NF05 – INTRODUCTION AU LANGAGE C

IMPLEMENTATION DU CHIFFREMENT RSA EN LANGAGE C

Ivann LARUELLE - Adriana MASCIULLI

Semestre A2018

utt
UNIVERSITÉ DE TECHNOLOGIE
TROYES

Introduction

Dans le cadre du cours d'introduction au langage C, nous avons réalisé un projet qui consistait à programmer un petit outil. Entre l'agenda, le traitement d'image et le chiffrement RSA, nous avons retenu le chiffrement RSA, car cela nous intéressait plus en tant que futurs ingénieurs Réseaux & Télécoms, de plus celui-ci intégrait une dimension mathématique et pas seulement informatique. Nous avons également décidé de travailler à deux dans une optique d'échange des connaissances et des compétences, d'autant plus que nous sommes de futurs ingénieurs. Un ingénieur travaille rarement seul, encore moins quand il développe un logiciel.

Le principe de cryptographie RSA permet d'envoyer un message confidentiel. Il repose sur deux clés : une privée et une publique. Une de ces clés permet de déchiffrer un message crypté avec l'autre et inversement. Chaque personne possède une clé privée et une clé publique : la clé privée est connue de son propriétaire uniquement, tandis que sa clé publique est connue de tous. Si les clefs sont suffisamment longues, on ne peut pas retrouver dans un temps raisonnable la clef privée avec seulement la clef publique. On s'intéresse ici à un message envoyé par un expéditeur (Alice), crypté à l'aide de la clé publique du destinataire (Bob), qu'il connaît. Ainsi, seul Bob peut décrypter le message, à l'aide sa clé privée qu'il est le seul à connaître. Le message est protégé.
On cherche à créer un programme capable de chiffrer et déchiffrer un message à l'aide d'un couple de clés qu'il sera capable de générer.

Nous avons également souhaité avoir une plus-value pédagogique plus importante en essayant d'aller au-delà d'un simple programme qui fonctionne. Nous avons essayé de faire un programme qui fonctionne rapidement, qui limite l'occupation de la mémoire, facilement améliorable, accessible et qui prend en compte les erreurs que peut faire l'utilisateur dans l'utilisation du programme. Cela nous a poussé à sortir de notre zone de confort pour trouver des solutions et des méthodes adaptées à ces problèmes.

Table des matières

1. Principaux algorithmes

Vous trouverez ici la description du fonctionnement des fonctions les plus importantes pour le chiffrement RSA. La description de toutes les fonctions est accessible dans la documentation *Doxygen* (générée en Français avec un encodage en lecture Windows-1252).

Algorithme de Rabin-Miller

On génère $n = 2^s \times d + 1$, avec $30 \leq s\,(aléatoire) \leq 50\; et\; 10 \leq d\,(aléatoire) < 10^{10}\; avec\; d\; impair$, et on répète la génération tant que n pas impair ou divisible par 5. On génère a aléatoirement entre $10\; et \frac{n}{100}$

Si $a^d \,\%\, n = 1\; ou\; n - 1$, on dit que a passe le test.

Sinon, on fait *s* fois l'opération $reste = reste^2 \,\%\, n$, en regardant à chaque fois si reste est égal à n-1. Si on en trouve un seul, alors *a* passe le test.

Si on fait l'opération s fois sans trouver de reste égal à *n-1*, *a* ne passe pas le test et *n* ne peut pas être premier. On génère alors un nouveau *n*.

Si *a* passe le test, alors on régénère un nouveau a et on refait les opérations précédentes pour 4 *a* différents. Si un seul d'entre eux ne fonctionne pas, on régénère *n*.

Si on a 5 *a* différents qui passent le test de Rabin Miller, alors, *n* a une probabilité $1 - ¼^5 = 0.9990234375$ d'être premier.

Euclide étendu

L'algorithme d'Euclide étendu permet de déterminer le PGCD de deux entiers a et b ainsi qu'un couple de relatifs u et v tels que $au + bv = PGCD(a, b)$.

Pour $a > b$, on commence par faire la division euclidienne de a par b, on obtient un quotient q et un reste r. On obtient $1 \times a - qb = r$ (1)

u prend la valeur 1 et v la valeur -q.

On fait ensuite la division euclidienne de b par r, c'est-à-dire que le diviseur devient le dividende et le reste devient le diviseur.

On trouve un nouveau reste et un nouveau quotient tels que $1b - q'r = r'$ et d'après (1), on a $1b - q'(a - qb) = r'$. On a des nouvelles valeurs pour $u = -q'$ et $v = 1 + qq'$.

On répète ce schéma jusqu'à obtenir un reste de 0.

Le PGCD est le dernier reste différent de 0 obtenu. u et v sont sommées à chaque tour de boucle suivant les quotients et restes obtenus par division euclidienne.

Exponentiation modulaire rapide

L'exponentiation modulaire rapide repose sur le fait que $a \times b\; mod\; n = (a\; mod\; n) \times (b\; mod\; n)$

Si on écrit l'exposant **en binaire**, on a

$$message^{exposant} = \prod_{k=0}^{nbbits(exposant)-1} message^{2^k exposant.tableau[k]}$$

On a donc

$$résultat = \prod_{k=0}^{nbbits(exposant)-1} (message^{2^k exposant.tableau[k]} \ mod\ n)$$

On fait donc, pour chaque bit de e,

$$résultat = \left(résultat \times message^{(1 + le\ bit\ de\ e\ en\ cours)\ mod\ 2}\right) mod\ n$$

$$message = (message \times message)\ mod\ n$$

En pratique :

On pose résultat = 1.

