Binôme : Bollet – Grenier Année 2000

CODAGE RSA

SOMMAIRE

INTRODUCTION

I – Présentation du programme

1) Le principe du programme, fonctionnement général

2) Mode d’emploi et fonctionnalités (options fournies)

II – Algorithme Détaillé et outils mathématiques utilisés

1) Détermination des clés et du modulo.

2) Cryptage/Décryptage et autres fonctions

CONCLUSION

INTRODUCTION

Le but de ce projet était de faire un programme de codage RSA utilisant la technique de la signature et l'utilisation des grands nombres (opérations élémentaires, modulo et recherche de nombres premiers) pour nous permettre de mieux assimiler les outils mathématiques qu'on l'on peut employer lors d'une telle conception.

Après avoir cherché par nous même des algorithmes, nous avons fait des recherches sur Internet qui se sont révélées fructueuses puisque nous avons trouvé un site qui exhibait un programme RSA (écrit en C) dont nous nous sommes inspirés. Nous avons conservé le langage C puisque nous le connaissons bien et que l'utilisation d'Octave ne nous était pas très claire. Après avoir détaillé l'algorithme du programme fourni, nous l'avons modifié pour nos besoins (mais nous avons aussi rectifié ses erreurs !).

Nous allons, dans une première partie, présenter le principe général du programme ainsi que son utilisation. Ensuite nous allons expliquer les algorithmes utilisés pour la mise en œuvre du projet ainsi que les outils mathématiques utilisés.

