Introduction et historique

Ce TP se propose d'illustrer quelques notions simples de la cryptographie à clef publique.

Les principes généraux de la cryptographie ainsi que les outils mathématiques nécessaires sont présentés au fur et à mesure du TP. Nous allons, pendant cette séance, mettre en oeuvre la méthode de chiffrement RSA sur un exemple, en utilisant un compilateur C et la bibliothèque mathématique GMP (Gnu Multiprecision Arithmetic Library).

RSA est le système de chiffrement à clef publique le plus utilisé au monde : on le trouve dans les cartes bancaires, dans le système d'identification Kerberos, dans le standard de sécurisation de réseaux IPsec ou encore dans les décodeurs des chaînes numériques cryptées. Il est également utilisé dans le protocole sécurisé SSH, dans les cartes à puces de France Télécom, dans Internet Explorer ou dans Netscape Navigator. Près de 500 millions de logiciels l'utilisent actuellement dans le monde. Voici quelques rappels du cours précédent :

Le but d'un cryptosystème est d'assurer la confidentialité d'un message appelé texte en clair.
Le processus de transformation de ce message pour le rendre incompréhensible est appelé chiffrement.
Le processus de reconstruction du texte en clair à partir du texte chiffré est le déchiffrement.
Le texte en clair sera noté m ou x (suite de bits, fichier texte, voix numérisée, vidéo, etc.).
Le texte chiffré sera noté c ou y.
La fonction de chiffrement notée f ou E est la fonction définie par $f(x) = y$.
La fonction de déchiffrement est sa réciproque f^{-1} (ou D) définie par $f^{-1}(y) = x$.
On a bien sûr $f \circ f^{-1} = \text{Id}$ et ces fonctions peuvent dépendre de paramètres appelés clefs.

Il existe deux principaux types d'algorithmes cryptographiques : les algorithmes à clef secrète et les les algorithmes à clef publique :

• Dans les algorithmes à clef secrète, chaque intervenant (l'expéditeur et le destinataire) conviennent d'une clef secrète (un mot, un ou plusieurs nombres, etc.) commune qu'ils utiliseront pour chiffrer et déchiffrer les messages. La sécurité du chiffrement repose alors sur la confidentialité de cette clef. Si l'un des deux la perd ou la dévoile le système peut être piraté.

• Dans les algorithmes à clef publique, chaque intervenant possède une clef différente en deux parties. L'une des parties peut être dévoilée et l'autre doit rester secrète (en ce cas seul le possesseur de la clef connaît sa valeur). Ce type d'algorithme est beaucoup plus sûr car il n'y a aucun secret partagé et il peut être utilisé sans que les deux intervenants ne se soient jamais rencontrés. Il repose sur la difficulté de calculer l'inverse d'ne fonction même lorsque celle ci est connue. De telles fonctions s'appellent fonctions à sens unique.

Le concept de cryptographie à clef publique a été inventé par Diffie, Hellman et Merkle en 1976 dans l'article « New directions in cryptography ». C'est en 1977 que trois chercheurs du MIT, Ron Rivest, Adi Shamir et Leonard Adleman présentent une méthode simple et efficace pour mettre en oeuvre l'idée de Diffie et Hellman : les clefs publiques et privées sont calculées à partir de très grands nombres premiers et la fonction à sens unique repose sur la difficulté de décomposer en facteurs premiers les très grands nombres. Les articles originaux (en anglais) sont à votre disposition si vous souhaitez les consulter. Reportez-vous également à la bibliographie si vous souhaitez en savoir plus.

1 Quelques rappels mathématiques.

1.1 Les mots sont des nombres !

Comme il est expliqué plus loin, la méthode RSA s'applique sur des données numériques ; lorsque l'on souhaite chiffrer un message alphabétique, il est nécessaire de le convertir en nombre.

Pour cela, on utilise la table des codes ascii qui permet de convertir chaque lettre, signe ou chiffre en un nombre compris entre 0 et 255. On peut donc considérer qu'un mot est un nombre exprimé en base 256, chaque chiffre de ce nombre pouvant prendre 256 valeurs différentes.

