq \frac{1}{2} \tag{1}$$

$$\forall 2 \leq i \leq n, \|b_i^* + \mu_{i,i-1} b_{i-1}^*\|^2 \geq \frac{3}{4} \|b_{i-1}^*\|^2 \tag{2}$$

On appelera (1) condition de taille, et (2) condition de Lovász.
L'existence de telles bases dans un réseau quelconque n'est pas immédiate. Nous verrons dans la partie suivante un algorithme permettant la construction de base réduite, et donc une preuve constructive de leur existence. Mais intéressons nous d'abord à quelques encadrements qui seront pertinents pour la méthode de Coppersmith.

Propriété 3.1

Soit $b_1, ..., b_n$ une base réduite de L. On a alors

$$\forall 1 \leq j \leq i \leq n, \|b_j\|^2 \leq 2^{i-1} \|b_i^*\|^2 \tag{3}$$

$$\det(L) \leq \prod_{i=1}^{n} \|b_i\| \leq 2^{\frac{n(n-1)}{4}} \det(L) \tag{4}$$

$$\|b_1\| \leq 2^{\frac{(n-1)}{4}} \det(L)^{\frac{1}{n}} \tag{5}$$

Retenons en particulier, pour la suite, la majoration (5) de $\|b_1\|$.
Nous allons désormais décrire l'algorithme LLL qui permet l'obtention de bases réduites.

3.2 Algorithme LLL

L'algorithme LLL, du nom de ses créateurs A. Lenstra, H. Lenstra et L. Lovász, transforme une base d'un réseau en une base réduite.
Il consiste à agir sur la base $b_1, ..., b_n$ donnée en entrée de telle sorte que chaque transformation sur la famille conserve sa propriété de base du réseau. De plus, on voudra la fin d'une étape $k \in \{1, ..., n+1\}$ de l'algorithme, les invariants suivants soient vérifiés

$$\forall 1 \leq l < k, |\mu_{k,l}| \leq \frac{1}{2} \tag{6}$$

$$\forall 1 < i < k, \|b_i^* + \mu_{i,i-1} b_{i-1}^*\|^2 \geq \frac{3}{4} \|b_{i-1}^*\|^2 \tag{7}$$

9. Et même NP-Difficile, ce qui en fait un bon objet pour la cryptographie.
10. La remarque après (1.12) dans [2] permet de vérifier cette intuition.

Notons que si on atteint $k = n + 1$, on retrouve la définition d'une base réduite. On initialise k à 2. On peut ensuite résumer l'algorithme de la manière suivante :

On suppose être à l'étape d'indice k de l'algorithme. On commence par vérifier (6).
Si $|\mu_{k,l}| > \frac{1}{2}$ alors on effectue la transformation $b_k = b_k - [\mu_{k,l}]b_l$ [11] et on actualise la base et les coefficients de Gram-Schmidt associés. Cette transformation garantie (6).
Puis on vérifie si $\|b_k^* + \mu_{k,k-1}b_{k-1}^*\|^2 \geq \frac{3}{4}\|b_{k-1}^*\|^2$: Si cette condition n'est pas vérifiée, on échange b_k et b_{k-1}, on met à jour la base et les coefficients de Gram-Schmidt et on pose $k = \max(1, k-1)$. Dans la nouvelle base, la valeur de $\|b_{k-1}^*\|^2$ est inférieure strictement à $\frac{3}{4}$ son ancienne valeur [12]. Sinon si (7) est vérifiée, on incrémente k.

Le pseudo code détaillé de l'algorithme est décrit en trois parties dans l'annexe B. Une grande partie des fonctions **LLL**, **taille** et **lovasz** correspond au calcul ou à l'actualisation de la base et des coefficients de Gram-Schmidt.

On s'intéresse maintenant à l'étude de la validité de cet algorithme.

Propriété 3.2: Terminaison de LLL

L'algorithme LLL se termine.

En remarquant que k est incrémenté uniquement lorsque les invariants sont vérifiés, et avec la remarque du cas $k = n + 1$, l'algorithme fournit effectivement une base réduite.

Enfin, la force de l'algorithme LLL est sa complexité polynomiale pour trouver une base vecteurs courts, malgré le caractère NP-Difficile de la recherche des vecteurs les plus courts du réseau.

Théorème 3.1: Compléxité LLL

Soit L un réseau, et $b_1, ..., b_n$ une base de ce dernier. On pose $B = \max(2, \|b_1\|, ..., \|b_n\|)$. Alors l'algorithme LLL permet de trouver une base réduite en un nombre d'opération arithmétique qui est polynomial : $O(n^4 \log(B))$.