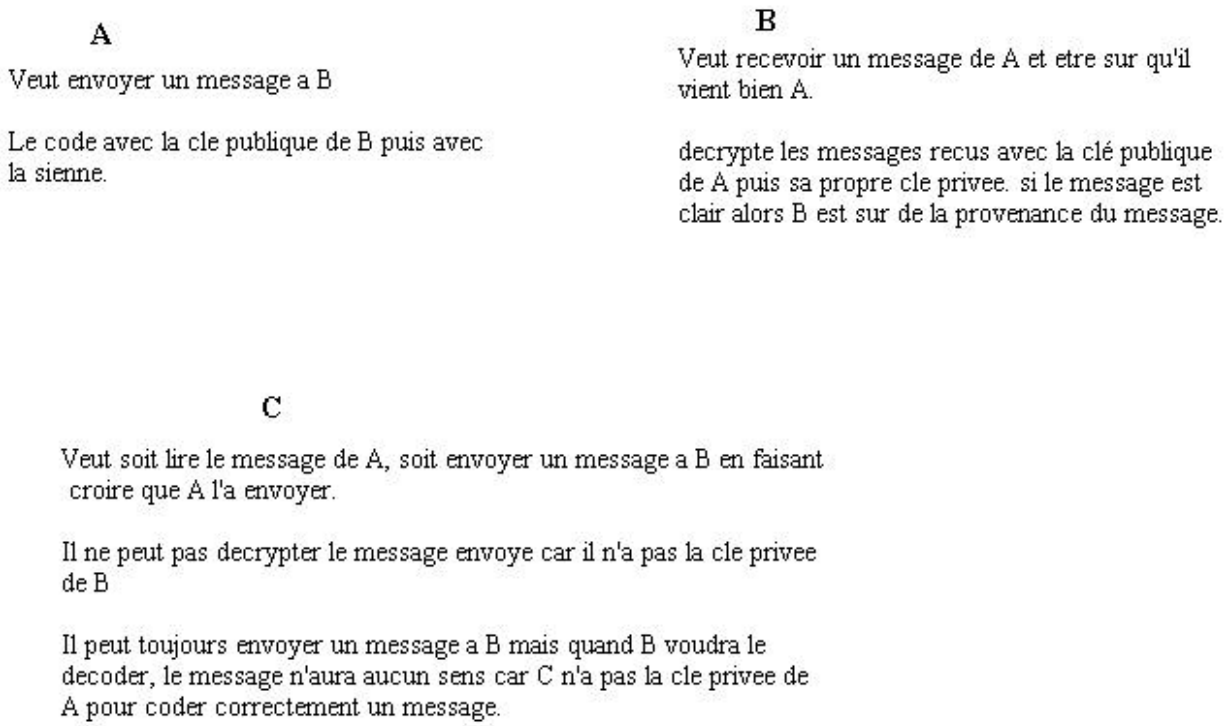
I – Présentation du programme

1) Le principe du programme, fonctionnement général

Le but de ce programme est le suivant : supposons qu'un individu A veut envoyer un message très important et confidentiel à un individu B sans qu'un individu C ne puisse le lire s'il parvient à l'intercepter, ce programme cryptera le dit message de façon à ce que B (et B seulement) puisse le lire. Plus encore, B aura la certitude que le message vient bien de A.

Pour ce faire, le programme utilise un système de codage à clé publique : le système RSA (du nom de ses créateurs). Chaque utilisateur se voit attribuer une clé publique (distribuée à tout le monde) et une clé privée (que seuls chacun des utilisateurs connaissent) qui permettent respectivement de coder et décoder un message. Ainsi, A codera son message avec la clé publique de B pour que celui-ci soit le seul à pouvoir décoder le message. Cependant ce système permettrait à C d'envoyer un faux message à B. C'est pourquoi A crypte à nouveau le message avec sa clé privée (on dit alors que A signe le message). Ainsi tout le monde peut décoder le message final (avec la clé publique de A) mais seul B peut le décrypter totalement avec sa clé privée.

Résumons la situation par un petit dessin :



2) Mode d'emploi et fonctionnalités (options fournies)

Les sources du programme sont présentées dans un fichier : Bollet_Grenier.tar.gz. Pour les décompresser, il faut, sous un « UNIX », utiliser la commande « tar –xvvzf Bollet_Grenier.tar.gz ». Ceci créera un dossier Bollet_Grenier contenant tous les fichiers nécessaires à l'utilisation.

Il faut aller dans le répertoire (commande « cd Bollet_Grenier »), et utiliser la commande « make » pour générer un exécutable du nom de rsa.

Note : La commande « make » génère un certain nombre de fichiers inutiles à l'utilisation du programme. Il peut donc être judicieux de taper la commande « make clean » avant de lancer le programme. Ceci effacera les fichiers devenus inutiles (sauf bien entendu les sources). Pour ne laisser que les sources (et enlever les fichiers créés par le programme), la commande « make propre » est tout aussi utile.

Pour lancer le programme il suffit alors de taper « ./rsa ».

Un menu s'offre alors a vous. Il permet :

1) de définir un nouvel utilisateur
2) d'avoir accès aux clés publiques de chaque utilisateur et même de voir votre propre clé privée.
3) De coder un message
4) De Décoder un message
5) De quitter le programme

1) Nouvel utilisateur :

Si vous choisissez cette option, il vous sera demandé

- le nom de l'utilisateur (attention : seuls les 8 premiers caractères seront pris en compte)

- un mot de passe (pour pouvoir d'identifier. Attention l'oubli de ce mot de passe entraîne l'impossibilité de coder ou décoder. Il faut donc bien s'en souvenir)

- Le nombre de chiffre (en base 256) que doit contenir le nombre de cryptage/décryptage. Plus le nombre de chiffres est élevé, plus il sera difficile de « casser » le codage. Cependant, plus le temps de cryptage/décryptage sera long. Pour avoir un aperçu du programme, 10 à 20 chiffres suffisent amplement ; pour envoyer en condition réelle un message, il ne sera pas ridicule de demander 100 chiffres.

- Le rapport entre cryptage / décryptage. Ce rapport déterminera le nombre de chiffres de la clé publique. Cette option est inutile si vous ne vous êtes pas mis

d'accord avec les autres personnes avec lesquelles vous devez communiquer. Si vous devez surtout recevoir des messages, il peut être intéressant d'entrer ici un grand nombre (près de cent comme 96) et de demander aux expéditeurs d'entrer un petit chiffre (comme 6). Ainsi votre temps de décryptage sera très court mais le temps de cryptage beaucoup plus long.

