Principaux algorithmes de cryptage

Rolland Balzon Philippe
Department of Computer Science
SEPRO Robotique
ZI les ajoncs, 85000 La roche sur Yon, France
prolland@free.fr

11 juillet 2002

Table des matières

Introduction

Le but de ce rapport est de donner les grandes lignes des principaux algorithmes de cryptage, ainsi que de fournir un ordre d'idée sur leur niveaux de sécurité et leurs temps d'exécution. Nous pourrons ainsi effectuer une comparaison entre chacun d'eux en terme de coût machine et d'efficacité.

La principale distinction se fera entre les codages de types symétriques et ceux de types asymétriques ou à clé privé.

Notez qu'aucun des codes présentés par la suite n'atteind le secret parfait. On considérera cependant qu'un code est suffisamment sûr si le temps de calcul pour déchiffrer le message crypté sans la clé n'est pas envisageable à l'echelle humaine.

1 Cryptographie Symétrique

1.1 DES (*Data Encryption Standard*)

1.1.1 Historique

Créé dans les années 70 (public en 1981), l'algorithme DES a été l'algorithme de cryptographie le plus usité jusqu'à ces dernières années. Il a été recertifié depuis (environ tous les 5 ans) et est encore aujourd'hui utilisé (mais pas sous sa forme simple). Sa plus récente version date de 1994.

1.1.2 Principe

L'algorithme DES est un algorithme de cryptographie en bloc. Il opère généralement sur des blocs de 64 bits et utilise une clé de 56 bits qui sera transformée en 16 sous-clés de 48 bits chacune.

Le chiffrement se déroule en 16 tours (ou rounds).

1.1.3 Description

Avant de débuter les 16 tours on effectue la transformation suivante :

Soit un bloc de texte clair x, une chaîne de bits est construite en changeant l'ordre des bits de x suivant une permutation initiale (IP) fixée. On écrit $x_0 = IP(x) = L_0R_0$, où L_0 contient les 32 premiers bits de la chaîne x_0 et R_0 contient les 32 restants.

On effectue ensuite 16 itérations du type :

$$L_i = R_{i-1}$$
$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

où $\oplus$ est le *ou*-exclusif,

et f est une fonction qui prend pour argument une chaîne de 32 bits et une chaîne de 48 bits et renvoie une chaîne de 32 bits :
Elle augmente le premier argument de 32 à 48 bits suivant une fonction d'expansion E. Le résultat est alors additionné modulo 2 avec le second argument. Le résultat appelé B sera découpé en 8 sous-chaînes de 6 bits chacune : $B = B_1B_2B_3B_4B_5B_6B_7B_8$. L'étape suivante utilise 8 boîtes S : $S_1, ..., S_8$. Chacune des S_i peut être vue comme une fonction qui prend en entrée une chaîne de 6 bits et produit une chaîne de 4 bits. On calcule ainsi $C_i = S_i(B_i)$. La chaîne $C = C_1C_2C_3C_4C_5C_6C_7C_8$ de longueur 32 sera alors transformée par une permutation P fixée qui sera renvoyée comme étant le résultat de la fonction f.

1.1.4 Performances

Au cours des 25 dernières années, l'algorithme DES s'est révélé être un algorithme solide. La seule attaque connue à ce jour est purement académique, et n'a aucune conséquence sur la sécurité

pratique de l'algorithme. Malheureusement, sa faiblesse réside dans la longueur de sa clé, qui n'est que de 56 bits. En effet, certains calculs ont montrés qu'il était possible d'effectuer une recherche exhaustive de clé, en 2^{56} essais. Ces calculs sont réalisables en un temps limité et avec un coût relativement raisonnable.

Pour palier à cela a été mis au point le 2-DES, c'est-à-dire crypter deux fois de suite le message clair avec l'algoritme DES. Malheureusement, cela n'apporte pas véritablement de sécurité suplémentaire, car il suffit d'une "*attaque dans le milieu*" pour obtenir un temps de cryptanalyse coûteux, mais faisable.

La solution réside donc dans l'emploi d'un triple DES, contre lequel aucun type d'attaque efficace n'est connu. Le message est alors crypté avec une clé k_1 puis décrypté avec une clé k_2 avant d'être recrypté avec la clé k_1. L'utilisation de seulement 2 clés (au lieu de 3) ne diminue pas la sécurité mais diminue le temps de calcul. La meilleure attaque connue contre ce système est de l'ordre de 2^{112} (donc irréalisable), si la même clé n'est pas utilisée plus de 256 fois. Malheureusement, même si cette façon de coder s'avère sûre, elle reste assez lente. En effet, le cryptage par DES se fait à une vitesse relativement moyenne, et le 3-DES est pratiquement 3 fois plus lent. De plus l'algorithme DES ne traite que des blocs de (seulement) 64 bits.

En pratique DES peut être implémenté de manière efficace, en matériel ou en logiciel. En effet, les seules opérations arithmétiques à effectuer sont les *ou*-exclusifs de chaînes de bits. Les différents calculs de la fontion f pouvant être effectués par des accès de tables (en logiciel) ou par un cablage de circuit adapté (en matériel).

1.1.5 Modes d'utilisation

DES étant un algorithme de chiffrement par bloc, il peut être appliqué de plusieurs façons. En réalité, 4 modes d'utilisation de DES ont été proposés : ECB (*Electronic CodeBook mode*), CFB (*Cipher FeedBack mode*), CBC (*Cipher Block Chaining mode*) et OFB (*Output FeedBack mode*).

ECB correspond au mode le plus simple : étant donné un texte clair $x_1x_2...$, chaque bloc de 64 bits est chiffré avec l'algorithme DES, donnant un texte chiffré $y_1y_2...$ Ce système présente un inconvénient flagrant : 2 blocs identiques seront codés de la même manière, et auront donc le même bloc chiffré. Il est donc par exemple possible de recenser tous les cryptés possibles (code books) puis par recoupements et analyses statistiques de recomposer une partie du message original sans avoir tenté de casser la clé de chiffrement. Cependant, si la clé fait 128 bits ou plus, cette attaque n'est pas exploitable en pratique de nos jours. Cette technique reste malgré tout sensible à l'inversion ou à la duplication de blocs sans que le destinataire s'en aperçoive.

Dans le mode CBC, chaque bloc de texte chiffré y_i opère une action sur le bloc de texte clair suivant x_{i+1} avant qu'il ne soit chiffré. Il permet d'introduire une complexité supplémentaire dans le processus de cryptage en créant une dépendance entre les blocs successifs. On commence avec un bloc initial IV (*Initial Value*) en posant $y_0 = \text{IV}$. On construit ensuite $y_1y_2...$ en suivant la formule $y_i = e_K(y_{i-1} \oplus x_i)$, $i \geq 1$.
Il est à noter que c'est le mode le plus courant.

Le mode CFB, également destiné au chiffrement par bloc, permet d'en autoriser une utilisation plus souple, qui s'apparente plus à celle des algorithmes en continu. On peut le considérer comme un intermédiaire entre les deux. En effet, en partant d'un algorithme en bloc utilisant une longueur standard de n bits/blocs, le mode CFB va permettre de chiffrer des blocs dont la longueur pourra varier de n à 1 bits/blocs. Sachant que dans ce dernier cas, il serait plus économique en calculs d'utiliser directement un algorithme en continu. Quant au cas où la longueur est celle de l'algorithme (à savoir n), le schéma de CFB se simplifie et ressemble quelque peu à celui de CBC.
Plus précisément, on commence là aussi avec $y_0 = \text{IV}$ (un bloc initial de 64 bits), et l'on produit la clé z_i en chiffrant le bloc de texte chiffré précédent : $z_i = e_K(y_{i-1})$, $i \geq 1$. On a alors $y_i = x_i \oplus z_i$, $z_i \geq 1$.