Tant que l'exposant n'est pas nul :

- Si l'exposant n'est pas un multiple de 2 (c'est-à-dire son écriture binaire comporte 1 pour les unités), on a $résultat = (résultat \times message)\ mod\ n$
- Sinon on ne touche pas au résultat, car son écriture binaire comporte un 0 pour les unités (on multiplie par 1).
- Dans tous les cas, on fait $message = (message \times message)\ mod\ n$ et $exposant = \frac{exposant}{2}$ (division entière) pour passer au bit suivant de l'exposant.

Algorithme de chiffrement

On récupère le clef n et l'exposant a.

Le programme commence par lire le fichier source en convertissant un à un tous les caractères en ASCII en base 2 sur 8 bits et en les écrivant dans un autre fichier (fichbin.txt) sur une seule ligne sous forme binaire (un caractère = 8 * '0' et '1').

On s'assure que la taille de ce nouveau fichier soit un multiple de $\text{Ent}(\log_2(n))$ (équivalent à (nb de bits de n en base 2) - 1), sinon on complète avec des '0' à la fin. On écrit au début du fichier de destination le nombre de '0' ajoutés.

On lit le nouveau fichier par bloc de $\text{Ent}(\log_2(n))$caractères (car 1 caractère = '0' ou '1' dans ce nouveau fichier).

Pour chaque bloc, on récupère $bloc^a\ mod\ n$, c'est notre message chiffré. On complète ce nouveau chiffre par des 0 pour qu'il ait une taille $\text{Ent}(\log_2(n)) + 1$, la taille maximale que le reste peut avoir. On écrit ce nombre dans le fichier de destination.

On répète la lecture jusqu'à ce qu'il n'y ait plus de blocs à lire.

Algorithme de déchiffrement

On récupère la clef n et l'exposant a.

On lit dans le fichier chiffré le nombre de zéros ajoutés.

On lit le fichier chiffré par bloc de $\text{Ent}(\log_2(n))$ caractères, donc nb de bits(n) - 1.

Pour chaque bloc, on enlève les zéros devant, on récupère $bloc^a \ mod \ n$, c'est notre message déchiffré.

On complète ce nouveau chiffre par des zéros pour qu'il ait une taille $\text{Ent}(\log_2(n))$, la taille maximale que le reste peut avoir.

Si on est sur le dernier bloc, on enlève les zéros ajoutés à la fin s'il y en a.

On écrit ce nombre dans le fichier temporaire.

On répète la lecture jusqu'à ce qu'il n'y ait plus de blocs à lire.

Le programme recommence à lire le fichier temporaire en convertissant les '0' et les '1' par bloc de 8 en caractères en ASCII et en écrivant le caractère dans le fichier de destination.

2. Problèmes rencontrés et solutions adoptées

Finalement, au cours de ce projet, le plus difficile n'aura pas été de comprendre le fonctionnement des mathématiques derrière les algorithmes, mais de réussir à les coder de la manière la plus optimisée possible. En effet, il nous est arrivé plusieurs fois de créer rapidement un programme qui fonctionne mais qui à cause de l'utilisation de grands nombres était beaucoup trop lent.

A. Nos grands nombres

La première question était de savoir quelle représentation avoir pour nos grands nombres. Nous avons fait le choix de pourvoir modéliser tous les entiers possibles avec cette structure :

```
typedef struct
{
    int *tableau;
    int indicemax;
    int signe;
} GRDNB;
```

En effet, nous devions absolument savoir la taille de notre tableau d'entiers, d'où la présence d'un entier pour indiquer la taille.

Concernant le signe, nous souhaitions dépasser le cadre du projet et faire une bibliothèque de grands nombres très large, donc prenant en compte les nombres négatifs.

Nous avons hésité entre plusieurs représentations du signe :

- Mettre un 1 au début du tableau, solution qui nous paraissait complexe à gérer de prime abord, car peu intuitive. La priorité à l'époque était de pouvoir écrire rapidement des fonctions mathématiques qui marchent, ce qui s'est avéré être un mauvais choix comme

vous le verrez par la suite. Nous étions obligés de faire une bibliothèque de fonctions mathématiques très large (toutes les bases et signes possibles) car nous n'avions pas encore réfléchi à ces questions.

- Rajouter un entier dans la structure, solution très intuitive. Le signe du GRDNB est le signe de l'entier 'signe'. Le tableau représentait la valeur absolue de notre nombre, tandis que le signe était stocké à part et facilement accessible.

Toujours dans l'optique d'avoir une bibliothèque très large nous avons également étudié la possibilité de stocker la base dans laquelle le nombre est écrit, ce que nous n'avons pas fait car nous avons préféré éviter d'alourdir la définition d'un grand nombre. La base est devenue un argument à toutes les fonctions mathématiques, ce qui nous permet d'avoir d'un seul coup la base de tous les paramètres ainsi que celle du résultat à renvoyer.

