Université de Limoges

Développement Logiciels Cryptographiques

RSA à jeu reduit d'instructions

Victor Delfour - Thomas Lavaur - Théo Bonnet

encadré par
Mr. Christophe Clavier

Master 2 cryptis - 2020/2021

Table des matières

1 Les fonctions

1.1 Fonctions d'arithmétique

✈ *mod*, *modulo* et *modulo_ui* :

Ces fonctions permettent de réaliser le modulo de différents types, de différentes façons. La première (*mod*) prend en entrée deux *unsigned int* et retourne le modulo en calculant le reste de la division euclidienne. La fonction *modulo* fait la même chose mais sur des *mpz_t* avec un argument d'entrée supplémentaire qui est le *mpz_t* de retour. Nous avons calqué les fonctions sur les *mpz_t* à celles de gmp à savoir de la forme void fonction(mpz_t sortie, mpz_t entrée...). Enfin *modulo_ui* réalise la même chose que *modulo* mais avec un module *unsigned int*.

✈ *ui_expo_ui*, *exp_mod* et *exp_mod_ui* :

Ces fonctions réalisent l'exponentiation, modulaire ou non. La fonction *ui_expo_ui* est une fonction qui réalise l'exponentiation rapide de deux *unsigned int* et stocke le résultat dans un *mpz_t*. On réalise le même procédé dans la fonction *exp_mod* mais sur des *mpz_t* en entrée et avec l'exponentiation rapide modulaire. Enfin, la fonction *exp_mod_ui* correspond exactement à la fonction d'exponentiation de gmp.

✈ La fonction *euclid_gcd* :

Cette fonction est l'algorithme d'Euclide étendu. En entrées elle prend, a et b les deux nombres sur lesquels on applique l'algorithme ainsi que x, y et *pgcd* qui sont les sorties de l'algorithme.

✈ *modular_inv* :

Cette fonction prend trois entrées a, b et t et stocke dans t la valeur $b^{-1}\ mod\ a$. Pour cela on utilise juste la fonction *euclid_gcd* qui nous renvoie ainsi les coefficients de Bezout qui permettent de trouver l'inverse modulaire.

✈ *taille_256* :

Cette fonction prend en entrée $n = p \cdot q$ et renvoie en sortie le plus grand entier i tel que $n > 256^i$

1.2 Fonctions pour la génération de clé

✈ *Miller_Rabin* :

Cette fonction est celle décrite dans le cours. Les notations que nous avons utilisé sont les mêmes que celles utilisées dans celui-ci hormis *nombre* qui joue le rôle de n.

✈ *optimized_crible_generation* :

Dans cette fonction nous utilisons le crible optimisé afin de générer un nombre premier. Pour cela, nous avons besoin d'un certain nombre de petits premiers choisi par l'utilisateur. Afin de pallier ce problème, nous avons donc un fichier texte annexe qui contient les 1000 plus petits premiers. Nous avons fait le choix de 1000 car cela semblait suffisant sans prendre trop de place mémoire. Les nombres premiers sont ainsi stockés dans un tableau et le reste de l'algorithme est semblable à celui décrit dans le cours.

✈ *generation_cle* :

Cette fonction permet de créer des couples de clés cryptographiques de taille choisie. Si la longueur désirée d est paire alors on calcule deux nombres premiers p et q de taille $\frac{d}{2}$ jusqu'à ce que le produit $p \cdot q$ donne la bonne longueur. Sinon, si d est impaire on prend alors p de taille $\lfloor \frac{d}{2} \rfloor$ et q de taille $\lceil \frac{d}{2} \rceil$, jusqu'à ce que le produit soit de nouveau de longueur d. On écrit alors, dans un second temps, les clés dans des fichiers binaires, celui contenant la clé publique ne contient que cela tandis que celui contenant la clé privée contient la clé privée ainsi que les autres secrets nécessaires au déchiffrement CRT.