Un nombre n exprimé en base b s'écrit sous la forme $d_k d_{k-1} ... d_1$ si et seulement si $n = \sum_{i=1}^{k} d_i b^{i-1}$

Ex : En décimal ($b = 10$), les digits sont $0, 1, 2, ..., 9$ et $n = 1823 \iff n = 3 + 2 \times 10 + 8 \times 10^2 + 3 \times 10^3$

Ex : En binaire ($b = 2$), les digits sont $0, 1$ et $n = 1011 \iff n = 1 + 1 \times 2 + 0 \times 2^2 + 1 \times 2^3$
Ex : En base 256, les digits sont $0, 1, ..., 255$ et $n =$BONJOUR
$\iff n = 82 + 85 \times 256 + 79 \times 256^2 + ... + 66 \times 256^6$

Le nombre de chiffres (on dira aussi de digits) d'un entier n exprimé en base b est

$$k = \lfloor \frac{\ln n}{\ln b} \rfloor + 1 = \lfloor \log_b n \rfloor + 1$$

où $\lfloor \quad \rfloor$ représente la partie entière.

k est alors l'unique entier vérifiant $b^{k-1} \leq n < b^k$
Le $i$ème digit en base b d'un nombre décimal n est donné par $d_i = \lfloor \frac{n}{b^{i-1}} \rfloor \mod b$

Le nombre de bits occupés en mémoire par un entier n n'est autre que le nombre de digits en écriture binaire. Lorsque l'entier est compris entre 2^{n-1} et 2^n, il peut alors s'exprimer à l'aide de n bits. La longueur des clefs en cryptographie est un paramètre très important et nous reviendrons, par la suite, sur les calculs de ce paragraphe.

1.2 Principe de RSA et arithmétique modulaire.

Dans un système à clef publique, chaque utilisateur possède une clef en deux parties : une clef publique et une clef privée. La clef publique E (pour « encryption » et nous l'avons également appelée f) sert habituellement à chiffrer le message et la clef privée D (pour « decipher » et nous l'avons aussi appelée f^{-1}) à le déchiffrer. E et D doivent être bien sûr réciproques l'une de l'autre.

Lorsque Alice veut envoyer un message à Bernard, elle récupère sa clef publique dans un annuaire, chiffre son message à l'aide de cette clef et l'envoie à Bernard. Celui-ci étant le seul à connaitre la fonction D, il est aussi le seul à pouvoir le déchiffrer. Dans RSA, les clefs E et D se calculent de la façon suivante :

1°. Chaque utilisateur choisit deux grands nombres premiers p et q.
2°. Il calcule $n = pq$ ainsi que le nombre $\phi(n) = (p-1)(q-1)$
3°. Il choisit un nombre e inversible modulo $\phi(n)$, c'est à dire tel que $\text{pgcd}(e, \phi(n)) = 1$
4°. Il calcule l'inverse d de e modulo $\phi(n)$, c'est à dire $d = e^{-1} \mod \phi(n)$
5°. Il détruit p et q qui ne serviront plus.

Tous ces calculs mathématiques seront exécutés par des fonctions qui existent déjà dans la bibliothèque GMP. Il n'y aura donc pas de calculs à effectuer par vous ou de fonctions mathématiques à coder.

La clef publique de l'utilisateur sera formée du couple (n, e) et sa clef privée de (n, d).

La première opération consiste à transformer un message alphabétique en un nombre entier selon la méthode expliquée dans le paragraphe précédent. L'entier x obtenu doit être plus petit que n afin de n'avoir qu'une seule représentation modulo n. Si tel n'est pas le cas, il faut découper le message en plusieurs blocs dont la longueur ne dépasse pas n bits.

Pour chiffrer un message x, l'utilisateur calcule $y = x^e \mod n$
Pour déchiffrer un message y, il calculera $y^d \mod n = x^{ed} \mod n = x \mod n$

En effet, e et d sont inverses l'un de l'autre modulo $\phi(n)$ et le théorème de Fermat-Euler démontre alors l'égalité (cf. cours).

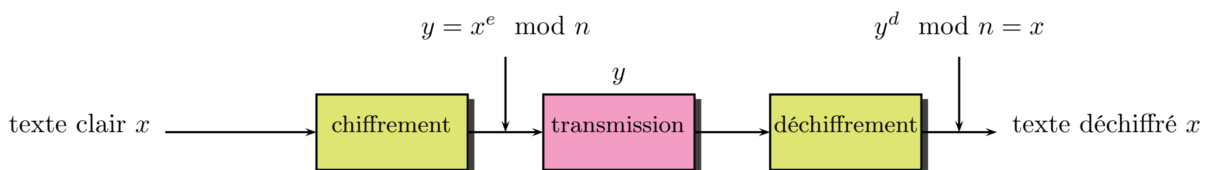

FIGURE 1 – Protocole RSA

2 La bibliothèque GMP.

2.1 Installation.

Si vous travaillez sous Visual C++ version 6 (ce qui est conseillé), les fichiers se trouvent sur le CD qui vous sera distribué. Il suffit de déposer le répertoire `gmp-4.1` dans votre répertoire de travail, puis de créer avec visual C++ un nouveau « worskspace » dans le répertoire `gmp-4.1`. Pour cela, aller dans `File`, `New`, onglet `workspaces`. Tapez