[13]

Voyons maintenant comment appliquer cette notion de réseau et de base réduite à la recherche de racines d'un polynôme modulo N.

4 Attaque de Coppersmith

Cette partie décrit la méthode, dite de Coppersmith, qui permet de trouver les petites racines modulaires d'un polynôme. Comme expliqué en 1.3 cela nous permettra d'exploiter des failles de RSA. On introduit donc dans cette partie les objets suivants : $P(x) = \sum_{k=0}^{d} a_k x^k \in \mathbb{Z}[x]$ un polynôme unitaire, $X \in \mathbb{N}^*$ et on suppose qu'il existe $x_0 \in \mathbb{Z}$ tel que $x_0 < X$ et $P(x_0) = 0[N]$. L'objectif est de trouver un tel entier x_0.
Plus précisément, on s'intéresse aux résultats de N. Howgrave-Graham qui a repris et simplifié la méthode décrite par Coppersmith.
L'idée directrice est de transformer la recherche des solutions de $P(x) = 0[N]$ en la recherche des racines entières d'un bon polynôme $R(x)$. Ce dernier problème est nettement plus simple à résoudre. En effet, il suffit d'appliquer la méthode de Newton et essayer les entiers les plus proches des solutions approchées trouvées [14].

4.1 Théorème de Howgrave-Graham et conséquence

On veut faire le passage de l'équation modulo N à la recherche de racines entières d'un polynôme.
D'après la majoration de x_0, on sait que $|P(x_0)| \leq \sum_{k=0}^{d} |a_k| X^k$. Ainsi, si les coefficients (a_k) était suffisemment « petits », on aurait $|P(x_0)| < N$ et on pourrait se contenter de résoudre l'équation $P(x) = 0$. Le théorème de Howgrave-Graham donne une condition plus précise de ce qui est petit. Avant d'énoncer le théorème on introduit la notation suivante :

11. $[x]$ désigne l'entier le plus proche de x.
12. Cela découle du fait qu'on fait le changement $\|b_{k-1}^*\|^2 = \|b_k^*\|^2 + \mu_{k,k-1}^2 \|b_{k-1}^*\|^2$.
13. Nous admettons ce théorème, mais une démonstration, assez fastidieuse, est proposée dans [2].
14. Dans mon implémentation j'utilise la fonction proposée par numpy qui s'appuie sur une méthode de recherche de valeurs propres.

Définition 4.1. *Pour $Q(x) = \sum_{k=0}^{d} q_k x^k$, on note le vecteur ligne $b_Q = (q_0, q_1 X, q_2 X^2, ..., q_d X^d)$. De manière réciproque, d'un vecteur quelconque on peut déduire un polynôme associé à ce vecteur de cette manière.*

De manière équivalente, on construit un isomorphisme entre $\mathbb{R}_d[X]$ et $\mathbb{R}^{d+1}$.

Théorème 4.1: Howgrave-Graham

Si $x_0 < X$ est une solution de $P(x) = 0[N]$ avec $\|b_P\| < \frac{N}{\sqrt{d+1}}$ alors $P(x_0) = 0$.

Appliquer ce théorème directement à P est très contraignant. Mais on peut se contenter de l'appliquer à un polynôme, ayant les mêmes racines modulaires que P, mais dont les coefficients sont plus petits, ou encore tel que le vecteur associé est plus court. On pense alors aux bases réduites décrites précédemment. Il s'agit alors de construire une base qui engendre un réseau de polynomes avec les bonnes propriétés.

Pour engendrer un réseau de polynômes qui ont les mêmes racines modulaires que P, une première idée est d'utiliser la famille $(G_i(x))_{0 \leq i \leq d-1} = (Nx^i)_{0 \leq i \leq d-1}$. Tout entier est racine modulo N de ces polynômes. Par combinaison linéaire à coefficients entiers, un polynôme du réseau engendré par $(b_{G_0}, ..., b_{G_{d-1}}, b_P)$ a les mêmes racines modulo N que P.

Propriété 4.1

Le réseau engendré par $(b_{G_0}, ..., b_{G_{d-1}}, b_P)$ est de dimension $d+1$ et a pour déterminant

$$\det(L) = X^{\frac{d(d+1)}{2}} N^d$$

Notons G le polynôme associé à b_1 où $(b_1, ..., b_{d+1})$ est la base réduite donnée par l'algorithme LLL à partir de $(b_{G_0}, ..., b_{G_{d-1}}, b_P)$. On a alors le résultat suivant.

Théorème 4.2

Soit G et P les polynômes construits comme précédemment. On suppose que $X < \frac{1}{\sqrt{2}(d+1)^{\frac{1}{d}}} N^{\frac{2}{d(d+1)}}$.
Alors si x_0 est une solution de $P(x) = 0[N]$ avec $|x_0 < X$, alors x_0 est une solution de $G(x) = 0$ sur $\mathbb{Z}$.