B. La gestion de la mémoire

La gestion de la mémoire a été une vraie problématique d'envergure dans ce programme. Nous ne nous sommes pas tout de suite rendu compte de l'impact de la simple affectation d'un grand nombre. Le gaspillage occasionné par la mauvaise gestion de la mémoire rendait le programme très lent.

a. L'allocation dynamique

Nous avons fait le choix d'allouer les tableaux de tous nos nombres dynamiquement, et cela pour deux raisons :

- Cela permet de travailler avec tous les nombres possibles et inimaginables.
- Cela permet une gestion plus fine, mais plus lente de la mémoire. En effet, si l'on alloue statiquement un tableau de 100 entiers par exemple, l'ordinateur sait à l'avance quelle taille font mes grands nombres, donc cela va plus vite pour lui. En revanche si je crée 100 nombres de 1 chiffre, on se retrouve avec 10 000 entiers en mémoire, alors qu'avec l'allocation dynamique, on aurait seulement 100 entiers en mémoire (un entier par nombre).

Comme nous travaillons sur de grands nombres, l'allocation dynamique est un excellent compromis entre vitesse et occupation de la mémoire, mais elle n'est pas sans danger, car elle engage la responsabilité du programmeur, la mémoire étant allouée et gérée manuellement par celui-ci.

b. L'initialisation des valeurs allouées

Au départ nous utilisions malloc, cependant cette fonction n'efface pas le contenu aléatoire présent dans les emplacements alloués.

Cette situation n'était pas convenable, car les nombres devenaient aléatoires, et faire un traitement dessus pouvait éventuellement faire planter le programme. De la même façon, cela empêchait l'établissement de convention d'appel de fonction, vu que l'on ne savait pas ce que malloc pouvait nous envoyer.

Ensuite nous avons utilisé calloc(), qui initialisait tous les contenus en même temps de faire l'allocation du contenant.

Cette fonction a grandement amélioré la situation en amenant de la fiabilité à nos fonctions.

D'un point de vue conventionnel, nous pouvions affirmer que l'appel creerGRDNB(1) renvoyait la valeur +0. (allocation avec calloc d'une seule case de tableau, avec la valeur du signe étant positif), ce qui était pratique pour générer rapidement le nombre zéro.

c. L'affectation des GRDNB entre eux

Si l'on prend le code suivant

```
int main()
{
    int b = 5;
    int bcopie = b;
    b = 6;
    printf("%d",bcopie); //affichera 5
    return 0;
}
```

Tout va bien. La variable bcopie contient bien une copie de b. On peut donc modifier b sans affecter la copie, et vice versa. En revanche, celui va poser souci :

```
int main()
{
    GRDNB b = str2grdnb("2125451");
    GRDNB bcopie = b;
    free(b.tableau);
    afficher("",bcopie); //affichera !?
    return 0;
}
```

En effet, le programme a pourtant effectué une copie de b, mais de sa structure !

C'est l'adresse du tableau qui a été copiée, et non le tableau. Les tableaux de b et de bcopie sont en réalité à la même adresse, ce qui signifie que si je modifie b, je modifie également sa copie.

Cette erreur nous a coûté beaucoup de temps, car détectée assez tard.

Nous avons alors implémenté une fonction de copie, qui va réaliser une copie intégrale, case par case, du GRDNB, dans un nouveau tableau.

d. Les fuites de mémoire

Ce que nous ne savions pas, c'est que la mémoire écrasée n'est pas libérée. Exemple :

```
void fuite()
{
    int *test = NULL;
    for(int i = 0; i<25; i++)
        test = calloc(100,sizeof(int));
}
```

Nous pensions naïvement que à chaque tour de boucle, la mémoire précédente était libérée, mais il n'en est rien. Le programme fuite() ci-dessus prendra à lui tout seul la place de 2 500 entiers.
Cela était imperceptible au début, car nos fonctions mathématiques de base (multiplication, addition, …) renvoyaient le résultat sans pour autant que nous nous rendions compte de la mémoire occupée.
Là où le vrai problème s'est posé, c'est lors de la réaffectation de certains GRDNB dans nos fonctions.
En effet :

```
GRDNB puissance_naive(GRDNB a, int k, int base)
{
    b = creerGRDNB(1);
    b.tableau[0] = 1;
    for(int i = 0; i<k; i++)
        b = mulnaive(b,a,base);
    return b;
}
```

Ce genre de fonction, même si elle fonctionne, est une catastrophe.
Par exemple, à chaque tour de boucle, le tableau de b sera écrasé et non libéré car alloué manuellement.
Une solution serait la suivante :

```
GRDNB puissance_naive_fausse(GRDNB a, int k, int base)
{
    b = creerGRDNB(1);
    b.tableau[0] = 1;
    for(int i = 0; i<k; i++)
    {
        free(b.tableau);
        b = mulnaive(b,a,base);
    }
    return b;
}
```

On libère le tableau de b juste avant l'affectation, pour être certain qu'il n'y ait pas de données orphelines, mais l'on a besoin du tableau de b pour effectuer la multiplication de a par b. Cette fonction est donc fausse.