1.3 Fonctions de padding

✈ $sha256sum$:

Cette fonction récupère le hachage par SHA256 d'un string à partir d'une commande openSSL.

✈ *int_to_hex* :

Cette fonction transforme un entier compris entre 0 et 15 en caractère hexadécimal.

✈ *hex_to_int* :

Cette fonction transforme un caractère hexadécimal en un entier correspondant à sa valeur entre 0 et 15.

✈ XOR :

Cette fonction calcule le XOR entre deux caractères écris en hexadécimal et renvoie le résultat sous forme d'un caractère en hexadécimal.

✈ $I2OSP$:

Cette fonction est décrite dans PKCS #1 v2.1 et est une sous-fonction de $MGF1$.

✈ $MGF1$:

Cette fonction crée un masque de la longueur en octets voulue à partir d'une graine et à l'aide de SHA256. Elle est déterministe et décrite dans PKCS #1 v2.1.

✈ $OAEP$:

Cette fonction crée les blocs par le padding OAEP décrit en section 3.2. Les blocs sont stockés dans un fichier. Ce fichier sera ensuite lu dans la fonction *encrypt* afin de chiffrer les blocs à l'aide de n et e.

➜ *inv_OAEP* :

Cette fonction récupère les messages à partir d'un bloc par le padding OAEP décrit en section 3.2. Les blocs sont stockés dans un fichier. Ce fichier sera ensuite lu dans la fonction *decrypt* afin de déchiffrer les blocs à l'aide de n et d.

1.4 Fonctions de chiffrement et de déchiffrement

➜ *encrypt* :

L'entrée de *encrypt* est le booléen *signature*. Si *signature* vaut 0 on chiffre, si *signature* vaut 1, on signe. Il n'y a pas de sortie. Cette fonction demande à l'utilisateur le nom du fichier contenant la clé publique et vérifie son existence. De même pour le fichier à traiter (chiffrer ou signer). Dans le cas où l'un de ces fichiers n'existe pas, on redemande le nom du fichier. Ensuite, l'utilisateur choisit le nom du fichier dans lequel sera stocké la signature ou le chiffré.

Si l'utilisateur désire chiffrer, alors la fonction récupère e et $n = p \cdot q$ dans le fichier contenant la clé publique. Ensuite, afin de pouvoir chiffrer dans de bonnes conditions et notamment pouvoir faire le padding, on crée un générateur aléatoire et on calcule la taille maximale en octets que l'on pourra mettre dans un bloc en comptant les blocs de padding. On récupère la taille du fichier puis on entame le chiffrement en lui-même. On regarde si la taille du fichier est inférieure à celle d'un bloc de donnée, auquel cas on adapte la variable *dernier* puis on appelle la fonction *OAEP*. Lorsque le bloc est fini ou que l'on arrive à la fin du fichier, on fait l'exponentiation $m^e \ mod \ n$ et on écrit ce nombre dans le fichier contenant le chiffré. Ensuite, on regarde si le prochain bloc est le dernier bloc, dans ce cas on adapte la taille du padding. Ensuite, on recommence jusqu'à avoir chiffré tout le fichier.

Si l'utilisateur veut signer, alors on effectue la même opération en utilisant d à la place de e.

➜ *decrypt* :

L'entrée de *decrypt* prend deux booléens : *signature* qui vaut 1 si on déchiffre avec la clé publique pour vérifier la signature ou 0 si on déchiffre de manière classique avec la clé privée. Dans ce second cas, on peut choisir si le second booléen *crt* vaut 0 ou 1. S'il vaut 0 on déchiffre directement, s'il vaut 1 on utilise le mode CRT. Il n'y a pas de sortie dans celle-ci non plus.