le nom de votre « worskspace » (par exemple rsa) et choisissez comme emplacement l'intérieur du répertoire `gmp-4.1`. Créez alors le fichier source dans lequel vous taperez votre programme en sélectionnant `File`, `New`, onglet `files`, puis `C++ source file`. Ce fichier doit se trouver dans le répertoire `gmp-4.1`, juste à la hauteur du dossier `rsa` de votre « worskspace ». Dans ce fichier, en préambule de toute déclaration, vous ajouterez la ligne :

```
#pragma comment(lib, "gmp.lib")
```

afin que le compilateur trouve les librairies correspondantes. Ce n'est pas très propre, mais cela devrait fonctionner. Enfin, vous indiquerez dans les déclarations de bibliothèques la ligne `#include "gmp.h"`.

Si vous travaillez sous DevC++, il suffit d'aller dans `Outils`, `Nouvelles versions/packages...` et de sélectionner la bibliothèque GMP. Elle s'installe alors automatiquement. Il faut également indiquer au compilateur d'utiliser cette bibliothèque pendant l'édition de liens. Pour cela, aller dans `Outils`, `Options du compilateur`, onglet `Compilateur`, cliquez sur `Ajouter ces commandes lors de l'édition de liens`, puis tapez `-lgmp` dans la fenêtre. Si cela ne fonctionne pas, vous avez un répertoire dans le CD qui vous est donné et qui contient la bibliothèque déjà compilée. Dans tous les cas, il faut indiquer dans le fichier source `#include "gmp.h"`.

Si vous travaillez sous linux avec gcc, il faut installer la bibliothèque avec `./configure` puis `make`. Les sources se trouvent sur le CD dans le répetoire `linux`. Reportez-vous à la documentation fournie sur le CD et sur le site http://gmplib.org/ car chaque distribution a sa propre méthode d'installation. La compilateur nécessite d'utiliser la commande `-lgmp` pour inclure la bibliothèque lors de l'édition de liens.

Si vous êtes allergiques au langage C, vous pouvez également faire le TP avec le logiciel de calcul formel Maple. Enfin, si vous ne voulez travailler que sur des logiciels libres, vous pouvez également utiliser Xcas.

2.2 Utilisation.

La bibliothèque GMP permet de travailler avec des nombres entiers de longueur arbitraire et d'avoir à disposition toutes les fonction arithmétiques utiles en cryptographie. La documentation au format `.pdf` est disponible, mais vous trouverez ci-dessous une liste de fonctions qui devrait suffire à l'élaboration de votre programme. Votre fichier source devra comprendre la ligne `#include "gmp.h"`.

Vous avez à votre disposition les types de données suivants :

`mpz_t`	grand nombre entier.
`gmp_randstate_t`	générateur de nombres aléatoires.
`mp_bitcnt_t`	longueur d'un grand nombre entier en bits.
`size_t`	longueur d'un grand nombre entier en caractères.

Dès que l'on déclare une variable de type `mpz_t` et avant de l'utiliser, il faut obligatoirement l'initialiser pour que de la mémoire lui soit allouée. Cela se fait grâce à la fonction `mpz_init(nombre)` où `nombre` est une variable de type `mpz_t`.

Pour initialiser le générateur pseudo-aléatoire de GMP et pouvoir engendrer des nombres premiers aléatoires, vous devrez taper les lignes suivantes :

```
gmp_randinit_default(etat);
srand(time(NULL));
mpz_set_ui(germe,rand()*rand());
gmp_randseed(etat,germe);
```

La fonction `time` se trouve dans la biblothèque `time.h`. Elle sert à définir le germe du générateur à l'aide d'une fonction dont la valeur change tout le temps (et pour cause). Les lignes précédentes définissent un générateur par défaut qui s'appelle `etat` et dont les valeurs sont initialisées à partir d'un nombre appelé `germe`.