On peut alors copier b dans une variable temporaire, libérer b, affecter à b le résultat en remplaçant b par la variable temporaire dans le calcul, puis effacer la variable temporaire. Cette solution est lourde. Une piste très intéressante, qui a été implantée, fut celle des piles de pointeurs. On pouvait empiler des adresses et avec un seul appel libérer toutes les adresses de la pile et la pile elle-même.

```
typedef struct chaineDePointeurs chaineDePointeurs;
struct chaineDePointeurs
{
    int *pointeur; /**< Le contenu de l'élément de la liste, un pointeur sur
un entier */
    chaineDePointeurs *suivant; /**< L'adresse du prochain élément. */
};

typedef struct
{
    chaineDePointeurs *premier; /**< Adresse du premier élément de la pile. */
} pileDePointeurs;
```

En effet, nous avons fabriqué notre propre système de libération automatique de mémoire qui fonctionnait sur le principe suivant : chaque fonction se voyait attribuer un paramètre supplémentaire, celui de la pile de pointeurs externe. A l'intérieur de chaque fonction, on crée une pile de pointeur interne, et on appelle les autres fonctions en leur mettant comme paramètre la pile de pointeur interne. A la fin, on libère la pile interne et ses pointeurs correspondants puis on affecte le tableau du résultat du calcul renvoyé par la fonction dans la pile externe. Exemple sur un calcul fictif :

```
GRDNB calcul(GRDNB a, GRDNB b, pileDePointeurs *externe)
{
    pileDePointeurs interne;
    interne.premier = NULL;
    // On effectue des calculs, en leur donnant en paramètre la pile interne
    GRDNB test = calcul_autre(a, b, &interne); //l'adresse du tableau de test
est dans la pile interne
    GRDNB autretest = autre_fonction(a, b, &interne); //l'adresse du tableau
de autretest est dans la pile interne
    GRDNB calcul_test;
    for(int i=0;i<10;i++)
    {
        calcul_test = fonction_calcul(a,b, &interne); //Chaque résultat est
ajouté dans la pile interne.
    }
    GRDNB retour = somme(test,calcul_test);
    empiler(retour.tableau, externe); //On empile le résultat sur la pile
externe.
    depiler(&interne); //On dépile et on libére tout le contenu de la pile
interne.
    return retour;
}
```

Cette solution avait le mérite d'être claire et efficace, cependant sur un code déjà existant celle-ci est beaucoup trop lourde à déployer (il faut recoder toutes les fonctions pour qu'elles fonctionnent avec ce nouveau système, et il faut faire attention à ne pas libérer de tableaux indispensables pour le résultat). La recherche de bug aurait été beaucoup plus dure par ailleurs, parce que nos données sont enfouies dans les piles, donc si on libère un tableau qu'il ne fallait pas libérer, on se retrouve avec des heures de recherche. De plus, tant que la fonction n'est pas finie, la mémoire n'est pas libérée.

La solution est passée par la création d'une fonction affectation, partant du principe qu'un appel s'exécute toujours de droite à gauche.

```
void affectation(GRDNB *a, GRDNB b)
{
    free(a->tableau);
    *a=b;
}
```

L'ordinateur évalue b, puis a et exécute la fonction. On peut libérer a, même si le résultat dépend de a, car celui-ci a déjà été réalisé. L'adresse du tableau de b, ainsi que son signe et son indicemax sont copiés dans a. Exemple

```
GRDNB puissance_naive(GRDNB a, int k, int base)
{
    b = creerGRDNB(1);
    b.tableau[0] = 1;
    for(int i = 0; i<k; i++)
        affectation(&b,mulnaive(b,a,base));
    return b;
}
```

La seule chose à ne jamais faire est une affectation d'une variable dans une autre sans réfléchir. La fonction affectation ne copie pas case par case le contenu de b dans a. Modifier b même après l'affectation modifiera a.[1] La fonction affectation doit seulement être utilisée pour affecter le résultat d'une fonction dans une variable, le calcul ne pouvant pas être modifié après avoir été réalisé.

e. Les libérations successives

Si l'on fait deux free() de suite, l'ordinateur ne lèvera pas d'erreur.

Cela peut cependant être très dangereux, comme le montre l'exemple ci-dessous :

```
void trop_liberer()
{
    GRDNB test = str2grdnb("1214");
    free(test.tableau); //On libère test
```

[1] Voir « L'affectation de GRDNB entre eux »

```
    GRDNB tmp = str2grdnb("2124554"); //Tiens, je viens de libérer une
adresse, autant la réutiliser tout de suite pour ce nombre !
    free(test.tableau); //On libère encore test, sauf qu'on libère aussi tmp
au passage...
}
```

La solution pour cela est simple :

- Soit on fait attention à ne pas libérer deux fois la même chose,
- Soit on met le pointeur à NULL juste après l'avoir libéré.

La solution retenue pour les tests était l'implémentation de la fonction suivante, pratique et efficace :

```
void liberer(GRDNB *a)
{
    free(a->tableau);
    a->tableau = NULL;
}
```

Si on appelle deux fois de suite liberer(), cela ne posera pas de problème car le pointeur est à NULL la deuxième fois, c'est-à-dire que l'ordinateur va faire free(NULL), ce qui d'après le manuel de free, ne fait rien du tout.[2]

En réalité nous n'avons pas eu le temps de remplacer tous nos free par cette fonction, qui nous a servi que pendant nos tests. Nous avons préféré garder des fonctions qui marchent plutôt que tout chambouler avec une solution de dernière minute.