- Le nombre de tests de Miller-Rabin. Ce test permet de tester si le nombre utilisé pour le codage est bon. Si le nombre de tests est insuffisant, vous risqueriez d'avoir certains problèmes. Il est conseillé d'entrer 5 pour avoir une bonne probabilité. Encore une fois, plus le nombre sera grand, plus le temps pour trouver une clé sera long. Il est peu judicieux d'entrer un nombre plus grand que 9.

Un fichier « nom».cle sera alors créé, où « nom» est le nom de l'utilisateur. Il contiendra outre le nom de l'utilisateur, son mot de passe, et ses différentes clés. Pour lire ce fichier, il faut utiliser :

2) Lecture d'un fichier .cle

Il vous sera demandé le nom de l'utilisateur dont vous voulez lire le fichier .cle ainsi qu'un mot de passe. Si le mot de passe correspond bien avec celui du fichier, vous aurez accès à toutes ses clés, sinon, vous aurez accès seulement aux clés publiques. La connaissance en soit de ces clés ne vous apporte rien, sauf si vous voulez coder ou décoder à la main (ou via un autre programme de codage rsa) un message.

3) Coder un message

Pour coder un message, il est indispensable que l'expéditeur ET le destinataire du message aient déjà un fichier « .cle ». Si ce n'est pas le cas, revenir dans l'option 1.

Il sera demander le nom de l'expéditeur, son mot de passe ainsi que le nom du destinataire. Attention : si le mot de passe est erroné, vous retournerez immédiatement au menu principal sans avoir pu coder votre message.

Vous devrez ensuite entrer le nom du fichier à crypter. Le fichier en question devra avoir au préalable enregistré avec l'extension « .txt » dans le répertoire où se trouve rsa. La liste des fichiers est affichée.

Vous pourrez alors choisir le nom du fichier destination. Par défaut ce sera le même que le fichier source l'extension « .txt » étant changée en « .rsa ».

Vous verrez alors votre message s'afficher petit à petit. Un morceau est affiché dès qu'il est codé. Une fois votre message totalement affiché (et donc codé), un indicateur vous dira en combien de morceaux il a été découpé. Enfin, le résultat sera découpé à nouveau pour apposer votre signature sur chaque morceau. Si les clés utilisées sont sensiblement de même taille, le découpage sera à peu près identique et vous saurez donc à l'avance le nombre de morceaux qui seront signés.

4) Décoder un message

De même que pour coder, il faut que les utilisateurs soient définis, que le destinataire s'identifie grâce à son mot de passe et qu'il indique de qui il pense que le message provient.

Encore une fois, si le mot de passe est incorrect, un retour forcé au menu sera au programme !

Le nom demandé sera en principe celui du fichier original suivi de l'extension « .rsa » au lieu de « .txt ».

Le programme commencera par enlever la signature par morceaux. Le nombre de morceaux est celui qui avait été indiqué lors de la signature.

Une fois cette étape finie, le message commencera à se décoder petit à petit. Si effectivement un message apparaît, il s'agit bien de celui attendu ; si une suite incohérente de symboles apparaît, le message est un faux.

5) Quitter

Appuyer sur la touche f du clavier permet de quitter le programme.

II – Algorithme Détaillé et outils mathématiques utilisés

1) Détermination des clés et du modulo.

Pour coder un message m, on applique la simple formule $c = m^d$ modulo n où d est la clé publique et n un modulo. n est un nombre (grand) composé de la multiplication de deux nombres premiers p et q. Trouver p et q permet de « casser » le code, chacun de ces nombres devra être grand. Le nombre de chiffres de n est celui déterminé par l'utilisateur. d est un nombre premier avec Ö(n) où Ö est la fonction d'Euler (Ö(n) = (p-1)*(q-1)).

Pour décoder un message codé x, on applique la formule $m = x^e$ modulo n ou e est l'inverse de d modulo n (c'est la clé privée).

Pour signer un message coder c, on applique toujours les mêmes formules, sauf que la clé publique est remplacée par la clé privée et vice versa.

Le grand problème est donc de trouver p, q (puisque n = p*q), d, et e. Dans tout le projet on a travaillé en base 256. Le choix de cette base a été motivé par deux arguments : comme nous devions manipuler des grands nombres, il était évident qu'une base > 10 allait nous permettre de manipuler des nombres comportant moins de chiffres (d'où un temps de calcul plus court) ; de plus nous devions coder des messages composés de caractères qui s'écrivent naturellement en C en base 256 (code ASCII). Cette base nous permet donc d'éviter des conversions en manipulant directement des caractères.