Cette majoration de X est malheureusement très restrictive. Par exemple, pour $d = 3$ et $N = 1000$, on trouve $X < 1,4$ ... Dans la sous-partie suivante on cherchera à augmenter la borne X, ce qui constituera la méthode de Coppersmith.

4.2 Méthode de Coppersmith

La méthode de Coppersmith est fonctionne de manière similaire à celle de la partie précédente. La différence est le choix de la base de polynômes pour construire le réseau. La dimension du réseau sera aussi plus importante.

Avant de caractériser la base de polynômes considérée commençons par énoncer le théorème voulu.

Théorème 4.3: Coppersmith

Soit $0 < \varepsilon < 0,18(1 - \frac{1}{d})$. On suppose $X < \frac{1}{2} N^{\frac{1}{d} - \varepsilon}$. Si x_0 est une solution de $P(x) = 0[N]$ telle que $|x_0| < X$ alors x_0 peut être retrouvé en un temps polynomial.

Pour construire la base utilisée dans la méthode de Coppersmith on introduit un entier $h = \lceil \frac{d-1}{d^2 \varepsilon} + \frac{1}{d} \rceil$ [15].

Définition 4.2. *Soit $0 \leq i < d$ et $0 \leq j < h$. On note*

$$G_{i,j}(x) = N^{h-1-j} P^j(x) x^i$$

On considère alors, dans la méthode de Coppersmith, le réseau de dimension hd engendré par les vecteurs $(b_{G_{i,j}})_{0 \leq i < d, 0 \leq j < h}$.
Pour justifier l'intuition derrière ce choix de polynômes, remarquons notemment une chose :

15. Cet entier qui peut paraître arbitraire mais est choisit de manière à faire apparaître la puissance ε dans le théorème de Coppersmith.

Si x_0 est tel que $P(x_0) = 0[N]$ alors $G_{i,j}(x_0) = 0[N^h]$. En cherchant des racines modulo N^h, on augmente la borne de droite dans le théorème de Howgrave-Graham, ce qui rend plus facile le passage à une équation sur les entiers.

Le bon choix de ces polynômes relève aussi des majorations astucieuses obtenues lors des calculs de bornes dans la démonstration du théorème énoncé.

5 Conclusion

La méthode qui a permis d'attaquer le cryptosystème RSA se résume de la manière suivante.
La première étape est de transformer la recherche du mot chiffré en une équation polynomiale de la forme $P(x) = 0[N]$. On note x_0 une solution recherchée. En introduisant une bonne base de polynômes $G_{i,j}$, on peut engendrer un réseau de polynômes qui admettent aussi x_0 comme racine modulaire. Appliquer l'algorithme LLL de réduction des réseaux peut permettre (à condition que x_0 soit suffisement petit) de trouver en un temps polynomial, un polynôme G dont x_0 est racine. Il suffit alors d'énumerer les racines de G et voir lesquelles vérifient $P(x) = 0[N]$.

A Résultats de l'implémentation

Le programme de l'annexe D.1 est une implémentaion de RSA. En l'exécutant, on peut observer que les clés publiques et privées on bien été générées. Un test est fait avec le message :

$$m = 31415926535897932384626433832795028841971693993751 0$$

En suite, un exemple d'instance du programme fournit le chiffre suivant :

$c =$136003711371874689727902302836586333661509762681347626675366874875161386822501237131912933834053920071958235017363804701725911676284489307501473468642344170359741030823143623101518713334126920222779068769087744703141381162097183105100368157536078547574720930354477340984940194872777779750840635448271985320527343784848575839095017824568622697609331288639544867738733447524289321636524231384476018619432570600443208002798122986717644404746306124809339857616953554975537180361482570987420179393812653105451933840037481512886353127863227040605785083039414160404374336617780194532825633524272203603355486379434946045383 24

Dans tous les cas, on parvient à déchiffrer le message par la suite.

Dans l'annexe D.2, on implémente l'algorithme LLL. Le test est fait sur un petit réseau de dimension 3 engendré par les vecteurs $((1,1,1),(-1,0,2),(3,5,6))$. La base réduite obtenue est $((0,1,0),(1,0,1),(-1,0,2))$. On peut vérifier à la main que le résultat est correct. L'exemple suivant validera aussi l'implémentation de LLL.

Enfin, dans l'annexe D.3, on implémente les méthodes des sections 4.1 et 4.2 pour la recherches de petites racines modulaires d'un polynôme. On reprend l'exemple 19.1.6 proposé dans [5] avec le polynôme $P(x) = x^3 + 10^2 + 5000x - 222$ et $N = 10001$. Avec les deux méthode on trouve 4 comme solution. Cette solution convient effectivement.