Ce projet nous a permis d'apprendre beaucoup de choses sur la gestion manuelle de la mémoire en C.

C. La vitesse d'exécution

Nous devions réaliser les calculs le plus rapidement possible.

A la base, nous avions tout coder pour pouvoir changer rapidement de base si nécessaire, et cela s'est avéré utile ! Il a fallu trouver un compromis là encore.

a. Le choix de la base

En effet, un nombre écrit par exemple en base 64 prendra toujours moins de place qu'un nombre écrit dans la base 10. En réalité, nos nombres sont des tableaux d'entiers et un entier peut contenir des

[2] https://stackoverflow.com/questions/6084218/is-it-good-practice-to-free-a-null-pointer-in-c

valeurs très grandes, et pas seulement des chiffres. La différence est l'augmentation du temps de calcul.

Par exemple, il est plus simple d'utiliser la base 2 pour lire les clés et les données à chiffrer en ASCII mais on lui préférait la base 10 pour les calculs mathématiques, plus intuitive. Nous avons donc presque écrit tout notre programme pour qu'il fonctionne avec les deux bases et finalement utilisé la base 2, plus couteuse en espace, mais bien plus rapide en calcul : par exemple une somme peut se faire en utilisant simplement des fonctions logiques (ET, OU, OUX, …).

b. L'implémentation des algorithmes

Notamment avec l'algorithme de la division euclidienne, il nous a fallu trouver les solutions du problème d'une autre manière, avec un autre point de vue. Nous sommes ainsi passé d'une simple incrémentation de 1 à un algorithme bien plus performant : là où nous comptions combien de fois on pouvait soustraire b dans a, ce qui prenait un temps phénoménal, nous avons implémenté une autre solution que nous avons pensé en cours de route : compter, pour chaque puissance du quotient, combien de fois celle-ci pouvait rentrer dans a, ce qui réduisait, par rapport à la division naïve, la complexité au pire de $quotient$ à $nombreChiffres(quotient) \times (base - 1)$. En base 2, nous étions même à $nombreChiffres(quotient) \times (1)$.

Plus globalement, voilà ce que nous avons appris :

- Il faut toujours veiller à ne pas faire deux calculs identiques : dans Rabin Miller par exemple, pour chaque n testé, nous calculions deux fois la valeur de n-1. C'était autant de temps et de mémoire perdue. L'algorithme a accéléré après avoir calculé et stocké une seule fois n – 1.
- Il faut veiller aussi à éviter les calculs inutiles : lors du chiffrement et du déchiffrement, nous avions un calcul récurrent dont le résultat était exprimé en base 2. Nous passions ensuite dans nos calculs mathématiques en base 10, et nous retransformions ce nombre en base 2 pour l'écrire dans un fichier. Quand nous avons passé tous nos calculs en base 2, ces opérations ont été réellement accélérées.
- On doit essayer de se ramener à des opérations dont le résultat est connu d'avance, ou rapide à calculer. Dans plusieurs cas, il a été inutile de passer par la division euclidienne. Par exemple, diviser ou multiplier un nombre par sa base revient à supprimer ou ajouter le dernier chiffre (celui des unités) : on réalloue directement le tableau. Ou encore, nous avons trouvé une logique mathématique pour réaliser très facilement la division par 2 en base 10.[3] Le fait de ne pas avoir eu besoin de passer par la division euclidienne nous a fait gagner pas mal de temps. Sommer sur notre nombre des puissances de la base revient à incrémenter sa case correspondante, inutile de calculer la puissance et la somme si notre nombre est plus grand

[3] Voir la documentation doxygen pour div2 dans le module grdnb.c

que la puissance concernée, et que l'incrémentation ne permet pas à la case de dépasser la valeur de la base, ce qui est valide dans le calcul du quotient de la division euclidienne par exemple.

- Il faut faire attention aux fonctions récursives, elles ont tendance à prendre beaucoup d'espace en mémoire si on les code sans trop réfléchir à la mémoire utilisée. Il faut essayer de se ramener à une fonction itérative, sur laquelle on voit mieux les opérations effectuées (et donc on peut optimiser plus rapidement la mémoire utilisée et la vitesse).
- Il faut se documenter. Les travaux de recherche, ainsi que les forums spécialisés nous ont aidé à trouver des solutions à nos problèmes mathématiques. Des chercheurs, des ingénieurs ou d'autres étudiants avaient des problèmes légèrement différents, et en tant que futurs ingénieurs nous devons être capable d'adapter une solution déjà existante à notre problème et l'améliorer si possible.

D. Faciliter la résolution de bug et l'amélioration du logiciel

Il a fallu trouver des solutions pour pouvoir rapidement débloquer nos programmes. On passe parfois beaucoup de temps à chercher une erreur et à tout remettre dans l'ordre une fois que l'on a fini. C'est une problématique importante dans la gestion d'un projet informatique, car le temps que l'on dépense à résoudre un bug n'est pas du temps passé à optimiser les algorithmes, c'est une perte de temps. Nous avons développé plusieurs solutions pour accélérer la recherche de bug.

a. Affichage et mode de débogage

Lors du développement ou de l'amélioration d'une fonction, il est très utile de mettre des fonctions d'affichage un peu partout pour comprendre le fonctionnement de la fonction. Quand la fonction marche, on enlève les affichages, mais quand on cherche à optimiser la fonction, ou que l'on développe une autre fonction qui dépend de celle-ci, il faut réécrire tous les affichages, c'est du temps perdu.