La fonction récupère, dans un premier temps, les éléments nécessaire à son fonctionnement en demandant à l'utilisateur de lui fournir les fichiers contenant le chiffrer, la clé publique et, s'il s'agit de déchiffrement classique, la clé privée. Dans un second temps, la fonction calcule la longueur du fichier afin de savoir jusqu'où déchiffrer. Pour le déchiffrement, on récupère le dernier bloc non lu qui correspond à un *mpz_t*, on l'élève à la puissance correspondant au mode choisi. Il faut par la suite supprimer le padding qui consiste simplement à appeler la fonction

inv_OAEP. Enfin, on récupère, octet par octet et avec divisions successives, le message clair d'origine et on l'écrit dans un nouveau fichier.

- ✈ $verification\_signature$:

$verification\_signature$ ne prend aucune entrée et ne renvoie rien (ce qui est une décision qui est facilement modifiable et qui pourrait renvoyer un booléen pour une utilisation intérieure au programme). Cette fonction affiche simplement à l'écran si une signature est valide ou non pour une clé publique donnée.

Pour cela, on hashe le déchiffrer en mode signature grâce à la fonction *decrypt* et SHA_256. Une fois cela réalisé, on obtient un fichier au nom fixe de « $signature\_a\_verifier$ ». On demande alors à l'utilisateur de quel fichier il veut vérifier la signature puis on compare octet par octet avec le haché de celui-ci s'il correspond au fichier précédant. Si tout correspond, alors la signature est valide.

2 Le programme

Le programme en lui-même ne fait qu'organiser les différentes fonctions principales qui sont souvent sans retour et sont donc indépendantes. Nous avons pris la décision de faire tenir tout notre programme dans un seul code. Notre fonction *main* est en fait un simple menu et permet juste d'initialiser le générateur aléatoire de *gmp* ou à l'utilisateur de naviguer entre les différentes options.

3 Problèmes rencontrés et choix d'implémentation

3.1 Problèmes rencontrés

Lors de la création de notre programme, il a fallu faire un choix de padding. En effet, l'absence de padding introduit deux propriétés néfastes à la sécurité. Étant donné que RSA est déterministe, la fonction n'est pas sémantiquement sûre. Si le dernier bloc contient un faible nombre de caractères, il est possible par brute force de déterminer ces caractères en essayant d'encoder divers caractères. De plus, l'absence de padding rend le décodage du dernier bloc différent de celui des autres et peut entraîner une ou plusieurs erreurs.

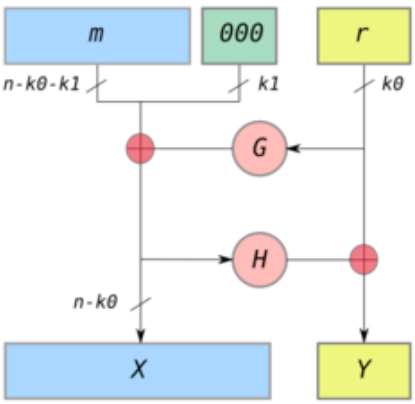
Un autre problème qui nous a pris un peu de temps, c'est l'encodage des données. Notamment, lorsque nous avons déchiffré un sous-message. Le problème était de passer les données d'une valeur écrite en décimal à une valeur écrite en hexadécimal dans un fichier.

3.2 Choix d'implémentation

Choix du padding :
Nous avions, dans un premier temps, choisi pour padding un dérivé de celui présenté dans PKCS#1 v1.5. Les blocs se présentent sous la forme suivante : $00||10||P||00||D$ où P est le padding. P est constitué de 8 octets non nuls générés aléatoirement sur chaque bloc sauf le dernier, où il est de 8 octets ou plus. Le second 00 sert de séparateur entre les données D et le padding. Il est possible alors d'encoder au plus $taille\_256(n) - 11$ octets de donnée par bloc. Cela impose alors une taille de n minimale. Il n'est pas conseillé d'utiliser le programme avec un n de longueur inférieure à 150 bits, une longueur d'au moins 1024 est recommandée. Le programme avec ce padding est fourni sous le nom : « padding_1_5 »