La structure utilisée est donc un entier contient la longueur du nombre (nombre de chiffres) et une liste chaînée des caractères (⇔ chiffres en base 256).

a) Détermination de p et q

Le nombre de chiffre voulu pour n étant connu (fixé par l'utilisateur = n.lon), on cherche un nombre premier de moitié moins de chiffre (n.lon/2).

Pour ce faire, on appelle une fonction « Hasard » dans laquelle on passe en paramètre un nombre de (n.lon/2 + 1) chiffre dont le chiffre de poids fort est à un (en fait pour créer ce paramètre, on crée simplement un nombre initialisé à 1 que l'on décale de (n.lon/2+1) ; décalage ⇔ multiplie par 256 à la puissance).

La fonction Hasard créé alors automatiquement un nombre de même longueur que celui passé en paramètre (donc n.lon/2 + 1) en forçant le chiffre de poids fort à une valeur strictement inférieure à celle du nombre initial. Le chiffre de poids (n.lon/2 + 1) est donc forcément nul (sur le nombre déterminé), ce qui nous donne un nombre de n.lon/2 chiffres au maximum.

Une fois ce nombre aléatoire obtenu, on cherche le nombre premier le plus proche (supérieurement) à ce nombre. Dans un premier temps, on vérifie qu'il n'est pas pair. S'il l'est, on lui ajoute un.

Ensuite, on aurait pu tester le nombre directement par la méthode de Miller-Rabin. Mais ce test étant long (en plus de 8*(n.lon/2)² opérations), on préfère vérifier d'abord s'il n'est pas divisible par 3 ou par 5 (test en n.lon/2 opérations). Ensuite on se sert du théorème de Fermat : en effet si p est premier, alors pour tout a < p, $a^{p-1}=1[p]$, en particulier pour a = 2. Un site sur Internet nous affirme que si un nombre passe tous ces tests, il a plus de 75% de chances d'être premier (démonstration non présente sur le site). Enfin, s'il a passé avec succès ces tests, on lui applique la méthode de Miller-Rabin un nombre de fois défini par l'utilisateur. S'il passe encore avec succès, on estime qu'il est premier. Dès qu'un de ces tests n'est pas concluant, on ajoute 2 au nombre (pour avoir encore un nombre impair !), et on repasse les tests sans aller plus loin.

Les tests de divisibilité par 2, 3 et 5 sont évidents. Le calcul de a^k [p] (ici k = p-1) l'est beaucoup moins : voici donc son algorithme.

b) Calcul d'exponentiation modulaire

Pour ce calcul, comme pour tous les suivants, on profite du choix de la base pour simplifier les calculs : en effet la base 256 à été choisi en informatique car en binaire 255 = $(11111111)_b$. On peut alors facilement se servir de l'écriture binaire de chaque chiffre pour créer des opérations rapides.

Les opérations que nous avons utilisées n'étant pas intuitives, nous allons les expliquer à travers un exemple concret : $150^{50}[1999]$.

Or 50 = $(00110010)_b$. Donc $150^{50}[1999] = 150^{32+16+2}[1999] = 150^{32}150^{16}150^{2}[1999]$. Or 32,16 et 2 sont des puissances de 2 donc $150^{50}[1999] = (150^{16}150^{8}150)^{2}[1999]$. On recommence avec 16 et 8 :

Finalement : $150^{50}[1999] = ((150^{2}150)^{8} * 150)^{2}[1999]$.

Algorithme :

On initialise le résultat d à 1.