A la place, la solution efficace que nous avons imaginée, c'est de faire une variable globale DEBUG et conditionner les affichages selon la valeur de DEBUG. Selon que l'on teste plusieurs parties du programme, on a juste à changer la variable DEBUG pour autoriser l'affichage dans un groupe de fonctions dont l'affichage est conditionné à cette valeur. (Si je veux tester toutes les fonctions mathématiques, je mets DEBUG à 2, les fonctions de traitement de fichier à 3, … Cela s'adapte selon les scénarios de notre choix, on peut mixer les deux). Par exemple :

```
void fonction()
{
    int debugEnCours = (DEBUG == 5 || DEBUG == 6);
    //...
    if(debugEnCours)
```

```
        printf("%d",valeur);
    //...
    if(debugEnCours)
        printf("%s",chaine);
}
```

Les informations de débogage ne s'afficheront que si DEBUG vaut 5 ou 6. Il suffit de modifier la condition au début de la fonction pour générer un affichage selon plusieurs combinaisons.
Cela a été supprimé une fois que toutes nos fonctions fonctionnaient ensemble pour ne pas alourdir la lecture du code.

b. Utilisation du débogueur

Une fois que nous avions appris à utiliser une partie du débogueur, la solution précédente a été petit à petit abandonnée. En effet, le débogueur nous permet d'avoir les mêmes résultats sans alourdir le code.
Le débogueur à travers ses nombreuses interfaces pouvait nous dire exactement à quel appel de fonction, à quelle ligne le logiciel avait planté, quelles étaient les valeurs lors du crash, via le *call stack* (listes et ordre des fonctions appelées), le *memory dump* (permet de voir l'état de la mémoire) et les *watches* (permet de voir le contenu des variables). Cela nous a permis de gagner un temps considérable.

c. Tests « uniques »

Lors du développement, nous avons codé nos premières fonctions mathématiques tête baissée, les unes dépendant parfois des autres. Lors du premier test, rien ne fonctionnait, et cela a tout compliqué pour trouver les erreurs. Nous souhaitions également réaliser des tests de performance, tant sur la mémoire utilisée que sur la vitesse d'exécution. Il a donc fallu trouver une méthode la plus rigoureuse de test de nos fonctions, et nous avons essayé de faire la suivante :
On commence par effacer le contenu du main(), ou on recrée un nouveau projet.

- On initialise nos valeurs à tester.
- On exécute nos fonctions (plusieurs fois si c'est un test de performance)
- On met un getchar() à la fin pour mettre le programme en pause car dès que le programme se termine, CodeBlocks affiche la vitesse, mais libère également la mémoire, ce qui ne nous permet pas de constater la mémoire occupée par le test effectué.
- On compare les résultats avec les résultats attendus. (temps d'exécution indiqué en bas de l'écran, et mémoire indiquée dans le gestionnaire des tâches avant d'appuyer sur une touche, d'où le getchar).

Il faut bien veiller entre chaque test à ne faire varier qu'un seul paramètre.

Dans le cas d'un test simple d'une fonction, on connait le résultat à l'avance et on regarde ce que la fonction nous retourne. On se fait accompagner éventuellement du débogueur pour voir les détails.
Dans le cadre d'un test de performance, on fait de nombreuses itérations successives, avec des valeurs différentes à chaque fois, et on regarde le temps mis et l'espace occupé. On exécute ce test pour deux fonctions différentes censées renvoyer le même résultat, et on compare leur vitesse d'exécution ainsi que la mémoire occupée.

d. Programmation modulaire

Nous avons dès le début décidé de faire une programmation avec le plus de fonctions possibles, où les fonctions mathématiques basiques des GRDNB étaient dans un module bien précis (grdnb.c).
De même, les fonctions strictement liées au chiffrement RSA ont été placées dans un autre module, tout comme les fonctions de saisies sécurisées.
Cela a des avantages :

- Le code est plus clair. On sait tout de suite où peut se trouver une fonction selon ce qu'elle fait et les objets sur lesquels elle intervient. C'est un gain de temps dans le développement.
- On peut travailler en groupe : chacun pouvait modifier un fichier sans empêcher l'autre de modifier des fonctions dans un autre fichier. Il n'y avait pas de conflits de synchronisation.
- Plus les fonctions peuvent traiter des cas larges, mieux c'est. Le (dé)chiffrement par exemple, consiste à l'origine à récupérer la clef privée pour le déchiffrement et publique pour le chiffrement, et (dé)chiffrer le fichier. Nous avons décidé de séparer dès le début ce travail en plusieurs fonctions, ce qui faisait que l'on pouvait chiffrer avec n'importe quel exposant. Nous avons donc pu par la suite implémenter la signature en deux clics, en envoyant la clef privée à la fonction de chiffrement plutôt que la clef publique.

La programmation modulaire et avec beaucoup de fonctions permet de faire évoluer son code très rapidement, et surtout de pouvoir innover sans limitations, car les fonctions peuvent traiter des cas larges. Il faut juste modifier les paramètres qu'on leur transmet.