Le mode OFB est une variante du mode CFB. La séquence de clé est engendrée par les chiffrements itérés d'un bloc initial IV de 64 bits : $z_0 = \text{IV}$. On calcule alors la séquence de clé $z_1z_2...$ par la formule : $z_i = e_K(z_{i-1})$, $i \geq 1$. La chaîne de texte clair est chiffrée en calculant $y_i = x_i \oplus z_i$, $i \geq 1$.
Ce mode d'utilisation présente cependant certains problèmes de sécurité et il est peu conseillé, sauf dans le cas où sa longueur est égale à celle de l'algorithme utilisé.

D'une manière générale, les quatres modes d'utilisation ont des qualités et des défauts. Dans les modes ECB et OFB, si l'on change 1 bloc de texte clair x_i (mal transmis par exemple), cela ne perturbe que le bloc de texte chiffré y_i. Ainsi, le mode OFB est souvent utilisé pour des transmissions par satellite. Cependant, dans certaines situations, cela peut être une propriété indésirable. Dans les modes CBC et CFB, par contre, la modification d'un bloc de texte clair x_i modifiera le bloc de texte chiffré y_i et tous les suivants.
Ces modes sont donc particulièrement adaptés au problèmes d'authentification et sont utilisés dans les MAC (*Message Authentification Code*). Un MAC est une section de message, ajoutée au texte clair (ou au texte chiffré), afin de garantir l'intégrité du message envoyé.

Pour le mode CBC, par exemple, on démarre avec le bloc initial IV dont tous les bits sont nuls. On construit ensuite le texte chiffré $y_1...y_n$ avec la clé K (en mode CBC), et le MAC sera tout simplement le bloc y_n. L'émetteur transmet le message $x_1...x_n$ avec le MAC y_n. Quand le

destinataire reçoit le message $x_1...x_n$, il construit $y_1...y_n$ avec la clé secrète et peut vérifier que y_n est identique au MAC reçu.

Il arrive aussi que deux interlocuteurs combinent sécurité et intégrité. L'auteur du message utilise alors une clé K_1 pour calculer un MAC sur $x_1...x_n$, puis définit x_{n+1} par ce MAC. Le message est alors chiffré par une seconde clé K_2, ce qui donne le message chiffré $y_1...y_ny_{n+1}$ (le calcul du MAC et le chiffrement peuvent bien sûr être inversés). Le destinataire le déchiffre alors (avec K_2) et vérifie que x_{n+1} est bien le MAC de $x_1...x_n$ (avec K_2).

1.2 IDEA (*International Data Encryption Algorithm*)

1.2.1 Historique

Plus récent que le DES, l'IDEA, contrairement aux autres algorithmes de codage, a été breveté par la société suisse Ascom. L'utilisation non commerciale de cet algorithme, essentiellement utilisé dans le système PGP (très répandu), est cependant permise sous réserve d'une autorisation d'Ascom.

1.2.2 Principe

L'IDEA opère sur des blocs de 64 bits et utilise généralement une clé de 128 bits qui sera transformée en 52 blocs de 16 bits.

Les algorithmes de cryptage et de décryptage sont les mêmes.

1.2.3 Description

Le bloc d'entrée de 64 bits est divisé en 4 blocs de 16 bits A ,B, C, et D qui devienent les blocs d'entrée de l'algorithme.

Les 8 premières sous-clés sont directement tirées de la clé principale. Les 8 clés suivantes sont obtenues de la même façon, après une permutation circulaire à gauche de 25 bit, et ainsi de suite.

Lors de chacun des 8 tours, trois opérations sont effectuées : une addition, un Xor (*ou*-exclusif) et une multiplication.

L'addition modulo $2^{16} + 1$ et le Xor se font de façon simple, mais la multiplication est elle légèrement plus compliquée. La multiplication par zéro donne toujours zéro et n'est pas inversible. La multiplication par un nombre, modulo n, n'est pas inversible non plus quand ce nombre n'est pas premier avec n. Pour que cette multiplication (modulo n) puisse être effectuée, il faut que le nombre soit inversible.

1.2.4 Détail de l'algorithme

Appelons K(1) à K(52) les 52 sous-clés.
Avant le premier tour, on multiplie A par K(1), on additionne B par K(2), et C par K(3) et on multiplie C par K(4).

Décrivons alors le premier tour :

Calcul de A Xor C (appelons E le résultat) et de B Xor D (appelons F le résultat).
On multiplie alors E par K(5) et on additionne cette nouvelle valeur de E à F.
On multiplie la nouvelle valeur de F obtenue par K(6). Cette nouvelle valeur de F est alors additionnée à E.
On change alors A et C en effectuant un Xor avec F et B et D en effectuant un Xor avec E.
Enfin, on permute B et C.

Pour chacun des 7 autres tours, il suffit d'effectuer le même travail, en prenant les sous-clés suivantes 6 par 6 jusqu'à la sous-clé 48. Il reste alors à multiplier A par K(49), additionner B à K(50) et C à K(51) et à multiplier D par K(52).

Le décryptage se passe alors de la même façon, à ceci près que les sous clés sont modifiées. En effet, en raison du placement de la permutation, pour chaque tour les clés de déchiffrement (KD) sont calculées de la manière suivantes :

KD(1) = 1/K(49)
KD(2) = -K(50)
KD(3) = -K(51)
KD(4) = 1/K(52)
KD(5) = K(47)
KD(6) = K(48)
Puis ,
KD(7) = 1/K(43)
KD(8) = -K(45)
KD(9) = -K(44)
KD(10) = 1/K(46)
et ainsi de suite...

1.2.5 Performances

Cet algoritme est considéré comme étant assez nettement supérieur au DES en terme de sécurité. Sa vitesse d'éxécution reste comparable avec le DES. Ses implementations hardware sont simplement légèrement plus rapides.

1.3 Blowfish

1.3.1 Principe

Créé en 1994, Blowfish est un algorithme de chiffrement par blocs basé sur le DES, mais avec des clés plus longues et plus d'aléas lors du codage : on n'utilise plus des tables fixées par la NSA, mais des tables à chaque fois différentes, déterminées par le mot-de-passe.

1.3.2 Description

Blowfish effectue un codage par blocs de 64 bits, et utilise une clé de longueur variable. L'algorithme est scindé en deux parties : une partie expansion de la clé et une partie encodage des données. La partie expansion de la clé consiste à convertir la clé de départ (maximum 448 bits) en plusieurs sous-clé totalisant 4168 bytes.

Le chiffrement des données s'effectue au cours de 16 itérations. Chacune d'elle est constituée d'une permutation dépendante de la clé, et d'une substitution dépendante de la clé et des données. Toutes les opérations sont des Xor et des additions sur des mots de 32 bits.

1.3.3 Détail de l'algorithme

Blowfish utilise un grand nombre de sous-clés. Celles-ci doivent être calculées avant d'effectuer le codage ou le décodage des données.