On effectue une boucle sur la longueur de b (nombre de chiffres dans b. Ici on passe une seule fois dans la boucle) :

On effectue une boucle 8 fois (car b est au plus égal à 255 donc 8 bits) :
A chaque passage, d → (d*d) [n].
Puis si le bit correspondant est à 1, d →(a*d)[n]

Application :

d =1
Pour j = 0 à 0
    Pour i = 128 à 1 (i = i/2 à chaque pas)
        Pour i = 128 : d = 1*1[1999] = 1[1999]
        Pour i = 64 : d = 1*1[1999]
        Pour i = 32 : d = 1*1[1999]
                d = 150*1[1999] (car 50 = 32 +16 + 2)
        Pour i = 16 : d = 150^2[1999]
                d = 150^2*150[1999]
        Pour i = 8 : d = $(150^2*150)^2$[1999]
        Pour i = 4 : d = $(150^2*150)^4$[1999]
        Pour i = 2 : d = $(150^2*150)^8$[1999]
                d = $((150^2*150)^8 * 150)$[1999]
        Pour i = 1 : d = $((150^2*150)^8 * 150)^2$[1999] = 150^{50}[1999].

Le seul problème restant est donc de calculer a*b[n].

Il est évident que si a et/ou b est plus grand que n, on peut toujours calculer a_1 = a[n] et b_1 = b[n] et se ramener au problème a_1*b_1[n].

A partir de là, toujours avec la décomposition en binaire de b, on calcule par additions successives.

Prenons l'exemple suivant : 150 * 50[1999] = 150 * (32+16+2)[1999] = (150*32 + 150*16 + 150*2)[1999] = ((150*2+150)*8+150)*2[1999].

On est ainsi passé d'une multiplication à des additions (plus simple).

Remarquons qu'à ce stade, en fait on a simplement effectué a*b et ensuite on a appliqué le modulo. On a donc l'algorithme de notre multiplication rapide (en $O(n.lon^2)$). La multiplication indiquée dans le cours est en $O(n.lon^{1.5})$ donc a priori plus rapide. A l'implémentation, le contraire a été prouvé : en effet, la multiplication du cours imp lique de définir (récursivement) 9 grands nombres (a0,a1,a2,b0,b1,b2,p0,p1 et p2). Comme l'appel est récursif, on définit de nombreuses variables temporaires. Or la structure utilisée impose une réservation de mémoire pour chaque variable qui est relativement lente. A l'utilisation (par comparaison des temps de calculs avec l'une puis l'autre des méthodes), la méthode du cours est applicable à notre structure seulement si les chiffres comportent au moins 200 chiffres (en base 256 – nombre estimé par extrapolation les temps de calculs étant si longs que l'essai n'a pas été fait) ! Et encore, quand bien même la méthode du cours est plus efficace, elle utilise beaucoup plus de ressources que la nôtre (99,8% du CPU sur igualdad contre 74% pour notre méthode !).

Il reste enfin à savoir calculer a[n]. On peut remarquer préalablement que la division du cours est elle aussi contre-indiquée pour notre utilisation (par tests comparatifs encore une fois). Nous avons trouver un moyen de contourner cette difficulté (qui, s'il ne donne pas de résultats plus rapides, demande moins de ressources).

En réalité, on utilise un principe similaire à la multiplication « à l'envers ». Prenons un exemple en base 10 pour expliquer la méthode :

On cherche 2500[20]. On calcule la différence du nombre de chiffres de 2500 par rapport à celui de 20. Ici 4-2=2. On décale alors 20 de 2 chiffres (soit 20*10*10 = 2000). On compare 2500 à 2000. 2500 > 2000, donc on retire 2000 à 2500. Il reste 500.

Mathématiquement parlant : 2500 = (2500- 20*10*10)[20] = 500[20]
On réitère l'opération : 500 < 2000 donc on passe à 20*10. 500 > 200 donc
2500[20] = 500[20] = (500-200)[20] = 300[20] = 100[20] et 100 < 200 donc on recommence avec 20.
2500[20] = 500 (=2500-20*10*10)[20] = 100 (=500-20*10-20*10)[20] = 0 (=100-20*5)[20].
On est donc bien arrivé au résultat 2500[20] = 0[20].

Cette méthode s'avère très lente si le nombre de chiffres entre a et n est très important, mais, pour nos besoins, elle est (presque) aussi rapide que la division rapide et prend moins de ressources. Pour chiffrer la différence de temps, un essai avec un nombre n de 76 chiffres (en décimal) a donné 3 secondes de plus avec notre méthode qu'avec la division rapide pour crypter un fichier (sur un temps total de 2 minutes 45 !).