B Pseudo-code de l'algorithme LLL

Algorithme 1 LLL

Input : Une base $b_1, ..., b_n$ d'un réseau L
Output : La base $b_1, ..., b_n$ transformée en une base réduite
for $i = 1, ..., n$ **do**
  $b_i^* = b_i$
  for $j = 1, ..., i-1$ **do**
    $\mu_{i,j} = \frac{\langle b_i, b_j^* \rangle}{B_j}$
    $b_i^* = b_i^* - \mu_{i,j} b_j^*$
  end
  $B_i = \|b_i^*\|^2$
end
$k = 2$
Appliquer **taille**$(k, k-1)$; `// (*)`
if $\frac{3}{4} B_{k-1} > B_k + \mu_{k,k-1}^2 B_{k-1}$ **then**
  Appliquer **lovasz**(k)
  Aller à (*)
end
for $l = k-2, ..., 1$ **do**
  Appliquer **taille**(k, l)
end
if $k = n$ **then**
  Arrêter l'algorithme
end
$k = k + 1$
Aller à (*)

Algorithme 2 Lovasz

Input : Un entier k
Output : Fait l'échange de b_k et b_{k-1} et calcule les nouveaux coefficients de Gram-Schmidt
$\mu = \mu_{k,k-1}$
$B = B_k + \mu^2 B_{k-1}$
$\mu_{k,k-1} = \frac{\mu B_{k-1}}{B}$
$B_k = \frac{B_{k-1} B_k}{B}$
$B_{k-1} = B$
Échanger b_k et b_{k-1}
for $j = 1, ..., k-2$ **do**
  Échanger $\mu_{k-1,j}$ et $\mu_{k,j}$
end
for $i = k+1, ..., n$ **do**
  $\mu' = \mu_{i,k-1}$
  $\mu_{i,k-1} = \mu_{k,k-1}\mu_{i,k-1} + (1 - \mu_{k,k-1}\mu)\mu_{i,k}$
  $\mu_{i,k} = \mu' - \mu\mu_{i,k}$
end
if $k > 2$ **then**
  $k = k - 1$
end

Algorithme 3 taille

Input : Un couple (k, l)
Output : Un vecteur b_k et des valeurs $\mu_{k,j}$ avec $j = 1, ..., l-1$ et $|\mu_{k,l}| \leq \frac{1}{2}$
if $|\mu_{k,l}| > \frac{1}{2}$ **then**
  $b_k = b_k - [\mu_{k,l}]b_l$
  for $j = 1, ..., l-1$ **do**
    $\mu_{k,j} = \mu_{k,j} - [\mu_{k,l}]\mu_{l,j}$
  end
  $\mu_{k,l} = \mu_{k,l} - [\mu_{k,l}]$
end

C Preuves

Preuve : Théorème 2.1 (RSA)

Soit $m \in \mathbb{N}$. Par hypothèse il existe $k \in \mathbb{N}$ tel que $ed = 1 + k\phi(N)$. Montrons que $m^{ed} - m = 0[p]$. Si $p|m$ c'est immédiat. Sinon, d'après le petit théorème de Fermat, on a $m^{ed} = m^{1+k\phi(N)} = m(m^{p-1})^{k(q-1)} = m[N]$. Alors de même $m^{ed} - m = 0[q]$ et puisque $p \wedge q = 1$, d'après le théorème des restes chinois, $m^{ed} - m = 0[pq]$. Soit $m^{ed} = m[N]$.

Preuve : Propriété 3.1

Soit $2 \leq i \leq n$. Par inégalité triangulaire dans (2), on a $\|b_i^*\|^2 \geq (\frac{3}{4} - \mu_{i,i-1}^2)\|b_{i-1}^*\|^2$ et d'après (1) $\|b_i^*\|^2 \geq \frac{1}{2}\|b_{i-1}^*\|^2$. On déduit alors d'une récurrence que pour tout $1 \leq j \leq i \leq n$, $2^{i-j}\|b_i^*\|^2 \geq \|b_j^*\|^2$ (*). Ainsi, pour $1 \leq i \leq n$

$$\begin{aligned}
\|b_i\|^2 &= \|b_i^* + \sum_{j=1}^{i-1} \mu_{i,j} b_j^*\|^2 \\
&= \|b_i^*\|^2 + \sum_{j=1}^{i-1} |\mu_{i,j}|^2 \|b_j^*\|^2 && \text{par orthogonalité} \\
&\leq (1 + \frac{1}{4}\sum_{j=1}^{i-1} 2^{i-j})\|b_i^*\|^2 && \text{d'après (*) et (1)} \\
&\leq (1 + \frac{1}{2}(2^{i-1} - 1))\|b_i^*\|^2 \\
&\leq 2^{i-1}\|b_i^*\|^2 && (**)
\end{aligned}$$