Voici les prototypes de fonctions utiles et leur action :

Fonctions de conversion et d'affectation :

`void mpz_init(mpz_t x)`	initialise la valeur de `x` à 0 et alloue de la mémoire.
`void mpz_clear(mpz_t x)`	efface la mémoire allouée à `x`.
`void mpz_set(mpz_t y,mpz_t x)`	copie la valeur de `x` dans `y`.
`void mpz_set_ui(mpz_t x,unsigned long int n)`	convertit un entier long `n` en un grand nombre `x`.
`int mpz_init_set_str(mpz_t x,char *chaine,int base)`	convertit une chaîne écrite en base `base` en nombre `x`.
`char* mpz_get_str(char *chaine,int base,mpz_t x)`	convertit `x` en une chaîne `chaine` écrite en base `base`.
`size_t mpz_sizeinbase(mpz_t x, int b)`	nombre de digits en base b pour représenter x.

Fonctions arithmétiques :

`void mpz_add(mpz_t z,mpz_t x,mpz_t y)`	calcule `z=x+y`.
`void mpz_add_ui(mpz_t z,mpz_t x,unsigned long int n)`	calcule `z=x+n`.

`void mpz_sub(mpz_t z,mpz_t x,mpz_t y)`	calcule `z=x-y`.
`void mpz_sub_ui(mpz_t z,mpz_t x,unsigned long int n)`	calcule `z=x-n`.
`void mpz_mul(mpz_t z,mpz_t x,mpz_t y)`	calcule `z=x×y`.
`void mpz_mul_ui(mpz_t z,mpz_t x,unsigned long int n)`	calcule `z=x×n`.
`void mpz_divexact(mpz_t z,mpz_t x,mpz_t y)`	calcule `z=x/y` pour trois grands entiers `x,y,z`.
`void mpz_divexact_ui(mpz_t z,mpz_t x,unsigned long int n)`	calcule `z=x/n`.

Fonctions d'arithmétique modulaire :

`void mpz_powm(mpz_t y,mpz_t x,mpz_t a,mpz_t n)`	calcule $y = x^a \mod n$.
`void mpz_powm_ui(mpz_t y,mpz_t x,unsigned long int a,mpz_t n)`	calcule $y = x^a \mod n$.
`void mpz_pow_ui(mpz_t y,mpz_t x,unsigned long int a)`	calcule $y = x^a$.
`void mpz_ui_pow_ui(mpz_t y,unsigned long int x,unsigned long int a)`	calcule $y = x^a$.

`int mpz_probab_prime_p(mpz_t x,int n)`	Teste si un nombre x est premier avec n itérations.
`void mpz_nextprime(mpz_t p,mpz_t x)`	Trouve le nombre premier p immédiatement supérieur à x.
`void mpz_invert(mpz_t d,mpz_t e,mpz_t n)`	Calcule $d = e^{-1} \mod n$.

Fonctions de comparaison :

`void mpz_cmp(mpz_t x,mpz_t y)`	renvoie un nombre positif si $x > y$, 0 si $x = y$ et un nombre négatif sinon.
`void mpz_sgn(mpz_t x)`	renvoie 1 si $x > 0$, 0 si $x = 0$ et -1 si $x < 0$.

On dispose enfin d'une fonction `void mpz_urandomm(mpz_t x,gmp_randstate_t etat,mpz_t n)` qui engendre, de façon uniforme, un nombre entier aléatoire compris entre 0 et $n - 1$.

3 Travail à effectuer.

3.1 Fonction de chiffrement et de déchiffrement.

A l'aide des fonctions précédentes, écrire un programme permettant de créer des clefs RSA, de chiffrer et de déchiffrer des textes selon l'algorithme RSA. Votre programme devra pouvoir effectuer les actions suivantes :

- Création de la partie publique et privée d'une clef RSA dont la longueur en bits sera définie.
- Lecture d'un texte au clavier ou depuis un fichier texte.
- Conversion du texte en nombre entier et conversion d'un entier décimal en chaîne de caractères.
- Chiffrement du texte avec affichage du résultat à l'écran ou dans un fichier texte.
- Déchiffrement du texte pour retrouver le message initial.

Vous donnerez plusieurs exemples d'exécution pour des tailles de clefs différentes et réalistes.

3.2 Authentification par la méthode RSA.

En plus d'assurer la confidentialité des messages, le protocole RSA permet d'assurer également l'authentification et interdit par conséquent une attaque du type « man in the middle ».