Le champ P comprend 18 sous-clés de 32 bits : P_1, P_2, ... , P_{18}. 4 boîtes S de 32 bits avec 256 entrées chacune sont également utilisées :

$$S_{1,0}\,;\, S_{1,1}\,;\, \ldots\,;\, S_{1,255}$$
$$S_{2,0}\,;\, S_{2,1}\,;\, \ldots\,;\, S_{2,255}$$
$$S_{3,0}\,;\, S_{3,1}\,;\, \ldots\,;\, S_{3,255}$$
$$S_{4,0}\,;\, S_{4,1}\,;\, \ldots\,;\, S_{4,255}$$

L'algorithme de codage Blowfish comporte 16 itérations. L'entrée est un bloc de 64 bits, appelé x. x est tout d'abord divisé en 2 moitiés de 32 bits : x_L, x_R. Chaque tour de chiffrement i se fera ensuite de la façon suivante :
On calcule tout d'abord $x_L = x_L \oplus P_i$ et $x_R = F(x_L) \oplus x_R$, avant d'échanger x_L et x_R.

Après le dernier tour, on échange x_L et x_R (ce qui annule l'échange précédent), et on effectue les deux calculs suivants : $x_R = x_R \oplus P_{17}$ et $x_L = x_L \oplus P_{18}$. On recombine alors x_R et x_L pour obtenir ainsi le message chiffré.

La fonction F utilisée dans cet algorithme divise x_L en 4 quarts de 8 bits chacun : a, b, c et d, puis calcule :

$$F(x_L) = (S_{1,a} + S_{2,b}\ mod\ 232) \oplus (S_{3,b} + S_{4,d}\ mod\ 232)$$

Le décodage est similaire à l'encodage à ceci près que P_1, P_2, ... , P_{18} sont utilisés dans l'ordre inverse.

Génération des sous-clés :

La méthode de construction des sous-clés est la suivante :

Initialiser, premièrement, le champ P, puis les 4 boîtes S dans l'ordre avec une chaîne de caractères fixe. Cette chaîne est constituée de chiffres hexadécimaux de p_i.

Additionner P_1 avec le premier bloc de 32 bits de la clé, modulo 2, et P_2 avec le second bloc, et ainsi de suite pour tous les bits de la clé (jusqu'à P_{14}).

Encoder la chaîne de 0 avec l'algorithme Blowfish, utilisant les sous-clés décrites dans les deux étapes précédentes.

Remplacer P_1 et P_2 avec les sorties de l'étape précédente.

Encoder la sortie de l'étape 3 utilisant l'algorithme Blowfish avec les sous-clés modifiées.

Remplacer P_3 et P_4 avec les sorties de l'étape précédente.

Continuer le processus, en remplaçant tous les P, et ensuite les 4 boîtes S dans l'ordre, avec les sorties, toujours changeantes, de l'algorithme Blowfish.

Au total, 521 itérations seront nécessaires pour générer toutes les sous-clés. En pratique, les applications peuvent stocker les sous-clés au lieu d'exécuter ce processus de dérivation plusieurs fois.

1.3.4 Performances

Blowfish est un algorithme très performant en terme de sécurité en apparence, et très rapide. Une étude, lors d'un concourt organisé par le Docteur Dobbs Journal, a pourtant montré que Blowfish contenait quelques failles. En pratique, ces failles ne sont cependant pas exploitables. Blowfish est encore relativement neuf et n'est pas très répandu. On manque donc d'information pour dire si cet algorithme est véritablement performant.

1.4 RC4

1.4.1 Historique

RC4 est un algorithme de chiffrement en continu à clé de longueur variable développé en 1987 par Ron Rivest pour RSA.

Il est longtemps resté secret avant d'être publié, et est maintenant beaucoup utilisé, en particulier dans le protocole SSL.

1.4.2 Description

La structure du RC4 se compose de 2 parties distinctes.

La première, *key scheduling algorithm*, génère une table d'état S à partir des données secrètes, à savoir 64 bits (40 bits de clé secrète et 24 bits d'IV) ou 128 bits (104 bits de clé secrète et 24

bits d'IV), par exemple.

La deuxième partie de l'algorithme RC4 est le générateur de données en sortie, qui utilise la table S et 2 compteurs, i et j.

L'algorithme appliqué est le suivant :
$i = i + 1 \ mod \ 256$
$j = j + S_i \ mod \ 256$
permutation de S_i et S_j
$S_t = S_i + S_j \ mod \ 256$
$K = S_t$

Ces données en sortie forment alors une séquence pseudo-aléatoire.

1.4.3 Performances

Le chiffrement RC4 est extrêmement rapide, sûrement le plus rapide des chiffrements utilisés à l'heure actuelle même.

Cependant, il comporte quelques failles de sécurité qui sont exploitables de façon plus efficace qu'une recherche exhaustive de clé.

Fluhrer, Mantin et Shamir ont en effet explicité 2 faiblesses dans la spécification de l'algorithme RC4.

La première repose sur le fait qu'il existe de larges ensembles de clés dites faibles, c'est-à-dire des clés dont quelques bits seulement suffisent à déterminer de nombreux bits dans la table d'état S (avec une forte probabilité), ce qui affecte directement les données produites en sortie ; c'est l'attaque nommée «*invariance weakness*».

La deuxième attaque connue est la «*known IV attack* ». Elle nécessite, comme son nom l'indique, la connaissance de l'IV (*Initiale Value*), ce qui peut être le cas lorsqu'il circule en clair sur le réseau, ainsi que la connaissance du premier octet du message M (à retrouver). Dans un certain nombre de cas (les *cas résolus*, suivant l'expression de Fluhrer, Mantin et Shamir), la connaissance de ces 2 éléments permet de déduire des informations sur la clé K.

Selon les tois chercheurs, ces 2 attaques sont applicables et peuvent permettre une récupération complète de la clé avec une efficacité bien supérieure à l'attaque par recherche exhaustive.

Cependant, il faut savoir que ces attaques ne sont pas réalisables dans tous les cas. Ainsi l'utilisation du chiffrement RC4 dans le protocole SSL, par exemple, est faite de façon à éviter ses deux types d'attaques.

1.5 AES (*Advanced Encryption Standard*)

1.5.1 Historique

En 1997, le NIST (*National Institute of Standards and Technology*) américain lance un appel d'offre afin de trouver un remplaçant au DES. 15 algorithmes sont alors étudiés, et en fonction des différents critères, fin 2000, l'algorithme belge Rijndaël (à prononcer raindal), proposé par Joan Daemen et Vincent Rijmen, est retenu. Il faut savoir que chacun des modèles a été testé sur plusieurs types de surface et Rijndaël n'a été le premier sur aucune d'elle, mais il a montré qu'il gardait à chaque fois des performances très intéressantes. Il a donc été choisi autant pour son adaptabilité que pour ses performances.

Du fait de son origine, l'AES est un standard. Il est donc libre d'utilisation, sans restriction d'usage ni brevet.

1.5.2 Principe

L'AES opère sur des blocs rectangulaires de 4 lignes et Nc colonnes, dont chaque terme $x_{i,j}$ (appelé octet ou byte) est composé de 8 bits ($b = b_7b_6b_5b_4b_3b_2b_1b_0$), et peut être représentés algébriquement sous forme de ploynômes de degrés ≤ 7 ($b = b_7X^7 + b_6X^6 + b_5X^5 + b_4X^4 + b_3X^3 + b_2X^2 + b_1X + b_0$), à coefficients dans F_2 (0 ou 1). La clé peut être d'une longueur de 128, 156, ou 256 bits, de même pour le message clair et le message chiffré.