Donc, d'après (*) et (**), pour $1 \leq j \leq i \leq n$, $\|b_j\|^2 \leq 2^{j-1}\|b_j^*\|^2 \leq 2^{i-1}\|b_i^*\|^2$; d'où (3).
Puisque le déterminant est multilinéaire et alterné, on a $\det(L) = |\det(b_1^*, ..., b_n^*)|$. Puis, puisque $b_1, ..., b_n$ est orthogonale [a], $\det(L) = \prod_{i=1}^n \|b_i^*\|$. De plus, puisque la projection orthogonale contracte les normes : pour tout $1 \leq i \leq n$, $\|b_i\| \geq \|b_i^*\|$. Ainsi, avec (3) on a

$$\det(L) \leq \prod_{i=1}^n \|b_i\| \leq \prod_{i=1}^n 2^{\frac{i-1}{2}}\|b_i^*\| \leq 2^{\frac{n(n-1)}{4}} \det(L)$$

D'où (4). Enfin, d'après (3) avec $j = 1$, et en passant au produit pour $i = 1, ..., n$, on retrouve (5).

a. On peut évoquer pour cela l'interprétation géométrique du déterminant. Une preuve théorique est cependant envisageable.

Preuve : Propriété 2.2 (Terminaison de LLL)

Pour démontrer la terminaison de LLL nous introduisons le déterminant de Gram : $d_i = \det(\langle b_j, b_k \rangle_{1 \leq j,k \leq i}) = \text{Gram}(b_1, ..., b_i)$. Notons de plus que $b_i = b_i^* + \alpha$ avec α le projeté orthogonale de b_i sur $\text{Vect}(b_1, ..., b_{i-1})$. On montre alors par bilinéarité du produit scalaire et n-linéarité du déterminant que $\text{Gram}(b_1, ..., b_i^*) = d_i$. Or par orthogonalité, $\text{Gram}(b_1, ..., b_i^*)$ est diagonale par blocs ce qui permet de trouver que $d_i = d_{i-1}\|b_i^*\|^2$. On montre alors par récurrence que $d_i = \prod_{j=1}^i \|b_i^*\|^2$.
Posons ensuite $D = \prod_{i=1}^{n-1} d_i$. D'après l'expression des d_i, la valeur de D est modifiée si celle d'un $\|b_i^*\|^2$ l'est. On remarque dans le pseudo-code que ces valeurs ne sont modifiées que dans l'appelle de la fonction **Lovasz** : il existe $a < \frac{3}{4}$ tel qu'on ait le changement $\|b_{k-1}^*\|^2 = a\|b_{k-1}^*\|^2$ et $\|b_k^*\|^2 = \frac{1}{a}\|b_k^*\|^2$; les autres $\|b_i^*\|^2$ sont inchangés. Il en découle que d_{k-1} est réduit d'un facteur $< \frac{3}{4}$ et que les autres d_i sont inchangés. Et donc que D est réduit d'un facteur $< \frac{3}{4}$.
Nous admettons [a] que D est minoré par un réel strictement positif. Cela implique que l'on peut appliquer **Lovasz** qu'un nombre fini de fois. Or, puisque k est décrémenté dans **Lovasz** mais incrémenté sinon, cela implique que l'algorithme se termine.

a. Cela découle d'une minoration de la taille du plus court vecteur non nul d'un réseau.

Preuve : Théorème 2.1 (Howgrave-Graham)

Il suffit de remarquer, en vertu de l'inégalité de Cauchy-Schwarz, que

$$P(x_0) \leq \sum_{k=0}^d |a_k| X^k \leq \sqrt{d+1}\sqrt{\sum_{k=0}^d (a_k X^k)^2} = \sqrt{d+1}\|b_P\| < N$$

Preuve : Propriété 3.1

La famille $(G_0, ..., G_{d_1}, P)$ est libre par théorème des degrés étagés, donc $b_{G_0}, ..., b_{G_{d-1}}, b_P)$ qui en est l'image par l'isomorphisme qui fait la correspondance avec les vecteurs l'est aussi. De plus la famille de vecteur est représentée par la matrice triangulaire M suivante.

$$M = \begin{bmatrix} N & 0 & \dots & 0 & 0 \\ 0 & NX & \dots & 0 & 0 \\ \dots & \dots & \dots & \dots & \dots \\ 0 & 0 & \dots & NX^{d-1} & 0 \\ a_0 & a_1X & \dots & a_{d-1}X^{d-1} & X^d \end{bmatrix}$$

dont le déterminant est $\det(M) = \det(L) = X^{\frac{d(d+1)}{2}} N^d$.