Une fois ce premier padding réalisé, nous avons choisi d'implémenter le padding RSA-OAEP. A cette fin, nous avons d'abord dû implémenter la fonction $MGF1$: la fonction de masquage préconisée par PKCS #1 v2.1. Le padding fonctionne de la manière suivante : on génère aléatoirement 8 octets r puis on crée un masque de $k - 10$ octets par $MGF1$ avec pour graine r. On XOR ensuite ce masque avec $m||00$. Le résultat de ce XOR est X. On passe ensuite les 8 premiers octets de X dans $MGF1$ pour créer un masque de 8 octets que l'on XOR avec r afin d'obtenir Y. Le bloc envoyé est alors $(X||Y)^e \ mod \ n$.
Une fois le bloc déchiffré, pour enlever le padding on prend les 8 premiers octets du bloc puis on les passe dans $MGF1$ avant de XOR le résultat avec Y pour récupérer r. Une fois r récupéré, on utilise $MGF1$ sur r afin de récupérer le masque obtenu lors du chiffrement et par un XOR on récupère m.

Dans le cas du dernier bloc, le padding se passe différemment : r n'est pas aléatoire mais correspond au nombre d'octets de m. De plus, m est paddé par des 00 jusqu'à attendre la taille $k - 9$.



Dans ce schéma d'OAEP, G et H correspondent à $MGF1$ et n correspond à k le nombre maximal d'octets contenus dans un bloc en fonction de $n = p \cdot q$.

Nous avons choisi d'implémenter des fonctions quasiment indépendantes qui stockent les informations dans des fichiers. La majorité de nos fonctions n'ont pas de structure, comme les fonctions de GMP qui ont des sorties vides.

En ce qui concerne les choix algorithmiques, nous avons choisi d'implémenter le crible optimisé pour la génération des nombres premiers avec Miller-Rabin comme test de primalité.

Dans ces algorithmes nous avons également dû faire des choix de précisions, par exemple pour le crible optimisé nous avons décidé d'utiliser les 1000 premiers nombres premiers et d'utiliser 10 comme paramètre de sécurité pour Miller-Rabin.

4 Les améliorations introduites

Tout d'abord une des toutes premières versions de notre programme effectuait un chiffrement RSA seulement sur les fichiers txt. Par la suite, nous l'avons amélioré pour qu'il puisse chiffrer n'importe quel type de fichier comme par exemple des images.

En lien avec la première amélioration, notre programme de base écrivait un chiffrer bien plus gros car pour écrire un bloc dans le fichier, nous écrivions directement caractère par caractère ce nombre. Mais maintenant, nous écrivons un bloc directement comme fichier binaire avec la fonction de gmp pour stocker des mpz_t.

Une autre amélioration apportée est la récupération de la taille des fichiers. Initialement, nous avons utilisé un compteur dans une boucle while qui lisait tout le fichier afin de récupérer la taille. Notre version finale utilise la fonction $ftell$ placé à la fin du fichier, ce qui est bien plus efficace.

De même, nous n'utilisions pas de fonctions de hachage et par la suite nous avons ajouté SHA_256 que nous utilisons via le terminal.

Également, notre programme gère aussi les erreurs de gestion de fichiers grâce à des goto et évite ainsi l'écrasement accidentel de fichiers existants.

Dans notre programme nous utilisions le padding PKCS#1 v1.5 et donc une des améliorations introduites est l'utilisation du padding OAEP, plus résistant en pratique et recommandé par la version actuelle de PKCS#1. Cela demande alors l'utilisation d'une fonction de hachage pouvant elle aussi être implémentée.

5 Les améliorations possibles

On pourrait également écrire une fonction pour que le programme possède son propre générateur aléatoire au lieu d'utiliser celui de la bibliothèque GMP. Dans la même idée, on pourrait ajouter au programme sa propre fonction de hachage.