Remarque : avec cette méthode retrouver le quotient de a par n est un jeu d'enfant : à chaque soustraction, il suffit d'ajouter 1 décalé d'autant de chiffres que n au quotient (en initialisant le quotient à 0).

La détermination de p et q, faisaient appel à l'exponentiation modulaire, mais aussi à la méthode de Miller-Rabin.

c) Méthode de Miller-Rabin

Cette méthode nous permet de savoir (avec une certaine probabilité) si un nombre n est premier ou non. Encore une fois, elle utilise le théorème de Fermat avec des nombres pris au hasard (plus petits que n-1 !).

On calcule m = n-1, on génère un nombre aléatoire a plus petit que m, et on calcule a^m[n]. Si le résultat est différent de 1, n n'est pas premier à coup sur. Il a été montré que la réitération de ce calcul avec 5 nombres aléatoires différents donne un nombre premier avec une grande probabilité (on peut remarquer qu'on a déjà calculé 2m[n] avant d'appliquer cette méthode … on applique donc en réalité une fois de plus que ce que l'utilisateur a demandé).

On a alors une méthode pour trouver p et q. Pour avoir n, il suffit de multiplier p par q avec la méthode déjà vue. Il reste encore à trouver les clés publiques et privées.

d) détermination des clés.

On veut trouver dans un premier temps un nombre d premier avec Ö(n). On détermine la longueur de d selon la rapidité cryptage/décryptage voulue par l'utilisateur (en effet si d est grand, le temps de cryptage sera plus long). On détermine un nombre aléatoire x de la longueur voulue (toujours par la même méthode) et on cherche l'entier premier avec Ö(n) le plus proche (supérieurement). On calcule donc le pgcd entre x et Ö(n) et s'il n'est pas égal à un, on ajoute 1 à x et on réitère l'opération.

Pour trouver e, il suffit de calculer l'inverse de d modulo Ö(n).

e) calcul du pgcd.

La propriété utilisée est celle (montrée comme programme de référence en algorithme !) du pgcd : pgcd(a,b) = pgcd(a-b,b) si a > b en posant pgcd (a,0) = a.

Pour aller plus vite, on utilise l'algorithme d'Euclide, et on obtient directement pgcd(a,b) = pgcd(a%b,b) où a%b représente le reste de la division euclidienne de a par b.

Par exemple pour 20 et 14 :
20 = 14 * 1 + 6 pgcd(20,14) = pgcd(6,14)
14 = 6 * 2 + 2 pgcd(20,14) = pgcd(6,2)
6 = 2 * 3 + 0 pgcd(20,14) = pgcd(2,0) = 2

Le pgcd est alors 2 (le dernier reste non nul (s'il existe) ⇔ le dernier diviseur).

La propriété d = pgcd(a,b) = pgcd(a-b,b) est évidente : en effet, d = pgcd(a,b) divise a et b (en temps que pgcd), donc d divise a-b et b. Supposons alors qu'il existe d' qui divise a-b et b. Il divise alors (a-b+b) = a et b. d étant le pgcd entre a et b, d' divise d. Ceci prouve bien que d est le pgcd entre a-b et b. En itérant pgcd(a,b)=pgcd(a-b,b)=pgcd(a-2*b,b) … on obtient pgcd(a%b,b).

f) calcul de l'inverse modulaire

On applique l'algorithme d'Euclide tout en faisant une remarque intéressante pour « remonter » les calculs.

Prenons un exemple : calculons l'inverse de 13 modulo 23.
On descend comme dans Euclide :
23 = 13 * 1 + 10 (on retient 1 et 10)
13 = 10 * 1 + 3 (on retient 1 et 3)
10 = 3 * 3 + 1 (on retient 3 et 1)

Pour remonter, Euclide nous donne la méthode suivante :

1 = 10 – 3*3	= 10*1 – 3*3
1 = 10 – 3*(13-10*1) = -3*13 + 4 * 10	= -3 * 13 + (3*1 + 1) * 10
1 = -3*13 + 4*(23-13*1) = -7 * 23 + 4 * 23	=> le –7 vient de -3 + 4*(-1) = -(3 + 3*1 + 1)

Bien entendu, quand (comme dans ce cas) l'inverse calculé y est négatif, on pose e = Ö(n) - |y|.