Imaginons qu'Alice et Bernard possèdent chacun une clef RSA. Notons (n_A, e_A) la clef publique d'Alice, (n_A, d_A) sa clef privée, (n_B, e_B) la clef publique de Bernard et (n_B, d_B) sa clef privée. Alice veut envoyer un message à Bernard. Elle commence par chiffrer le message en utilisant sa clef privée. Puis elle chiffre le résultat avec la clef publique de Bernard et lui envoie le tout. Bernard déchiffre ce qu'Alice lui a envoyé. Il est le seul à pouvoir le faire, puisqu'il est le seul à posséder la clef secrète correspondante. Ensuite, il utilise la clef publique d'Alice pour déchiffrer ce qu'il a obtenu après le premier déchiffrement. Si ce dernier message est un texte en clair, alors c'est obligatoirement Alice qui l'a envoyé puisqu'elle était la seule à posséder la clef secrète correspondante. Ainsi, chacun est sûr que le message a bien été envoyé et reçu par les bonnes personnes.

Echangez avec quelqu'un d'autre votre clef publique et envoyez-lui par mail un message en suivant le protocole ci-dessus. Vérifiez que l'on retrouve bien les messages initiaux.

3.3 Casser un code RSA.

Casser un code RSA est équivalent à factoriser $n = p \times q$. En effet, connaissant p et q, on peut calculer facilement $\phi(n) = (p - 1) \times (q - 1)$ et en déduire d qui est à la fois l'inverse de e modulo $\phi(n)$ et la clef secrète.

Vous venez d'intercepter un message chiffré à destination d'une personne P. Ce message se trouve dans le fichier `secret.txt`. Vous savez qu'il a été chiffré ligne par ligne selon la méthode RSA en utilisant l'alphabet ascii usuel. En vous reportant à l'annuaire des clefs publiques, vous trouvez celle de P. Les valeurs de e et n qui constituent cette clef se trouvent dans les fichiers `e.txt` et `n.txt`.

C'est mal, mais vous allez tenter de pirater ce message. Vous pouvez utiliser pour cela le programme `factorize.c` qui se trouve dans le répertoire `demos` de GMP, ou le programme exécutable `factor.exe` qui se trouve sur le CD, ou bien encore Maple ou Xcas. Attention, il faut plusieurs minutes pour que le calcul aboutisse...

Finalement, vous pouvez recommencer les opérations précédentes avec les fichiers `secretbis.txt` `ebis.txt` et `nbis.txt` pour déchiffrer un second message avec une clef plus conséquente (comptez plusieurs dizaines d'heures pour la factorisation).

3.4 La taille des clefs publiques en 2010.

L'exemple précédent est une clef de très faible taille et peut se factoriser rapidement.

Les cartes bleues de l'ancienne génération possèdent une clef publique RSA de 320 bits (96 chiffres décimaux). C'est cette clef qui a été factorisée durant l'été 1998. En quelques heures, on peut aujourd'hui factoriser une telle clef sur un ordinateur de bureau. Les cartes bleues actuelles ont une clef RSA de 768 ou 1024 bits (231 chiffres décimaux). En 1999, une équipe d'informaticiens a réussi à factoriser un nombre de 512 bits en quelques mois avec une centaine de machines. En novembre 2005, c'est un nombre de 640 bits qui a été factorisé (l'équipe de mathématiciens a empoché 20000 $).

Au mois de décembre 2009, un nombre de 768 bits a été factorisé par une équipe internationale de chercheurs appartenant à six laboratoires universitaires. Il aura fallu deux ans de calculs pour parvenir au résultat, en utilisant un cluster de 1700 processeurs. Pour factoriser un nombre RSA de 1024 bits, on estime qu'il faudrait multiplier par 1000 la puissance de calcul nécessaire et que cela pourrait bien être possible d'ici une dizaine d'année. L'article présentant la factorisation a été publié la semaine dernière (des photocopies sont disponibles).

On estime donc maintenant que pour être sûre, une clef RSA doit faire au moins 1024 bits de long.

La puissance de calcul des ordinateurs se calcule en Mips (million d'opérations par seconde).
Un Mips/an représente environ 3×10^{13} instructions.
10000 Mips est la puissance d'un Pentium IV ou d'un Athlon 64.
5×10^7 Mips est la puissance de calcul du TERA-10, le nouvel ordinateur du CEA (supercalculateur formé de 4352 processeurs Intel).

Une grande entreprise possède une puissance d'environ 10^8 Mips/an. Un gouvernement environ 10^{10} Mips/an.
Le tableau ci dessous donne la puissance de calcul (très approximative) nécessaire à la factorisation d'une clef :

Nb bits	Mips/an
512	8000
768	10^9
1024	10^{14}
2048	10^{20}