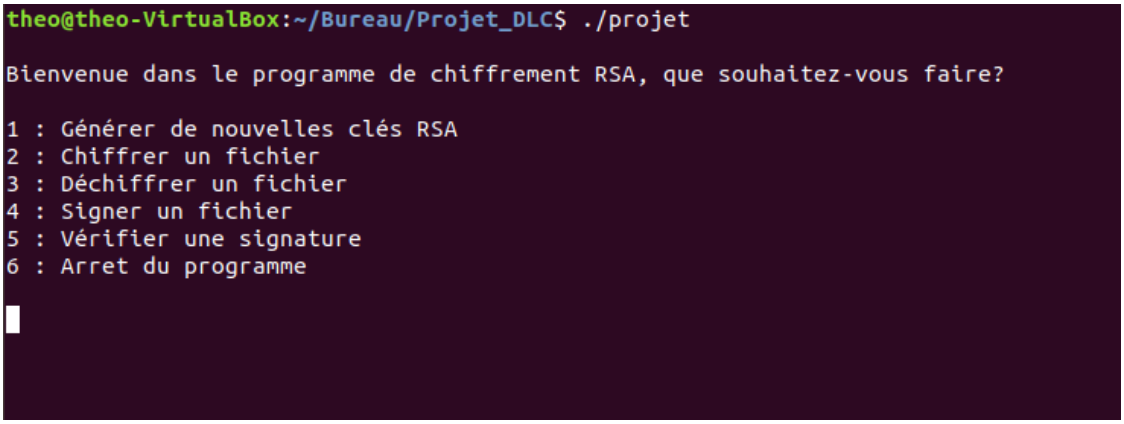
Plus spécifiquement, en ce qui concerne l'optimisation, une première amélioration serait d'optimiser l'espace mémoire utilisé. Par exemple, lors de l'ouverture du fichier contenant les clés privées nous pourrions éviter de charger p et q lorsque nous sommes en mode standard. Une deuxième optimisation serait de choisir un crible plus performant mais plus difficile à implémenter.

Il est également possible d'implémenter la fonction de hachage SHA256 afin de ne pas dépendre d'openSSL.

Ce projet se base sur le fait que l'on n'a pas toujours la place d'importer la bibliothèque GMP. Une solution à ce problème consisterait à créer une bibliothèque GMP plus légère contenant seulement les fonctions nécessaires au fonctionnement du programme.

6 Un exemple d'utilisation

Lors du lancement, notre programme propose à l'utilisateur un choix d'action à effectuer :



6.1 La génération de clés RSA

```
Quelle est la longeur de la clé publique souhaitée (en bit) ?

1024

Quel est le nom du fichier dans lequel vous désirez stocker la clé publique?

publique

Quel est le nom du fichier dans lequel vous désirez stocker la clé privée?

privee

Que souhaiez-vous faire maintenant?

1 : Générer de nouvelles clé RSA
2 : Chiffrer un fichier
3 : Déchiffrer un fichier
4 : Signer un fichier
5 : Vérifier une signature
6 : Arret du programme
```

Si l'on rentre le nom d'un fichier déjà existant pour stocker les clés, deux cas interviennent :

- ✈ soit on écrase le fichier déjà existant :

```
Quel est le nom du fichier dans lequel vous désirez stocker la clé publique?

publique

Attention, le nom de fichier saisi existe déjà, êtes-vous sûr de vouloir l'effacer?[Y/N]

Y
```

- ✈ soit on peut changer le nom que nous avions entré :

```
Quel est le nom du fichier dans lequel vous désirez stocker la clé publique?

publique

Attention, le nom de fichier saisi existe déjà, êtes-vous sûr de vouloir l'effacer?[Y/N]

N

Quel est le nom du fichier dans lequel vous désirez stocker la clé publique?

new_pblique
```