Remarque : Nous avons aussi mis en œuvre pour nos calculs des fonctions de calculs (comme l'addition des grands nombres, la conversion des nombres de la base 256 à la base 10 et inversement) dont les algorithmes sont intuitifs et ne seront donc pas présentés ici.

2) Cryptage/Décryptage et autres fonctions

Les outils mathématiques utilisés ici ont déjà été explicités dans la partie précédente. Nous nous contenterons donc de rappeler le principe du codage RSA avec signature.

Aux utilisateurs A et B sont associés les fichiers A.cle et B.cle. Chaque fichier contient le nom de l'utilisateur, son mot de passe ainsi que les nombres n, d et e (que nous noterons avec des indices). La connaissance (et donc l'utilisation) de nombre e_i ne peut se faire qu'à la seule condition que le mot de passe associé soit correct.

Ainsi, si l'utilisateur A veut envoyer le message m d'un fichier toto.txt à B sans que C ne puisse le lire ni envoyer de faux message à B, le message sera codé de la façon suivante :

Soit n_i.lon le nombre de chiffres (en base 256) de n_i. Le message m sera découpé en morceaux m_j de taille $(n_A.lon / 2) - 1$. Le dernier morceau n'aura pas forcément cette taille si le fichier original ne contient pas un nombre de caractères multiple de $(n_A.lon / 2) - 1$.Chaque m_j subira alors la transformation suivante : $x_j = m_j^{dB}[n_B]$. Chaque x_j sera alors de la taille n_B au maximum et sera enregistré dans un fichier toto.rs1. A leur tour les x_j seront transformés en $y_j = x_j^{eA}[n_A]$. Chaque morceau sera aussi de taille constante et enregistré dans un fichier toto.rsa. Ce fichier contiendra une entête permettant de retrouver le nom du fichier original ainsi que la taille totale du fichier (qui sera donc un multiple de la taille des morceaux). On peut noter que seul A dispose de eA, la provenance du fichier est donc sécurisée. Le fichier toto.rs1 est, bien entendu, immédiatement effacé.

La seule façon de décoder ce message est, pour B, est de faire subir à chaque morceau la transformation inverse : $x_j = y_j^{dA}[n_A]$.

En effet, $y_j^{dA}[n_A] = (x_j^{eA})^{dA}[n_A] = x_j^{1kö(nA)}[n_A] = x_j*(x_j^{ö(nA)})^k[n_A]$.

Or pour tout $w < n_A$, $w^{ö(nA)} = 1[n_A]$ (d'après le théorème de Fermat).

Donc $y_j^{dA}[n_A] = x_j[n_A]$. On a pris $x_j < n_A$ donc on a bien retrouvé le fichier toto.rs1 (qui est alors appelé toto.tp). De même avec $m_j = x_j^{eB}[n_B]$.. B étant la seule personne connaissant e_B, lui seul peut décoder le message (qui sera enregistré dans un fichier toto.ok).

CONCLUSION

Nous avons atteint l'objectif que nous nous étions fixé à savoir de faire un programme de codage basé sur le système RSA avec signature. Nous avons ainsi pu confronter plusieurs algorithmes pour mettre en place les grands nombres.

Par ailleurs, ce projet nous a permis de mettre en œuvre les outils mathématiques vus en cours.

Le programme résultant, s'il n'est pas très efficace sur un ordinateur (manque de rapidité), serait facilement adaptable à une utilisation sur circuits logiques. En effet, les algorithmes utilisés fonctionnent sur les bits des chiffres (base 256 = 2^8). Or les circuits logiques travaillent aussi sur les bits. De plus, les problèmes liés à la lenteur d'allocation de mémoire seraient réglés puisque l'on travaillerait alors sur une mémoire fixe donc équivalente à un tableau. On peut donc aisément penser à une application « de poche » du programme ; par exemple un encrypteur de la taille d'une calculatrice.