1.5.3 Description

Tout d'abord la clé est transformée en une clé étendue rectangulaire de Nc colonnes et $Nc(Nn+1)$ lignes, où Nn est le nombre de tours de l'algorithme (voir ci-dessous). Le chiffrement et le déchiffrement s'effectuent de la même manière et sont composés de Nn tours comprenant chacun 4 opérations.

Le nombre de tour (Nn) dépend de la longueur de la clé (Nk) et de la longueur du message clair (Nm).

Tableau du nombre de tours :

Nn	$Nm = 4$ (128 *bits*)	$Nm = 6$ (192 *bits*)	$Nm = 8$ (256 *bits*)
$Nk = 4$ (128 *bits*)	10	12	14
$Nk = 6$ (192 *bits*)	12	12	14
$Nk = 8$ (258 *bits*)	14	14	14

1.5.4 Détail de l'algorithme

Chaque tour est composé de 4 opérations, nommées *SubBytes*, *ShiftRows*, *MixColumns*, et *XorRoundKey*. Il faut savoir, en termes mathématiques, que toutes les opérations sont effectuées dans le groupe $GF(2^8)$. $GF(2^8)$ est isomorphe à $GF(2)[X]/m(X)$, où $m(X) = X^8 + X^4 + X^3 + X + 1$ est un polynôme irréductible dans $GF(2^8)$. L'addition considérée sera l'addition des polynômes dans $F_2[X]$ (addition des coefficients des mêmes monomes modulo 2). La multiplication sera la

multiplication des polynômes avec réduction modulo le polynôme $m(X)$. La multiplication dans $GF(2^8)$, contrairement à l'addition ne correspond à aucune opération simple au niveau des octets.

Décrivons le principe général de chaque opérations de ce chiffrement.

SubBytes :

La procédure SubBytes effectue une *inversion* dans le groupe $GF(2^8)$, suivie d'une *application affine*, définie par :
Pour $0 \leq i < 8$,

$$b_i = b_i \oplus b_{(i+4) \ mod \ 8} \oplus b_{(i+5) \ mod \ 8} \oplus b_{(i+6) \ mod \ 8} \oplus b_{(i+7) \ mod \ 8} \oplus c_i$$

Soit, sous forme matricielle :

$$\begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Cette transformation est effectuée pour chaque octet b du bloc ($b = b_7b_6b_5b_4b_3b_2b_1b_0$). Il est à noter qu'en terme de programmation, cette transformation peut simplement être remplacée par un tableau (*Sbox*) à deux entrées de 4 bits chacune (le "demi-octet" de gauche en entrée horizontale et le "demi-octet" de droite en entrée verticale) donnant directement l'octet résultat

ShiftRows :

Cette opération effectue une simple *permutation circulaire* à gauche par ligne de chacun des octets du bloc. La première ligne sera inchangée. Dans la seconde, chaque octet sera décalé d'un cran à gauche (le premier prenant la place du dernier). Dans la troisième, chaque octet sera décalé de 2 crans à gauche, et dans la quatrième, chaque octet sera décalé de 3 crans à gauche.

MixColumns :

Cette procédure effectue un *"mélange"* à l'intérieur de chaque *colonne*. Chaque octet de la colonne (sous forme polynômiale) est multiplié par le polynôme $a(X) = 3X^3 + X^2 + X + 2$, modulo $X^4 + 1$. Les coefficients sont ensuite réduits modulo 2.
Nous aurons, sous forme matricielle :

$$\begin{bmatrix} x_{3,j} \\ x_{2,j} \\ x_{1,j} \\ x_{0,j} \end{bmatrix} = \begin{bmatrix} 02 & 01 & 01 & 03 \\ 03 & 02 & 01 & 01 \\ 01 & 03 & 02 & 01 \\ 01 & 01 & 03 & 02 \end{bmatrix} \cdot \begin{bmatrix} x_{3,j} \\ x_{2,j} \\ x_{1,j} \\ x_{0,j} \end{bmatrix} \quad , \text{pour } 0 \leq j \leq Nc$$

Le terme $x_{i,j}$ correspond à l'octet du bloc situé sur la i-ième ligne et la j-ième colonne.

XorRoundKey :

L'opération XorRoundKey effectue simplement l'*addition* modulo 2 avec la section de la clé étendue correspondant au tour r durant lequel s'effectue la transformation.

La clé étendue est de la forme : $k_0k_1k_2k_3k_4...$ où chaque k_c est une matrice 4 lignes et 1 colonne. Pour le XorRoundKey, on prendra Nc colonnes k_c différentes à chaque tour.

Calcul de la clé étendue :

Les Nk premières colonnes de la clé étendue correspondent à la clé de départ, non modifiée. Les autres colonnes sont calculées en effectuant un Xor de la colonne précédente et de celle située Nk positions avant. Notons $Kexp$ la clé étendue, nous avons donc : $Kexp_i = Kexp_{i-1} \oplus Kexp_{i-Nk}$. Pour les colonnes de la clé étendue qui sont d'indice multiple de Nk, une transformation est appliquée à $Kexp_{i-1}$ avant le Xor. Il s'agit d'une permutation cyclique d'un cran vers la gauche, suivie d'un SubBytes pour chacun des octets de la colonne et d'un Xor avec un vecteur dépendant du tour ($Rcon$).

Ce calcul de clé étendue se fait une fois pour toute, avant de commencer le chiffrement.

Il est à noter qu'avant d'effectuer le premier tour, un XorRoundKey est effectué avec le message clair et la clé de chiffrement.
De plus, le dernier tour ne contient pas l'opération MixColumns.

Lors du déchiffrement, le principe reste le même, même si les 4 opérations sont effectuées dans l'ordre inverse. La clé étendue de déchiffrement est la même que celle de chiffrement, et le XorRoundKey est lui aussi identique. Par contre, les opérations SubBytes, ShiftRows, MixColumns sont remplacées par leurs opérations inverses.

1.5.5 Exemple simple

Dans cet exemple, tous les demi-octets seront écrits sous forme hexadécimale :

0000 : 0	0100 : 4	1000 : 8	1100 : c
0001 : 1	0101 : 5	1001 : 9	1101 : d
0010 : 2	0110 : 6	1010 : a	1110 : e
0011 : 3	0111 : 7	1011 : b	1111 : f

Nous ne présenterons ici que les résultats des calculs pour un seul tour de chiffrement. Certains calculs sont facilement vérifiables à la main (ShiftRows et XorRoundKey), mais les autres (SubBytes et ShiftRows) nécessitent l'emploi d'un ordinateur, (ou de pas mal de temps et de patience).

Soit le bloc à chiffrer :

19 a0 9a e9
3d f4 c6 f8
e3 e2 8d 48
be 2b 2a 08

La première étape consiste (comme nous l'avons vu) à effectuer un *SubBytes*. Ce qui nous donne :

d4 e0 b8 1e
27 bf b4 41
11 98 5d 52
ae f1 e5 30

Nous effectuons alors un *ShiftRows*, pour obtenir le bloc :

d4 e0 b8 1e
bf b4 41 27
5d 52 11 98
30 ae f1 e5

L'étape suivante est le *MixColumns*. Le bloc obtenu après cette opération sera :

04 e0 48 28
66 cb f8 06
81 19 d3 26
e5 9a 7a 4c

Supposons que la section de la *clé étendue* correspondant à ce tour soit de la forme :

a0 88 23 2a
fa 54 a3 6c
fe 2c 39 76
17 b1 39 05

L'addition *XorRoundKey* du bloc précédent avec cette section de la clé étendue nous donnera alors le résultat final du chiffrement de ce tour :

a4 68 6b 02
9c 9f 5b 6a
7f 35 ea 50
f2 2b 43 49

Détaillons ce calcul (qui est effectué terme à terme à l'intérieur du bloc), pour le terme x_{22} par exemple :
On additionne ici cb et 54 modulo 2. Soit, écrit sous forme de bits :

$$\begin{array}{r} (\text{cb})\ 1100\ 1011 \\ \oplus\ (54)\ 0101\ 0100 \\ \hline (9\text{f})\ 1001\ 1111 \end{array}$$

Le résultat obtenu (9f) est bien le terme x_{22} du nouveau bloc.