Preuve : Théorème 3.2

D'après la majoration (5) de la propriété 2.1, on a

$$\|b_1\| \leq 2^{\frac{n-1}{4}} \det(L)^{\frac{1}{n}} = 2^{\frac{d}{4}} \det(L)^{\frac{1}{d+1}} = 2^{\frac{d}{4}} X^{\frac{d}{2}} N^{\frac{d}{d+1}}$$

D'après le théorème de Hograve-Graham, une conditions suffisante pour avoir le résultat voulu est que

$$2^{\frac{d}{4}} X^{\frac{d}{2}} N^{\frac{d}{d+1}} < \frac{N}{\sqrt{d+1}}$$

ce qui est équivalent à

$$X < \frac{1}{\sqrt{2}(d+1)^{\frac{1}{d}}} N^{\frac{2}{d(d+1)}}$$

Preuve : Théorème 3.3 : Coppersmith

En ordonnant les $G_{i,j}$ par degrés croissants, la matrice associée aux $b_{G_{i,j}}$ est triangulaire avec les coefficients diagonaux $N^{h-1-j}X^{jd+i}$. En prenant le produit de tout ces termes, on obtient : $\det(L) = N^{\frac{d}{2}(h(h-1)} X^{\frac{hd}{2}(hd-1)}$.
Or, toujours en notant b_1 le vecteur de la base réduite associée à $(b_{G_{i,j}})$, on à la majoration (5) qui donne :

$$\|b_1\| \leq 2^{\frac{dh-1}{4}} \det(L)^{\frac{1}{dh}} = 2^{\frac{dh-1}{4}} N^{\frac{h-1}{2}} X^{\frac{dh-1}{2}}$$

Or en appliquant le théorème de Howgrave-Graham, une condition suffisante pour passer à l'équation sur $\mathbb{Z}$ est $\|b_1\| < \frac{N^h}{\sqrt{dh}}$. En combinant les deux égalité et en simplifiant le résultat obtenu, puis en remplaçant h par son expression, on obtient la condition suffisante [a] :

$$\sqrt{2}(dh)^{\frac{1}{dh-1}} X < N^{\frac{1}{d} - \frac{d-1}{d(dh-1)}} < N^{\frac{1}{d}-\varepsilon}$$

En notant $c(d,h) = \sqrt{2}(dh)^{\frac{1}{dh-1}} = \sqrt{2}(\frac{d-1}{d\varepsilon} + 1)^{\frac{d\varepsilon}{d-1}}$ (en faisant l'approximation qu'on omet les parties entières), cette condition devient $X < \frac{1}{c(d,h)} N^{\frac{1}{d}-\varepsilon}$.
Pour avoir la borne du théorème, il suffit que $\frac{1}{2} \leq \frac{1}{c(h,d)}$ soit $(\frac{d-1}{d\varepsilon}+1)^{\frac{d\varepsilon}{d-1}} \leq \sqrt{2}$. Une étude de $x \mapsto (1+\frac{1}{x})^x$ permet de déduire que cette majoration est valide si $\varepsilon \leq 0,18(1 - \frac{1}{d})$. Dans ce cas on obtient bien la borne voulue.
La méthode emploie l'algorithme LLL (polynomial en hd et $N^h\|b_p\|$) et la recherche des solution d'un polynôme (aussi en temps polynomial). Puisque les initialisation des bases se fait aussi en temps polynomial, la méthode de Coppersmith a une complexité polynomiale.

a. C'est aussi à ce moment que se fait le choix de h pour voir apparaître ε

D Implémentation

D.1 RSA

```
1 import random
```

```python

# ALGORITHMES D'ARITHMETIQUE USUELS

#Calcule m^e mod n (expodentiation rapide)
def mod_exp(m, e, n):
    result = 1
    base = m % n
    while e > 0:
        if e % 2 == 1:
            result = (result * base) % n
        base = (base * base) % n
        e //= 2
    return result

#Calcule le r = pgcd(a,b) ainsi que des entiers u et v tels que au+bv = r
def euclide_etendu(a, b):
    u1, u2, u3 = 1, 0, a
    v1, v2, v3 = 0, 1, b

    while v3 != 0:
        q = u3 // v3
        v1, v2, v3, u1, u2, u3 = (
            u1 - q * v1,
            u2 - q * v2,
            u3 - q * v3,
            v1,
            v2,
            v3,
        )
    return u3, u1, u2


# GENERATION DE NOMBRES PREMIERS

# Test de temoin
def test_temoin(a, n):
    k = 0
    d = n - 1
    while d % 2 == 0:
        k += 1
        d //= 2
    x = mod_exp(a, d, n)
    if x == 1 or x == n - 1:
        return False
    for _ in range(k - 1):
        x = (x * x) % n
        if x == n - 1:
            return False
    return True

# Test de Miller-Rabin
def miller_rabin(n, k):
    if n <= 3:
        return n == 2 or n == 3
    if n % 2 == 0:
        return False

    for _ in range(k):
        a = random.randint(2, n - 2)
        if test_temoin(a, n):
            return False
    return True