6.2 Chiffrer un fichier

```
Quel est le nom du fichier contenant la clé publique?

publique

Quel est le nom du fichier que vous désirez chiffrer?

image

Quel est le nom du fichier dans lequel vous désirez stocker le chiffré ?

chiffre

Que souhaiez-vous faire maintenant?

1 : Générer de nouvelles clé RSA
2 : Chiffrer un fichier
3 : Déchiffrer un fichier
4 : Signer un fichier
5 : Vérifier une signature
6 : Arret du programme
```

On obtient donc le fichier chiffré suivant :

projet.c × chiffre ×

```
1  0000 0080 064d 591b 18c4 c5f3 eea7 d540
2  936c b8fa 2f93 83da 8a35 da09 f116 45d1
3  1fd3 9c00 d342 ca62 be99 6147 b061 aa58
4  4343 1094 4719 d6dc 403a 4681 ba78 bae8
5  9d1e f596 982e 399c 0584 b45e cd83 5d80
6  ad10 9fa2 596e cfe8 aa44 bf3c cbfd 504d
7  937b b0b7 6b9f 21be f085 8e6e d386 cc10
8  7d5c da36 1477 ea85 2711 2078 b2d3 bccd
9  db89 5e4b 0000 0080 6437 a263 ee56 76d3
10 a7c0 79cf d52c 5015 d139 550f 1af2 268f
11 8296 a33a 8ecc 4fe1 4f7a 2c93 8110 5f3c
12 ece1 1437 d8ba 07c6 938e 41a1 707c 195b
13 f59f d64d 6e8f be69 98aa f489 7ca6 4b7a
14 bcf7 0814 645d b8e8 8366 f8b6 3f7e a94f
15 6b98 b904 c67f b46c db5a 9137 dafe 0c3b
16 272d d511 8fa2 9e13 de10 0bcf 1a76 2cad
17 bde9 6712 8fcb 27e5 0000 0080 6e85 a4c2
18 de10 deb2 e10b 31b5 223a 33b4 c309 4ecb
19 a495 62af fe4d aa2e 4643 7721 980b 56df
20 6128 1912 8b72 6ca6 a04c 3bde 0e79 28c4
21 6329 56ed a2fc ca73 d524 3b81 2e3f 7253
```

6.3 Déchiffrer un fichier

Pour déchiffrer on peut choisir entre deux possibilités :

✈ soit on déchiffre en mode classique :

✈ soit on déchiffre en mode crt :

Ces deux manières de déchiffrer donnent exactement le même résultat :

6.4 Signer un fichier

```
Quel est le nom du fichier contenant la clé publique?

publique

Quel est le nom du fichier que vous désirez signer?

image

Quel est le nom du fichier dans lequel vous désirez stocker la signature?

signature

Quel est le nom du fichier contenant la clé privée?

privee

Que souhaitez-vous faire maintenant?

1 : Générer de nouvelles clé RSA
2 : Chiffrer un fichier
3 : Déchiffrer un fichier
4 : Signer un fichier
5 : Vérifier une signature
6 : Arret du programme
```

6.5 Vérifier une signature

Deux cas peuvent se produire :

✈ soit la signature et le fichier original correspondent :

```
Quel est le nom du fichier contenant la signature?

signature

Quel est le nom du fichier contenant la clé publique?

publique

Quel est le nom du fichier original dont vous voulez vérifier la signature associée?

image

La signature est valide!


Que souhaitez-vous faire maintenant?

1 : Générer de nouvelles clé RSA
2 : Chiffrer un fichier
3 : Déchiffrer un fichier
4 : Signer un fichier
5 : Vérifier une signature
6 : Arret du programme
```

✈ soit la signature et le fichier original ne correspondent pas :

```
Quel est le nom du fichier contenant la signature?

signature

Quel est le nom du fichier contenant la clé publique?

publique

Quel est le nom du fichier original dont vous voulez vérifier la signature associée?

image2

La signature est invalide.
```

6.6 Arrêt du programme

```
Arret du programme.
theo@theo-VirtualBox:~/Bureau/Projet_DLC$
```