Cela nous donne donc la description complète (au moins pour les résultats) d'un tour de chiffrement d'AES. Les calculs sont effectués $Nn - 1$ fois (ici $Nc = 4$ et $Nk = 4$, Nn sera donc égal à 10 tours). Le dernier tour sera effectué de la même façon, à ceci près que le *MixColumns* ne sera pas effectué.

1.5.6 Modes d'utilisation

AES est le successeur direct de DES. C'est lui aussi un chiffrement par blocs, même s'il permet de chiffrer des blocs de taille nettement supérieure. Les 4 modes d'utilisation du DES (ECB, CFB, CBC et OFB) peuvent donc directement lui être appliqués.

1.5.7 Performances

L'AES a été choisi pour être totalement sûr et opérationnel sur tout type d'environnement. Il répond effectivement à ces obligations, puisqu'une recherche exhaustive de la clé n'est absolument pas envisageable en un temps limité (on parle de près de 149 milliards d'années) et aucune attaque ne lui est connue à ce jour.
Il est très efficace en terme de rapidité (nettement plus que le DES).
Ses besoins en ressources mémoires sont également très faibles.
Il est très flexible d'implémentation. Cela induit une grande variété de platformes et d'applications.
Il est possible de l'implémenter aussi bien sous forme logicielle que matérielle (cablé).
Enfin, nous pouvons ajouter que l'algorithme de l'AES est relativement simple.

Ce sont, entre autre, ces critères qui ont poussé le monde de la troisième génération de mobiles à adopter cet algorithme pour son schéma d'authentification "Millenage".

NB : Il existe d'autres algorithmes tout aussi récents que Rijndael qui ont été mis en place suite à l'appel d'offre du NIST. Parmis eux : Twofish, SERPENT, RC6 et MARS qui étaient les 4 autres "finalistes". Nous ne les présenterons cependant pas ici, et ce pour deux raisons : si l'algorithme belge a été choisi, c'est qu'il était plus approprié et/ou plus performant que ces-derniers ; et du fait qu'il ait été certifié par le NIST, l'AES est parfaitement libre de droit et d'utilisation, ce qui n'est pas le cas des autres algorithmes.

2 Cryptographie Asymétrique

Le principe d'un code asymétrique (aussi appelé à clé publique) est que, contrairement au code symétrique, les deux interlocuteurs ne partagent pas la même clé. En effet, la personne qui veut envoyer un message utilise la clé publique de son correspondant. Celui-ci déchiffrera alors ce message à partir de sa clé privée que lui seul connait.

On voit ici que contrairement à un codage symétrique, le chiffrement et le déchiffrement se font par des opérations complètement différentes.

En réalité ce type de chiffrement se base sur le fait que certaines fonctions mathématiques, même si l'on peut facilement montrer qu'elles sont inversibles sont très dures à inverser. En fait, elle ne sont pas inversibles en un temps limité avec les connaissances actuelles.

2.1 RSA (*Rivest-Shamir-Adleman*)

2.1.1 Historique

R.S.A. signifie Rivest-Shamir-Adleman, en l'honneur de ses inventeurs : Ron Rivest, Adi Shamir et Leonard Adleman qui l'ont créé en 1977. Le brevet de cet algorithme appartenait jusqu'au 6 Septembre 2000 à la société américaine RSA Data Security, qui fait maintenant partie de Security Dynamics et aux Public Key Partners, (PKP à Sunnyvale, Californie, états-Unis).

Tout le principe de RSA repose sur le fait qu'il est très difficile et donc très long de décomposer un très grand nombre en deux grands facteurs premiers, sauf cas particuliers.

2.1.2 Principe

Le codage RSA repose sur la factorisation d'un entier. En effet, il est très compliqué de factoriser un entier suffisamment grand n'ayant pas de petits facteurs premiers. Soient p et q deux nombres premiers, il est facile de calculer leur produit $n = p.q$. Toutefois retrouver p et q , en ne connaissant que n (i.e. de factoriser n) est loin d'être simple.

2.1.3 Description

Soit A l'émetteur du message à chiffrer et B le récepteur.

Création des clés (par B) :

Choisir p et q 2 entiers premiers distincts suffisamment grands et suffisamment éloignés.

Poser $n = p.q$; on a alors $\phi(n) = (p-1)(q-1)$ où ϕ est la fonction indicatrice d'Euler (nombre d'entiers premiers avec n et inférieurs à n).

Choisir $1 < e < \phi(n)$ tel que l'on ait $pgcd(e, \phi(n)) = 1$. (i.e. e et $\phi(n)$ premiers entre eux).

Calculer d tel que $e.d \equiv 1 \ (mod\ \phi(n))$ (i.e. $d \equiv e^{-1} \ (mod\ \phi(n))$).

Nous avons alors :
Clé publique de B : (n, e).
Clé privée de B : d.

Chiffrement (par A pour B) :
Récupérer la clé publique de B : (n, e).
Couper le message en entier m, tel que $0 \leq m \leq n-1$.
Calculer $c \equiv m^e \ (mod\ n)$.
Envoyer c.

Déchiffrement (par B) :
B calcule alors $m \equiv c^d \ (mod\ n)$ et obtient le message clair.

Explications mathématiques :
Le texte chiffré (que reçoit B) est $c \equiv m^e \ (mod\ n)$.
B calcule alors :

$$c^d \equiv (m^e)^d \ (mod\ n)$$
$$c^d \equiv m^{e.d} \ (mod\ n)$$

Or $e.d \equiv 1 \ (mod\ \phi(n))$
Ce qui nous donne bien (d'après la généralisation du petit théorème de Fermat) :

$$c^d \equiv m \ (mod\ n)$$

2.1.4 Exemple simple

Le principe du codage par factorisation restant le même quelle que soit la taille des nombres, nous allons nous contenter d'un exemple avec de petits nombres. Dans la pratique, on utilise des entiers de plus d'une centaine de chiffres.

Soit $n = 187$ et $e = 7$ la clé de B. n est un nombre composé par le produit des deux facteurs premiers $p = 17$ et $q = 11$ ($n = p.q = 17*11 = 187$). L'indicatrice d'Euler est $\phi(n) = (p-1)(q-1) = 16 * 10 = 160$. e et n sont tels que e et $\phi(n)$ soient premiers entre eux.
B doit alors déterminer sa clé privée (qui ne sera jamais divulguée). Il doit chercher le plus petit entier d tel que $d * e \equiv 1 \ (mod\ \phi(n))$.
Travaillons par exemple à partir de l'identité de Bézout :

$$160 = 7 * 22 + 6$$
$$7 = 6 * 1 + 1$$

Ce qui nous donne :

$$1 = 7 - 1 * 6 \ (mod\ 160)$$
$$1 = 7 - 1 * (160 - 7 * 22) \ (mod\ 160)$$
$$1 = 7 * 23 \ (mod\ 160)$$

Nous avons donc $d = 23$.