```

```
# Generateur de nombres premiers
def generate_prime(bits, k=100):
    while True:
        n = random.getrandbits(bits)
        n |= (1 << (bits - 1)) | 1
        if miller_rabin(n, k):
            return n

#Genere deux nombre premiers distincts
def generate_primes(bits = 1024):
    p = generate_prime(bits)
    q = generate_prime(bits)
    while p==q :
        q = generate_primes
    return p,q

# GENERATION CLES RSA


class RSA :

    def __init__(self) :
        self.generate_keys()

    #Cle publique : (e,n)
    #Cle privee : d
    def generate_keys(self, bits = 1024):
        p,q = generate_primes()
        self.n = p*q
        phi_n = (p-1)*(q-1)
        while True :
            e = random.randint(2, phi_n)
            r,u,v = euclide_etendu(e,phi_n)
            if r == 1:
                self.e = e
                self.d = u % phi_n
                break


# ENCODAGE D'UN MESSAGE

    def encode(self, m):
        return mod_exp(m, self.e, self.n)

# DECODAGE D'UN MESSAGE
    def decode(self, c):
        return mod_exp(c, self.d, self.n)


if __name__=="__main__":
    rsa = RSA()
    print('Cl  publique n  : ', rsa.n)
    print("\n")
    print('Cl  publique e  : ', rsa.e)
    print("\n")
    print('Cl  priv e d : ', rsa.d)
    print("\n")

    #premieres decimales de pi
    m = 31415926535897932384626433832795028841971693993751O
    print('Message : ', m)
    print("\n")

    c = rsa.encode(m)
    print('Message cod  : ', c)
    print("\n")
```

```
134
135     m2 = rsa.decode(c)
136     print('Message d cod   : ', m2)
```

D.2 LLL

```
1  import numpy as np
2
3
4  class Basis:
5
6      # Les constructeurs de cet objet sont : une matrice repr sentant la base, une
       matrice contenant les mu_{i,j} et un vecteur contenant les B_i
7      def __init__(self, basis):
8          self.basis = basis
9          self.gram_schmidt()
10
11     #Calcule,    partir d'une base, les coefficients de Gram-Schmidt
12     def gram_schmidt(self):
13         basis = self.basis
14         n = len(basis)
15         m = len(basis[0])
16         orthogonal_basis = np.copy(basis)
17
18         #Matrices contenant les coefficients de Gram-Schmidt
19         mu = np.zeros((n,m))
20         B = np.zeros(n)
21         B[0] = np.dot(basis[0],basis[0])
22
23         for i in range(1,n):
24             for j in range(i):
25                 mu[i,j] = np.dot(basis[i], orthogonal_basis[j])/np.dot(
       orthogonal_basis[j], orthogonal_basis[j])
26                 orthogonal_basis[i] -= mu[i,j]*orthogonal_basis[j]
27             B[i] = np.dot(orthogonal_basis[i], orthogonal_basis[i])
28         self.mu = mu
29         self.B = B
30
31
32     # Echange b_k et b_{k-1} et calcule les nouveaux coefficients de Gram-Schmidt
33     def lovasz (self, k):
34         n = len(self.basis)
35         mu0 = self.mu[k,k-1]
36         b = self.B[k] + mu0*mu0*self.B[k-1]
37         self.mu[k,k-1] = mu0*self.B[k-1]/b
38         self.B[k] = self.B[k-1]*self.B[k]/b
39         self.B[k-1] = b
40         self.basis[[k, k-1]] = self.basis[[k-1, k]] # Echange b_k et b_{k-1}
41
42         # Actualisation des coefficients de Gram-Schmidt
43         for j in range(k-1):
44             self.mu[k,j], self.mu[k-1,j] = self.mu[k-1,j], self.mu[k,j]
45         for i in range(k+1, n):
46             mu1 = self.mu[i,k-1]
47             self.mu[i,k-1] = self.mu[k,k-1]*self.mu[i,k-1]+(1-self.mu[k,k-1]*mu0)*
       self.mu[i,k]
48             self.mu[i,k] = mu1 - mu0*self.mu[i,k]
49
50         return max(k-1,1)
51
52
53     # Op re la transformation b_k = b_k - [mu_{k,l}]b_l et actualise les coefficiens
        de Gram-Schmidt
54     def taille(self, k, l):
55         if abs(self.mu[k,l])> 0.5+ 10**(-12) :
```

```
            # Le + 10^(-12) permet de r soudre un disfonctionnement dans le cas d'
        galit
            self.basis[k] = self.basis[k] - round(self.mu[k,l])*self.basis[l]

            # Actualisation des coefficients de Gram-Schmidt
            for j in range(l):
                self.mu[k,j] = self.mu[k,j] - round(self.mu[k,l])*self.mu[l,j]
            self.mu[k,l] = self.mu[k,l] - round(self.mu[k,l])




    # Applique l'algorithme LLL    la base
    def lll(self):
        n = len(self.basis)
        self.gram_schmidt()
        k = 1
        while (k < n):
            self.taille(k,k-1)
            if 0.75*self.B[k-1] > self.B[k] + self.mu[k,k-1]*self.mu[k,k-1]*self.B[k
    -1]:
                k = self.lovasz(k)
                continue
            for l in range(k-2, -1, -1):
                self.taille(k,l)
            k+=1


if __name__ == "__main__":

    '''
    a = np.array([[47.,215.],[95.,460.]])
    b = Basis(a)
    print("Base : \n ", b.basis)

    b.lll()
    print("Base r duite : \n", b.basis)
    '''

    a = np.array([[1.,1.,1.],[-1.,0.,2.],[3.,5.,6.]])
    b = Basis(a)
    print("Base: \n", b.basis)
    b.lll()
    print("Base r duite : \n", b.basis)
```

