Université de Toulon
Licence 2 Informatique

Langage C
Année 2020-2021

Projet : L'algorithme de chiffrement RSA et la signature numérique

1 Présentation

L'algorithme de chiffrement RSA est un algorithme de cryptographie asymétrique. Il permet, comme tout schéma de chiffrement à clé publique, à deux entités d'avoir des échanges sécurisés sans la nécessité d'échanger au préalable une clé scécète.

Article bien détaillé :
`https://fr.wikipedia.org/wiki/Chiffrement_RSA`

2 Objectif

L'objectf ici est d'implémenter l'algorithme de chiffrement RSA ainsi que la signature numérique avec cet algorithme.

3 L'algorithme RSA et la signature numérique

3.1 L'algorithme RSA

Création des clés

Comme pour schéma de chiffrement à clé publique, le processus de génération de clés produit deux clés. L'une sera publique et l'autre sécrète :

1. Générer deux nombres premiers distincts p et q.
2. Calculer le module de chiffrement : $n = p * q$.
3. Calculer la valeur de l'indicatrice d'Euler en n : $\phi(n) = (p-1) * (q-1)$.
4. Choisir l'exposant de chiffrement e tel que : $e \in \mathbb{Z}/\phi(n)\mathbb{Z}$ et pgcd(e, $\phi(n)$) = 1.

5. Calculer l'exposant de déchiffrement d : c'est l'inverse de e modulo $\phi(n)$.

Les paramètres p et q, qui doivent rester sécrets, peuvent être supprimés.
Le paramètre n est publique.
L'entier e est la clé publique et l'entier d est la clé privée.

Le chiffrement

Soit $m \in \mathbb{Z}/n\mathbb{Z}$, le message à chiffrer.
Le chiffré c de m est calculé comme suit : $c = m^e \pmod{n}$.

Le déchiffrement

Soit $c \in \mathbb{Z}/n\mathbb{Z}$, le message à déchiffrer.
Le message clair m correspondant à c est calculé comme suit : $m = c^d \pmod{n}$.

3.2 Création de l'empreinte d'un document

La signature d'un document se fait sur l'empreinte de ce dernier car signer le document en entier pourrait être trop couteux en temps et en ressources. L'empreinte d'un document s'obtient à l'aide des fonctions de hachage. Une fonction de hachage produit une empreinte de taille donnée à partir d'un document de taille quelconque. La page suivante présente assez bien les fonctions de hachage ainsi que les plus populaires : `https://en.wikipedia.org/wiki/Cryptographic_hash_function`

Dans ce projet, nous implémenterons une fonction de hachage fictive **MD9**. Elle prend en paramètres un message M de taille quelconque et la taille k de l'empreinte à produire. Elle retourne une empreinte de k bits.
Le protocole :

1. Compléter le message M avec autant de zéros que nécessaire pour que sa taille soit un multiple de k.
2. Diviser M en blocs de k bits : $M = M_0||M_1||\dots||M_t$. ($||$ désigne la concaténation)
3. L'empreinte de M est alors : $M = M_0 \oplus M_1 \oplus \cdots \oplus M_t$. ($\oplus$ désigne le xor)

3.3 La signature numérique

La signature numérique (ou signature électronique) est un mécanisme permettant de garantir l'intégrité d'un document électronique et d'en authentifier l'auteur.
Article bien détaillé : `https://fr.wikipedia.org/wiki/Signature_numérique`

Pour créer une signature numérique, on a besoin d'une fonction de hachage et d'un algorithme de chiffrement asymétrique. Dans ce projet, nous utiliserons la fonction de hachage fictive MD9 (définie plus haut) ainsi l'algorithme RSA.

Soient n, e et d l'ensemble des paramètres de l'algorithme de chiffrement RSA et k la taille en bits de n.

Signature

Soit M le document à signer. La signature se fait comme suit :

1. $c = MD9(M,\, k-1)$
2. $m = c^d \pmod{n}$ (correspond au déchiffrement)

m est la signature du document M.

Vérification

Pour vérifier que m est bien une signature de M, on procède comme suit :

1. $c_1 = m^e \pmod{n}$ (correspond au chiffrement)
2. $c_2 = MD9(M,\, k-1)$

Si c_1 est égal à c_2 alors la signature est valide, sinon elle ne l'est pas.

4 À propos de l'exponentiation modulaire

Soient g, x et p trois entiers. Le calcul $g^x \pmod{p}$ est une exponentiation modulaire. Lorsque les entiers manipulés sont suffisamment grands, l'approche naïve devient impossible en temps et en espace car implique trop un nombre trop élévé d'itérations et des produits temporaires trop grands. Une méhtode efficace pour effectuer cette opération est l'exponentiation modulaire binaire (aussi appelé *square-and-multiply*). Cette méthode utilise la représentation binaire de l'exposent pour accélérer de manière considerable le calcul $g^x \pmod{p}$.
Liens détaillants assez bien cette méthode :

— `https://fr.wikipedia.org/wiki/Exponentiation_modulaire`
— `https://en.wikipedia.org/wiki/Exponentiation_by_squaring`

5 À propos des grands entiers

Pour que ce protocole soit sûr, il est indispensable d'utiliser de grands entiers pour les opérations arithmétiques. Par défaut, les machines classiques permettent en général de manipuler des entiers de 64 bits au maximum. Cette taille est largement inférieure à

celles utilisées en cryptographie à clé publique qui varient de 256 à des milliers de bits. Par conséquent, il est donc nécessaire d'avoir recours à des librairies pour la manipulation de grands entiers.