B peut alors envoyer sa clé publique ($n = 187$,$e = 7$) à A (et à ses autres interlocuteurs) et garde sa clé privée $d = 23$.

A veut envoyer le message "arrive le 17" qui doit absolument rester secret. Il va donc le coder avec la clé publique de B. Pour commencer il convertit son texte en une série de chiffres, en prenant

par exemple : $a = 01$,$b = 02$,...,$espace = 00$,$0 = 30$,$1 = 31$,$2 = 32$,...
Cela donnera :

$$011818092205001205003137$$

Il regroupe ce nombre en tranches ayant moins de chiffres que le $n = 187$ de la clé. Ce seront donc des tranches de 2 chiffres :

$$01\ 18\ 18\ 09\ 22\ 05\ 00\ 12\ 05\ 00\ 31\ 37$$

Chaque terme sera considéré comme un message m à chiffrer avec le calcul : $c \equiv m^7\ (mod\, 187)$.

Le résultat sera :

$$1\ 171\ 171\ 70\ 44\ 146\ 0\ 177\ 146\ 0\ 177\ 146\ 0\ 47\ 125\ 181$$

Après avoir ajouter des 0 si nécessaire, le message que A enverra à B sera :

$$001171171070044146000177146000177146000047125181$$

B possède la clé $d = 23$ et va donc faire le calcul de déchiffrement pour chaque bloc de 3 chiffres :

$$m \equiv c^{23}\ (mod\ 187)$$

Il va ainsi retrouver 01 pour 001 (trivial), 18 pour 171 (car $171^{23}\ (mod\ 187) = 18$), et ainsi de suite, pour chacun des termes. En prenant le tableau de correspondance, il retrouve bien le message d'origine : "arrive le 17".

Un observateur qui voudrait cryptanalyser ce message ne peut le faire sans posséder la clé secrète $d = 23$. Pour pouvoir déchiffrer ce message il lui faudrait alors retrouver $\phi(n)$ (pour pouvoir obtenir d), et donc décomposer n en facteurs premiers (ce qui est peut-être facile avec 187, mais beaucoup plus difficile avec un nombre de 150 à 200 chiffres).

On s'aperçoit facilement avec cet exemple que si l'on garde l'algorithme tel quel (sans y ajouter l'addition avec une clé, une permutation, ou une autre opération), chaque terme identique sera chiffré de la même manière. Une cryptanalyse simple en terme de pourcentage d'apparition de lettres peut alors être effectuée. Cela est d'autant plus flagrant pour les caractères d'espacement. En effet, avec l'exemple précédent, on retrouverait tout ces caractères remplacés par des 0, il serait donc facile de reconnaître, dès le premier coups d'oeil, l'allure du message clair. Cependant, ce problème est uniquement dû au fait que l'on ait pris la même longueur de "tranches à chiffrer" que la longueur de chaque lettre (2 et 2). Ce problème ne se pose heureusement pas pour un cas général.

Notez enfin que ces calculs ont été effectués avec des entiers sous forme décimale, mais peuvent très bien être retranscrits sous forme binaire avant d'être envoyés, et inversement, retraduits avant le déchiffrage.

2.1.5 Variante : le chiffrement El Gamal

Un autre système, dérivé du RSA, est le système El-Gamal (1982) qui propose un système un peu plus complexe, mais qui se base toujours sur le fait que $(g^x)^y = g^{xy} = (g^y)^x$. A choisit un nombre x secret et calcule $X \equiv g^x\ (mod\ n)$ qu'il donne à B ; de même B choisit un nombre y

secret et calcule $Y \equiv g^y \ (mod\ n)$, avant de l'envoyer à A. A possédera une clé privée x, et B une clé privée y. La clé publique de B sera ici (n,g).

Pour chiffrer son message, A devra prendre Y, que lui aura fourni B et enverra $c \equiv m * Y^x \ (mod\ n)$, ainsi que X.
B calculera alors :

$$m \equiv \frac{c}{X^y} \ (mod\ n)$$
$$m \equiv \frac{m*(g^y)^x}{(g^x)^y} \ (mod\ n)$$

On remarque bien, après simplification, que l'on retrouve le message clair m.

Tout observateur ne peut récupérer que X et/ou Y, et même en connaissant g et n, il ne peut retrouver x ou y, car le logarithme discret $x \equiv log_g(X)$ est très difficile à obtenir et connaître g^x et g^y ne permet pas de trouver g^{xy}.

2.1.6 Performances

La cryptographie à clé publique possède un avantage majeur sur la cryptographie à clé secrète. En effet, si n utilisateurs souhaitent communiquer par le biais d'un cryptosystème à clé secrète, chacun d'eux doit disposer d'une clé différente par personne du groupe. Il faut donc pouvoir gérer en tout $n(n-1)$ clés. Sachant que n peut être de l'ordre de plusieurs milliers, il faut gérer des fichiers de plusieurs millions de clés. De plus, ajouter un utilisateur au groupe n'est pas une mince affaire, puisqu'il faut alors engendrer n clés pour que le nouvel utilisateur puisse communiquer avec les autres membres du groupe, puis distribuer les nouvelles clés à tout le groupe. En revanche, dans le cas d'un cryptosystème asymétrique, on stocke les n clés publiques des utilisateurs dans un annuaire. Pour rajouter un utilisateur, il suffit qu'il mette sa clé publique dans l'annuaire.

Le codage RSA est donc très intéressant de par le fait qu'il soit à clé publique. De plus, il donne parfaitement satisfaction en terme de sécurité. Malheureusement, il reste beaucoup plus lent qu'un code symétrique. De plus, il ne permet pas de coder des messages trop longs sans être obligé de les couper, car le calcul de déchiffrement risquerait d'être trop compliqué. Il est donc le plus souvent utilisé pour chiffrer un nombre aléatoire qui servira de clé dans un message à clé privée. Le temps de cryptage et de décryptage de la clé restent donc abordables, puisque la clé n'est pas trop longue. Le cryptage, ensuite, par un code symétrique permet de garder cette rapidité.

Il est également à noter que pour le moment, la sécurité du cryptage RSA repose sur la puissance des ordinateurs et sur les connaissances actuelles en arithmétique. Tout progrès dans un domaine comme dans l'autre (qui sont en constante évolution) peut obliger à allonger la clé, ou à chercher une autre fonction qui soit facilement appliquable dans un sens, et quasiment impraticable dans l'autre.

2.2 Systèmes de signature

Un codage asymétrique, quel qu'il soit, peut être très avantageux du fait que la clé privée ne soit détenue que par le receveur du message. Cela assure notamment que la clé privée ne puisse être divulguée, car elle n'est connue de personne d'autre que son utilisateur. Cependant, un problème apparaît rapidement : étant donné que la clé de chiffrement est publique, comment authentifier l'expéditeur. C'est là qu'intervient le système de signature. En effet, si la personne qui envoie le message y ajoute une signature qui lui est propre, il n'y a plus aucun doute sur son identité. Dans la réalité, la plupart du temps, l'expéditeur chiffrera un nombre particulier (issu ou non du message) d'une façon qui lui est propre, afin d'assurer le destinataire de l'intégrité du message qu'il reçoit.