D.3 Coppersmith

```
import numpy as np
from numpy.polynomial import Polynomial
from LLL import Basis
from rsa import RSA


#Opere la transformation polyn me-vecteur exposee dans le dossier
def poly_to_vect(poly, X, size=-1):
    d = poly.degree()
    n = max(size,d+1)
    vect = np.zeros(n)
    for i in range(n):
        if i<d+1:
            vect[i] = poly.coef[i] * X ** i

    return vect

# Opere la transformation polynome-vecteur exposee dans le dossier
```

```
def vect_to_poly(vect, X):
    n = len(vect)
    poly = Polynomial([0])
    for i in range(n):
        monomial = Polynomial([0] * i + [1])
        poly += (vect[i] / (X ** i)) * monomial

    return poly


# Construit la base de vecteurs pour la methode de la section 4.1
def poly_to_basis2(poly, N, X, h):
    d = poly.degree()
    a = np.zeros((d * h, d * h))
    for j in range(h):
        for i in range(d):
            monomial = Polynomial([0] * i + [1])
            G = (N**(h-1-j))*(poly**j)*monomial
            vect = poly_to_vect(G, X, size=h*d)
            a[i + d * j] += vect
    return Basis(a)

# Construit la base de vecteurs pour la methode de Coppersmith
def poly_to_basis1(poly, N, X):
    d = poly.degree()
    a = np.zeros((d+1 , d+1 ))

    for i in range(d):
        monomial = Polynomial([0] * i + [1])
        G = N*monomial
        vect = poly_to_vect(G, X, d+1)
        a[i] = vect
    a[d] = poly_to_vect(poly, X)
    return Basis(a)




poly = Polynomial([-222, 5000,  10, 1])
print(poly)
n = 10001
X = 10
h = 3

print('\n')

base1 = poly_to_basis1(poly, n, X)
base1.lll()
p1 = vect_to_poly(base1.basis[0],X)
print(p1)
print(p1.roots())

print('\n')

base2 = poly_to_basis2(poly, n, X, h)
base2.lll()
p2 = vect_to_poly(base2.basis[0],X)
print(p2)
print(p2.roots())
```

E Bibliographie

[1] R.L. Rivest, A. Shamir, L. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Comm. ACM 21(2) (1978), 120–126.

[2] A. K. Lenstra, H. W. Lenstra, Jr., and L. Lovász, Factoring polynomials with rational coefficients, Math. Ann. 261 (1982), 515–534.

[3] D. Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. Journal of Cryptology, 10(4), 233—260 (1997).

[4] N. Howgrave-Graham. (1997). Finding Small Roots of Univariate Modular Equations Revisited. Pages 131–142 of : Darnell, Michael (ed), 6th IMA International Conference on Cryptography and Coding. LNCS, vol. 1355. Cirencester, UK : Springer, Heidelberg, Germany.

[5] S. Galbraith. (2012). Coppersmith's Method and Related Applications. Pages 397-416 of : "Mathematics of Public Key Cryptography". Cambridge University Press.