E. Le travail de groupe

Développer en groupe fut intéressant, et une tâche plus ardue que prévu.
En effet, il a fallu trouver un support commun pour le partage du code.
Nous avons décidé de prendre une solution *drive* classique, ce qui s'est avéré ne pas être la meilleure idée. Après quelques erreurs qui nous ont couté du temps, il a fallu s'imposer des règles strictes : ne travailler qu'avec des copies du contenu du *drive* et écraser le contenu de celui-ci une fois seulement que la copie fonctionnait correctement. Il ne fallait jamais travailler directement dans le drive : le vrai problème c'est la gestion des versions, il n'y a pas d'historique des fichiers.

C'est une vraie problématique dans le développement en groupe, car on perd les avantages cités au-dessus : on pouvait difficilement travailler à deux en même temps, car les modifications de l'un effaçaient les modifications de l'autre (vu que l'on copiait ou écrasait tout l'espace de travail). On se retrouvait parfois avec des dizaines de copies pour avoir tout l'historique des fichiers. Le problème n'a pas été trop visible car nous étions proches géographiquement, donc nous pouvions régulièrement nous voir.

Cela nous a appris que travailler à plusieurs sur un projet informatique sans *drive* spécialisé (gestion des versions, des accès, de l'historique des fichiers…) et sans règles communes pouvait relever du calvaire si l'on est tous éloignés géographiquement ou que nous sommes nombreux à travailler dessus en même temps.

F. Faire un programme accessible et avec des saisies sécurisées

En tant que futurs ingénieurs, nous nous devons de réaliser un travail qui fonctionne, mais pas seulement. Il faut aussi qu'il soit accessible aux individus n'ayant pas forcément le même niveau de connaissance en informatique. Cela se fait en se mettant à la place de l'usager, par des interfaces claires et la récupération d'erreurs.
Nous avons par exemple fait le choix d'une interface épurée, facile d'utilisation et dotée d'explications à chaque étape du processus.
Ce qui a nécessité le plus de temps, c'est de prévoir tous les cas possibles de mauvaise utilisation du programme pour éviter qu'il ne plante (incompatibilité de type des données rentrées par l'utilisateur par exemple).

Il faut donc concevoir des fonctions de saisie sécurisée.
La fonction saisieBrute() le permet : on lit caractère par caractère la saisie de l'utilisateur, on la stocke dans un fichier temporaire, et on compte le nombre de caractères saisis ainsi. Il suffit de lire la ligne du fichier en ayant alloué la bonne taille pour la chaine de retour.
La saisie est sécurisée : aucun dépassement de mémoire possible alors que l'on ne sait pas à l'avance la taille ni le contenu de ce que l'utilisateur va rentrer.
Les fonctions saisieInt et saisieGRDNB fonctionnent grâce à saisieBrute, et ces fonctions vérifient la chaine pour s'assurer qu'elle peut bien être convertie dans le type de donnée correspondant.
Les fonctions de menus vérifient chaque retour pour s'assurer que la variable retournée correspond bien aux propositions faites à l'utilisateur.
Il est donc impossible à l'utilisateur de faire planter le programme avec une saisie.

Nous ne nous sommes pas arrêtés là : nous avons sécurisé le parcours de fichier.

Outre la traditionnelle vérification de la bonne ouverture de fichier, nous avons vérifié systématiquement que les données dans les fichiers correspondent à ce que nos fonctions peuvent traiter. Par exemple, nous ne pouvons lire dans les fichiers contenant les clefs que des formats bien précis. Nous nous assurons que tout est respecté, même jusqu'à la base dans laquelle sont écrites les clefs, et si un seul problème est détecté, nous affichons un message d'erreur clair à l'utilisateur, au lieu de le laisser lancer un calcul qui va planter.
Les seuls cas que nous n'arrivons pas à prédire sont le déchiffrement avec une mauvaise clef, et si le fichier à déchiffrer a été modifié avant d'être déchiffré.
Une solution pour le deuxième problème aurait été l'ajout d'un hash de la chaine chiffrée à la fin du fichier contenant la chaine chiffrée. Lors de la lecture, il aurait juste fallu vérifier si le hash correspond à la chaine de caractère précédente pour voir tout de suite si le fichier à déchiffrer a été modifié et arrêter tout de suite le déchiffrement. Nous n'avons pas eu le temps de développer cette solution, malheureusement.
Nous aurions également pu adapter Rabin Miller pour vérifier que les p et q rentrés manuellement soient premiers.

3. Mode d'emploi du programme

- Affichage du menu principal : on a le choix entre générer des clés, envoyer ou recevoir un message. Pour commencer, on choisit de générer les clés.
- La génération des clés nécessite deux entiers premiers positifs. L'utilisateur a alors le choix : rentrer lui-même ces deux entiers ou utiliser l'algorithme de Rabin-Miller, qui les génère automatiquement. Cette génération est aléatoire et peut donc prendre un peu de temps. Dans le cas de la génération manuelle, il peut choisir entre les bases 2 et 10.
- Stockage des clés dans des fichiers dont on choisit les noms (ex : clepublique et cleprivee, l'extension est ajoutée automatiquement et les clefs sont enregistrées dans les dossiers public_keys\ ou private_keys\ selon leur type).
- On revient au menu principal. On va maintenant choisir d'envoyer un message. On a le choix entre
    - Le chiffrement simple d'un document avec la clef publique du destinataire,
    - La signature d'un document avec notre clef privée,
    - La signature avec notre clef privée puis le chiffrement avec la clef publique du destinataire.
- Il faut écrire un fichier texte avec le message et le sauver sur l'ordinateur. Ensuite, entrer le nom de ce fichier avec son extension éventuelle ainsi que le nom du fichier qui va contenir le message crypté et aussi choisir la clé. Le chiffrage se fait.