2.2.1 Fonction de hachage

Dans le cas du système précédent, par exemple, pour signer le message envoyé, on procède généralement de la façon suivante : l'expéditeur applique au message clair une fonction mathématique (appelée fonction de hachage) qui produit un "résumé" (code haché) du message. Le résumé obtenu est propre à chaque message, à l'image d'une empreinte digitale. Le code haché peut ensuite être chiffré à l'aide de la clé privée de l'expéditeur et être annexé au message envoyé. C'est ce code qui constitue la signature numérique. Le destinataire du message peut ensuite vérifier que l'expéditeur est bien le bon en déchiffrant la signature numérique, au moyen de la clé publique correspondante, pour obtenir le code haché. Le destinataire applique ensuite la même fonction de hachage au message reçu ; si les deux codes sont identiques, l'expéditeur du message est bien le bon (*non-répudiation*) et le message n'a pas été altéré (*intégrité*).

Cette fonction de hachage doit être "*anti-collision*", c'est-à-dire qu'il doit être quasiment impossible de trouver un $m' \neq m$, tel que $h(m') = h(m)$. Il faut aussi qu'elle soit relativement facile à calculer.

Il existe plusieurs algorithmes proposant des fonctions de hachage. Les principaux sont :

MD4 et MD5 (*Message Digest*) qui furent développés par Ron Rivest. MD5 produit des hachés de 128 bits en travaillant les données originales par blocs de 512 bits.

SHA-1 (*Secure Hash Algorithm* 1) ; comme MD5, il est basé sur MD4. Il fonctionne également à partir de blocs de 512 bits de données et produit par contre des condensés de 160 bits en sortie. Il nécessite donc plus de ressources que MD5.

SHA-2 (*Secure Hash Algorithm* 2) qui a été publié récemment et est destiné à remplacer SHA-1. Les différences principales résident dans les tailles de hachés possibles : 256, 384 ou 512 bits. Il sera vraisemblablement bientôt la nouvelle référence en terme de fonction de hachage.

RIPEMD-160 (*Ripe Message Digest*) est la dernière version de l'algorithme RIPEMD. La version précédente produisait des condensés de 128 bits mais présentait des failles de sécurité importantes. La version actuelle reste pour l'instant sûre ; elle produit comme son nom l'indique des condensés de 160 bits. Un dernier point la concernant est sa relative gourmandise en termes de ressources en comparaison avec SHA-1 qui est son principal concurrent.

On peut se demander pourquoi il existe plusieurs tailles de condensés ou encore pourquoi celle-ci est fixe. Il faut garder à l'esprit le but ultime d'un haché qui est d'être le plus court possible, tout en gardant ses propriétés. Or, cela nous amène tout naturellement au problème des collisions (évoqué

précédemment), également connu sous la dénomination de *théorème* ou *paradoxe des anniversaires*.

Prenons donc notre haché H, qui présente une longueur de n bits. Nous pouvons d'ores et déjà déduire qu'il n'existe que 2^n hachés de ce type possibles (puisque chaque bit n'a que 2 valeurs possibles, 0 ou 1). Le problème réside dans le fait que de l'autre côté, nous pouvons avoir une infinité de textes initiaux (dont la taille, elle, n'est pas fixée). Nous risquons donc, un jour ou l'autre, de produire un haché qui pourrait correspondre à un autre texte original (ou à plusieurs) : c'est la perte de la propriété principale d'un condensé, qui est l'unicité. On dit alors que nous avons trouvé une *collision*.

Le *théorème des anniversaires* prouve qu'il faut $2^{\frac{n}{2}}$ essais pour trouver une collision au hasard. C'est le chiffre qui sera utilisé pour évaluer la force d'une fonction de hachage. Pourtant, il ne faut pas négliger le fait que la collision citée précédemment a été obtenue au hasard, ce qui n'est pas exploitable par une personne malveillante. En effet, le but serait de trouver un message significatif et bien formé conduisant au même haché, ce qui augmente considérablement les essais et calculs nécessaires (et le rend quasiment impossible). Cela suffit cependant en théorie pour briser la propriété d'unicité du condensé.

D'un point de vue pratique, il est généralement accepté que 256 calculs représentent un défit réalisable. Comme exemple, les clés DES de 56 bits sont considérées comme réellement faibles et "crackables". En conséquence, avec $\frac{n}{2} = 56$ (et donc $n = 112$), le théorême des anniversaires nous indique que les hachés de 112 bits sont faibles et donc insuffisants à l'heure actuelle. De la même manière, les hachés de 128 bits ($\frac{n}{2} = 64$) ne représentent plus véritablement une sécurité à moyen terme. C'est pour cela que la norme actuelle est à 160 bits ($\frac{n}{2} = 80$) voir même plus, dans le cas de SHA-2 par exemple.

2.2.2 Signature El Gamal-Schnorr

L'algorithme de Claus Schnorr permet de réaliser des signatures électroniques. Cet algorithme est basé sur le problème du logarithme dans $GF(p)$ et utilise, comme son nom l'indique, le système de chiffrement d'El Gamal. Un de ses avantages est qu'il n'existe pas de trappe dont la connaissance permette de casser tout le système.

Principe :

Ce type d'algorithmes numériques de signature se base sur le principe suivant : la personne qui connaît x construit un ensemble de deux ou trois nombres qui ont un rapport entre eux tels qu'ils peuvent seulement être produits avec la connaissance de x, mais toute personne ne connaissant que g^x peut vérifier l'auteur du message.

On suppose que l'émetteur A a une paire de clés (x,X), où X est publique et x est secrète, tel que $X \equiv g^x \ (mod\ p)$. Il choisit un nombre r au hasard et envoie $t \equiv g^r \ (mod\ p)$. Le destinataire B renvoie un nombre e aléatoire obtenu à partir d'une fonction de hachage. A calcule alors $\sigma \equiv r + x.e \ (mod\ (p-1))$ et l'envoie à B qui peut vérifier que $g^\sigma \equiv t.X^e$.
En effet, si tout se passe bien il vérifiera :

$$g^\sigma \equiv g^r.(g^x)^e \equiv g^{r+x.e}.$$

2.2.3 Système DSA (*Digital Signature Algorithm*)

Le système DSA est une variante du système El Gamal-Schnorr. Il a été créé et certifié par le NIST, et a été spécifié comme algorithme de signature digitale utilisant SHA-1 comme fonction de hachage. DSA sert uniquement comme système de signature et ne permet pas de chiffrer un message (ou du moins n'est pas optimiser pour).

La clé secrète opère sur le message généré par SHA-1. La longueur de la clé varie entre 512 et 1024 bits. La création de la clé se fait à la même vitesse que pour RSA, mais la vérification est beaucoup plus lente.

Principe :

Les exposants, e et d sont tous deux liés entre eux. Ainsi, un texte chiffré avec l'exposant public d peut avoir été écrit par n'importe qui, mais seul celui qui possède la clé e peut le lire. Inversement, un texte chiffré avec l'exposant privé e peut être lu par n'importe qui, mais seulement l'individu connaissant e peut l'avoir écrit. Exécuter une signature numérique utilisant l'algorithme principal de Diffie-Hellman comme base exige une approche légèrement différente.

Description :

Suppossons p premier tel que $p-1 = 2.q$; quand q est aussi premier, on a $q-1 = 2.r$, avec r premier. Cela signifie que q peut également servir de modulo approprié pour la création de la clé Diffie-Helman, bien qu'il soit plus petit que p.

Une personne voulant signer un document m a donc une clé secrète x, pour laquelle la clé publique sera $g^x \ (mod\ q)$ (plutôt que $mod\ p$). En outre, une clé y est générée pour laquelle $g^y \ (mod\ p)$, cette fois sera la clé publique correspondante.

g est choisit de sorte que $g^q \equiv 1 \ (mod\ p)$. Cela assure ainsi que g génère un panel suffisamment large de nombres entre 1 et $p-1$. C'est cela qui servira de base pour appliquer Diffie-Helman modulo p.