L'une des librairies les plus populaires et performantes dans ce domaine est la librairie GNU MP : `https://gmplib.org/`. Cette librairie permet de manipuler de grands nombres (pas seulemnt les entiers). Ici, nous ne nous intéresserons qu'à la manipulation des grands entiers. Voici quelques liens intéressants vers sa documention :

1. `https://gmplib.org/manual/` : la page principale de la documentation.
2. `https://gmplib.org/manual/GMP-Basics.html` : pour les connaissances générales de base sur la librairie.
3. `https://gmplib.org/manual/Integer-Functions.html` : pour la manipulation de grands entiers.
4. `https://gmplib.org/manual/Formatted-Input.html` : pour la gestion des entrées.
5. `https://gmplib.org/manual/Formatted-Output.html` : pour la gestion des sorties.
6. `https://gmplib.org/manual/Random-Number-Functions.html` : pour la génération de nombres aléatoires.

Ci-dessous, un exemple d'utilisation de la librarie. Ce code génère un nombre premier p de n bits, ensuite génère aléatoirement deux entiers a et b dans $\mathbb{Z}/p\mathbb{Z}$.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <time.h>
4  #include <gmp.h>
5
6
7  int main(void){
8
9    int n;
10   unsigned long seed;
11   mpz_t a, b, c, p;
12   gmp_randstate_t r;
13
14   mpz_inits (a, b, c, p, NULL);
15
16   seed = time(NULL);
17   gmp_randinit_default(r);
```

```
18    gmp_randseed_ui(r, seed);
19
20    n = 256;
21    // generation d'un nombre premier de n bits
22    do{
23      mpz_urandomb (p, r, n);
24      mpz_setbit (p, n-1);
25      mpz_nextprime (p, p);
26    }while (mpz_sizeinbase (p, 2) != n);
27
28    // generation de deux nombres aleatoires inferieurs a p
29    mpz_urandomm (a, r, p);
30    mpz_urandomm (b, r, p);
31    gmp_printf("\na = %Zd\n\n", a);
32    gmp_printf("b = %Zd\n\n", b);
33    gmp_printf("p = %Zd\n\n", p);
34
35    // addition
36    mpz_add (c, a, b);
37    gmp_printf("addition, c = %Zd\n\n", c);
38
39    // soustraction
40    mpz_sub (c, a, b);
41    gmp_printf("soustraction, c = %Zd\n\n", c);
42
43    // multiplication puis reduction modulaire
44    mpz_mul (c, a, b);
45    mpz_mod (c, c, p);
46    gmp_printf("multiplication modulaire, c = %Zd\n\n", c);
47
48    // exponentiation modulaire
49    mpz_powm (c, a, b, p);
50    gmp_printf("exponentiation modulaire, c = %Zd\n\n", c);
51
52    mpz_clears (a, b, c, p, NULL);
53    gmp_randclear(r);
54
55    return 0;
56  }
```

Remarques :

1. Pour utiliser la librairie GNU MP, il faut l'avoir installé auparavant ; elle est normalement déjà installée sur les machines de TP.
   Lien pour le téléchargement : `https://gmplib.org/#DOWNLOAD`
   Lien pour l'installation : `https://gmplib.org/manual/Installing-GMP.html`
2. Compilation : l'utilisation de cette librairie nécessite l'ajout de l'option 'lgmp' à la commande de compilation. Ainsi, si le code ci-dessus est enrégistré dans le fichier 'exemple.c', il pourra être compilé avec la commande :
   gcc -Wall exemple.c -o exemple -lgmp

6 Taches à effectuer

Pour ce projet, on utilisera la librairie GNU MP pour la manipulation des grands entiers. Les taches à effectuer sont les suivantes :

1. Implémenter l'exponentiation modulaire comme expliquée dans la section 4.

2. Implémenter le processus de génération de paramètres pour l'algorithme RSA.
   Ce processus prendra en paramètres les tailles lp et lq (en bits) des nombres premiers p et q qui composent le module de chiffrement n. Il faudra s'assurer que n soit de taille $(lp + lq)$ bits.

3. Implémenter les algorithmes de chiffrement et de déchiffrement pour RSA.

4. Implémenter les procédures de signature et de vérification comme expliquée plus haut.

5. Ecrire une fonction (la fonction *main*) qui :

   — Demande à l'utilisateur la taille k en bits des signatures à produire.

   — Génère les paramètres pour l'algorithme de chiffrement RSA.
   Remarque : Le module de chiffrement n devra être de taille $lp + lq = k$ bits. De plus, on prendra $lq = lp = k/2$ si k pair. Sinon, on prendra $lp = (k-1)/2$ et $lq = lp + 1$.

   — Lit un fichier texte et produit sa signature numérique, comme expliqué plus haut.

   — Vérifie que la signature est correcte.