- On revient au menu principal et on se met en mode réception. On choisit le déchiffrement. Il faut entrer le nom du fichier crypté (que l'on a choisi juste avant) et choisir un nom pour le message décrypté ainsi que prendre la bonne clé correspondante. Le déchiffrement se fait.
- On doit retrouver le message du début dans le fichier décrypté.

Vous devrez respecter les conditions suivantes :

- Les clefs doivent être enregistrées dans un fichier portant l'extension .key sous le format suivant : « n exposant » sur la même ligne, les deux écrits en base 2. Si c'est une clef privée, vous devez l'importer dans le dossier correspondant :
    - private_keys\ pour les clefs de la forme n d
    - public_keys\ pour les clefs de la forme n e
- Les clefs importées doivent être d'au moins 9 bits.
- Les caractères écrits dans les fichiers à chiffrer doivent être lisibles sur 8 bits.

Attention, sans le respect des conditions suivantes, le programme aura un comportement indéterminé :

- Vous devez être le destinataire légitime du fichier à déchiffrer car vous devez posséder les clefs correspondantes, et le fichier à déchiffrer ne doit pas avoir été modifié. Utiliser une mauvaise clef pour déchiffrer ou vérifier une signature peut mener à un comportement indéterminé.
- Vous ne devez pas modifier les valeurs binaires des clefs présentes dans les fichiers.
- p et q saisis doivent être premiers

Conclusion

Ce projet aura été une expérience vraiment enrichissante. D'une part par sa complexité, mais aussi parce que le chiffrage RSA est toujours utilisé aujourd'hui et son algorithme est inviolable si les clefs générées sont suffisamment grandes. Le projet nous a poussé à réfléchir sur les calculs réalisés pour aboutir au résultat chiffré. C'est donc un projet particulièrement adapté dans le cadre d'études d'ingénieur.

Notre avons choisi d'améliorer au mieux notre programme pour améliorer la plus-value pédagogique, en y intégrant les problématiques de rapidité, de fiabilité, d'espace mémoire et de facilité d'adaptation du code. Nous avons beaucoup appris de nos erreurs, et notamment développé de nouvelles méthodes et pratiques pour répondre à ces problématiques. Le programme pourra tout de même être optimisé. Une possible amélioration serait de réussir à libérer davantage d'espace dans nos différentes fonctions pour gagner de la mémoire et permettre au programme de s'exécuter plus vite. Même si nous n'avons pas réussi à optimiser au maximum le programme, les méthodes développées nous ont permis de pouvoir innover par rapport au sujet proposé.

Nous avons par exemple programmé la signature couplée au chiffrement. Nous n'avons dans le chiffrement traditionnel que la clef publique du destinataire. On génère un autre couple de clés, appartenant à l'expéditeur cette fois. Il pourra alors crypter deux fois le message, avec sa clef privée puis avec la clef publique du destinataire. Le destinataire décodera alors le message avec sa propre clef privée puis avec la clé publique de l'expéditeur supposé. Dans cette situation on a alors une double sécurité : la garantie sur l'identité de l'expéditeur et la garantie que seule la personne à qui est adressé le message sera en mesure de l'ouvrir.

Annexes

Voici une liste non exhaustive des ressources qui nous ont aidé durant ce projet :

[1]. Le forum StackOverflow regorge de développeurs expérimentés. On y trouve rapidement la solution à des erreurs parfois floues. On y apprend par ailleurs beaucoup de choses, car certaines personnes prennent le temps de détailler leurs réponses.

[2]. La partie « Teaching » de cet enseignant chercheur du Centre de Recherche Informatique de Lens fut très intéressante : http://www.cril.univ-artois.fr/~benferhat/

[3]. Le rapport d'un chercheur au CNRS à Polytechnique, sur les techniques de multiplication : http://www.lix.polytechnique.fr/~pilaud/enseignement/cours_premiere/multiplication.pdf

[4]. Algorithme de Karatsuba pour la multiplication rapide dont l'implémentation fut abandonnée : https://fr.wikipedia.org/wiki/Algorithme_de_Karatsuba

[5]. Nous aurions pu implémenter une application du Théorème des Restes Chinois au RSA : https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_des_restes_chinois

[6]. Ce site associatif, bien qu'amateur, a traité de long en large la division euclidienne classique : http://compoasso.free.fr/primelistweb/page/prime/euclide.php

[7]. Le site d'OpenClassrooms et sa partie dédiée au RSA nous a été utile pour comprendre quelques subtilités : https://openclassrooms.com/fr/courses/477751-lalgorithme-rsa/477335-chiffrer-et-dechiffrer