Y est alors calculé de cette façon : $Y \equiv g^y \ (mod\ p) \ (mod\ q)$. Le message et la signature sont deux nombres modulo q ; la signature s est calculée comme une fonction de X.

Divers type de signatures sont possibles. Tous ont en commun trois nombres a, b et c qui sont définis par les termes s, Y et m. Ces trois nombres satisfont l'équation :

$$a.x + b.y \equiv c \ (mod\ q)$$

Cela montre pourquoi il est extrèmement important qu'un y différent soit choisi pour chaque signature. a, b, et c sont tous publics ; ainsi, l'on a deux ensembles de a, de b, et de c pour le même x et y, on peut retrouver les clés secrètes x et y. Etant donnés s, m, $g^x \ (mod\ p)$ et $g^y \ (mod\ p)$ (dans le DSA, Y seulement est révélé) les relations entre a, b, et c peuvent être vérifiées à partir de l'équation :

$$(g^x \ mod\ p)^a * (g^y \ mod\ p)^b \equiv g^c \ mod\ p$$

Les valeur possibles pour a, b, et c sont données dans le livre de Bruce Schneier ("*Applied Cryptography*") comme étant :

$a = m, \ b = Y, \ c = s$
$a = 1, b = Y.m, c = s$
$a = 1, b = Y.m, c = m.s$
$a = 1, b = Y.m, c = Y.s$
$a = 1, b = m.s, c = Y.s$

La convention pour les signes pouvant être changée ; a, b et c peuvent être remplacés par leurs opposés. Toutes les différentes possibilités pouvant bien sûr être obtenues en changeant les signes. Ce ne sont cependant pas les seules possibilités. D'autres auteurs proposent :

$a = m, b = Y, c = s$

$a = Y$, $b = m$, $c = s$
$a = s$, $b = Y$, $c = m$
$a = Y$, $b = s$, $c = m$
$a = m$, $b = s$, $c = Y$
$a = s$, $b = m$, $c = Y$
Les trois éléments pouvant être dans n'importe quel ordre.

3 Symétrique / Asymétrique : Petit Comparatif

Dans la section précédente, nous venons de voir que la cryptographie à clé publique apportait une solution à bien plus de problèmes que la cryptographie à clé secrète. Alors on peut se demander quelle est l'utilité de la cryptographie symétrique (et de l'AES en particulier).

Il existe deux raisons fondamentales à ce choix.

D'une part, une raison pratique. En effet, la plupart des systèmes de chiffrement à clé publique sont très lents. Le RSA est par exemple plusieurs centaines de fois plus lent que l'AES en programmation logicielle et est complètement hors du coup en implantation matérielle. A notre époque où la vitesse de transmission de l'information constitue un enjeu crucial, l'algorithme de chiffrement ne doit pas être un facteur limitant.

D'autre part, du point de vue de la sécurité se posent des problèmes relatifs à la structure même des systèmes de chiffrement à clé publique.
Il est en effet frappant de constater que la taille des clés nécessaire en cryptographie à clé publique pour assurer une sécurité satisfaisante est plus grande que la taille des clés en cryptographie à clé secrète. La notion de l'importance de la taille de clé pour assurer la sécurité n'est légitime que dans le cas de la clé secrète.

En fait, ces systèmes reposent sur l'hypothèse que les seules attaques possible sont ce que l'on nomme les attaques exhaustives qui consitent à énumérer toutes les clés possibles. Par exemple, dans le cas d'une clé de 128 bits, la taille de l'espace à énumérer est 2^{128}. En revanche, dans le cas de la clé publique, la taille de clé n'a de légitimité que lorsqu'on considère le même système. En effet, le système RSA de 512 bits par exemple est bien moins sûr qu'un AES de 128 bits. La seule mesure légitime pour évaluer un cryptosystème à clé publique est la complexité de la meilleure attaque connue. Ce qui fait toute la différence on n'est jamais à l'abri de percées théoriques. Très récemment, un groupe de chercheurs est parvenu à factoriser un nombre de 512 bits. En conséquence, pour avoir une sécurité suffisante pour les années à venir, on conseille généralement d'utiliser des nombres n de 1024 bits.

En chiffrement, il vaut donc mieux utiliser des algorithmes de cryptographie à clé secrète quand c'est possible.

Une solution tout à fait intéressante et acceptable est le compromis élaboré par Zimmermann pour concevoir PGP.

Le principe du chiffrement est le suivant : supposons que l'émetteur A et le destinataire B souhaitent communiquer de manière intègre, en utilisant un algorithme à clé secrète (en l'occurence, pour PGP, c'est l'algorithme IDEA). Ils se mettent d'accord sur la clé secrète par un protocole d'échange de clés. Ce genre de protocole utilise des propriétés de cryptographie à clé publique. Ensuite ils communiquent en utilisant l'algorithme IDEA qui est à clé privée. Une fois que leur conversation est terminée, ils jettent la clé de session.

La même façon de procéder peut être utilisée avec un codage AES, au lieu d'un codage IDEA.

Un tel système combine alors les avantages des deux types de crytographie : non partage des clés et intégrité des messages, ainsi que rapidité et sécurité d'exécution.

Conclusion

Après la présentation de ces différents algorithmes de codage, nous pouvons dire qu'il est nécessaire de distinguer les deux cas symétriques et asymétriques. En effet, chacun d'eux a ses particularités et n'auront pas les mêmes applications.

Dans les cas où les deux interlocuteurs sont sûrs, il est préférable d'utiliser un chiffrement symétrique. Parmis les chiffrements symétriques, le chiffrement AES, très récent, s'avère être le plus efficace. Il est d'ailleurs très employé depuis son officialisation.

Dans le cas où il est préférable que les deux interlocuteurs ne possèdent pas la même clé, pour quelque raison que se soit, un codage asymétrique comme le RSA s'avère plus approprié. Cependant, les codes asymétriques s'avèrent nettement plus lents que les codes symétriques. Il est donc conseillé de choisir un mot aléatoirement puis de le chiffrer avec la clé publique, avant de l'envoyer. Ce nombre pourra alors servir de clé privée pour un codage symétrique. Cela permet alors même de vérifier directement que les messages suivants proviennent du même expéditeur, sans avoir la nécessité absolue d'y ajouter une clé.

Bibliographie et Références

"Cryptography, Theory and Practice"
Douglas R. Stinson
(http ://www.cacr.math.uwaterloo.ca/ dstinson/CTAP.html)

"A Specification for the AES Algorithm"
Dr. Brian Gladman
(http ://www.gladman.uk.net/)

"Applied Cryptography, protocols, Algorithms, and source codes in C"
Bruce Schneier

De nombreux sites m'ont également été utiles dans mes recherches ; voici les adresses des plus intéressants.

Présentation de tous les principaux systèmes de cryptographie :
http ://www.securiteinfo.com/

De très nombreux exemples simples de calculs sur différents algorithmes cryptographiques :
http ://home.ecn.ab.ca/ jsavard/crypto/entry.htm

Site du département Computer Security Resource Center du NIST :
http ://csrc.nist.gov/encryption/

Présentation détaillée du système RSA :
http ://www.rsasecurity.com/rsalabs

Quelques exemples de calculs et d'implémentation du système RSA :
http ://minimum.inria.fr/ raynal/

Présentation explicative des fonctions de hachage :
http ://home.pacbell.net/tpanero/crypto/cryptonotes.html