N° d'ordre : 26 / 012 – M / MT

REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE

MINISTERE DE L'ENSEIGNEMENT SUPERIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE « HOUARI BOUMEDIEN »

FACULTE DE MATHEMATIQUES

MEMOIRE

Présenté pour l'obtention du diplôme de MAGISTER

EN : MATHEMATIQUE

Spécialité : ALGEBRE ET THEORIE DES NOMBRES

Par : BEKHOUCHE SAÏDA

Sujet

FONDEMENTS MATHEMATIQUES ET FONCTIONNEMENT DU STANDARD DE CHIFFREMENT AVANCE *(AES)*

Soutenue publiquement , le : 23 / 01 / 2012, devant le jury composé de :

Mr AIDER Meziane	**Professeur à l'USTHB**	**Président**
Mr HACHAÏCHI M -Salah	**Maître de Conférences/A à l'USTHB**	**Directeur du mémoire**
Mr ZITOUNI Mohamed	**Professeur à l'USTHB**	**Examinateur**
Mr HERNANE Mohand Ouamar	**Professeur /A à l'USTHB**	**Examinateur**

Remerciements

Je voudrais remercier profondément Monsieur HACHAÏCHI Mohamed Saleh,

mon directeur de mémoire et maître de conférences à l' USTHB , de m'avoir proposé ce sujet . Je le remercie vivement pour son aide précieuse , ses conseils et sa patience .

Je remercie vivement Monsieur ZITOUNI Mohamed , professeur à l' USTHB d'avoir accepté d'examiner ce travail en consacrant son temps à lire ce mémoire. Je remercie aussi Monsieur HERNANE Mohand Ouamar professeur à l'USTHB , d'avoir accepté d'examiner ce travail . Que Monsieur AIDER Meziane , professeur à l' USTHB , qui ont bien voulu faire partie du jury , trouvent ici mes remerciements les plus sincères .

Mes derniers et profonds remerciements vont à ma famille , sans oublier mes amis et tous ceux qui ont contribué , de près ou de loin à ma formation.

TABLE DES MATIERES

Université des Sciences et de la Technologie Houari Boumediène (USTHB)

Faculté de Mathématiques

Résumé du Mémoire de Magister en Algèbre et Théorie des Nombres

Présenté par :

Melle BEKHOUCHE Saïda

Sous la direction de : ***HACHAICHI Mohamed-Salah*** (*Maître de Conférences-A*)

Thème

Fondements mathématiques et fonctionnement du standard de chiffrement avancé *Rijndael (AES)*

RESUME

Le ***Rijndael***, est un algorithme de chiffrement par blocs itérés mis au point par deux cryptographes belges *J. Daemen* et *V. Rijmen*, fut choisi en Octobre 2000 par le *NIST (National Institute of Standards and Technology-USA)* en tant que standard de chiffrement avancé ***AES*** (*Advanced Encryption Standard*), en remplacement du ***DES*** (*Data Encryption Standard*) dont la sécurité a été remise en cause par l'attaque dite «*brute force attack*» opérant sur l'espace de ses 2^{56} clés possibles.
L'***AES*** procède par blocs de 128 bits avec des clés de longueur 128, 192 ou 256 bits. Chaque bloc subit une séquence de cinq transformations répétées 10 (***AES 128***), 12 (***AES 192***) ou 14 fois (***AES 256***):

1. ***Addition de la clé secrète*** (par un *XOR* $\oplus$).
2. ***Substitution d'octets*** : les 128 bits sont répartis en 16 blocs de 8 bits, eux-mêmes dispatchés dans un tableau 4×4. Chaque octet du bloc est remplacé par un octet d'une table de substitution «***S-Box***» construite au moyen d'une transformation non linéaire opérant sur le *corps de Galois* $\boldsymbol{GF(2^8)}$.
3. ***Décalage de lignes***: les trois dernières lignes sont décalées cycliquement vers la gauche: la $2^{ème}$ ligne est décalée d'une colonne, la $3^{ème}$ ligne de 2 colonnes, et la $4^{ème}$ ligne de 3 colonnes.

4. ***Brouillage des colonnes*** : Chaque colonne est transformée par combinaisons linéaires des différents éléments de la colonne, ce qui revient à multiplier une matrice 4×4 par une autre matrice 4×4, la multiplication est effectuée dans l'anneau

$$\frac{GF(2^8)[x]}{(x^4+1)GF(2^8)[x]}$$

5. ***Addition de la clé de tour*** : à chaque tour, une clé de tour est générée à partir de la clé secrète par un sous-algorithme (dit de cadencement). Cette clé de tour est ajoutée par un « *XOR* » au dernier bloc obtenu.

En Juin 2003, le gouvernement américain a annoncé: «l'architecture et la longueur de toutes les tailles de clés de l'algorithme ***AES*** sont suffisantes pour protéger des documents classifiés jusqu'au niveau «*SECRET*». Le niveau «*TOP SECRET*» nécessite des clés de 192 /256 bits »

INTRODUCTION

Le 15 mai 1973 le *NBS* des Etats-Unis (*National Bureau of Standards*, aujourd'hui appelé *National Institute of Standards and Technology- NIST*) a lancé un appel dans le *Federal Register* pour la création d'un algorithme de chiffrement répondant aux critères suivants :

- posséder un haut niveau de sécurité lié à une clé de petite taille servant au chiffrement et au déchiffrement
- être compréhensible
- ne pas dépendre de la confidentialité de l'algorithme
- être adaptable et économique
- être efficace et exportable

Fin 1974, **IBM** propose « ***Lucifer*** », qui, sous l'instigation de la *NSA* (*National Security Agency-USA*), est modifié le 23 novembre 1976 pour donner le ***DES*** (*Data Encryption Standard*). Le ***DES*** a finalement été approuvé en 1978 par le *NBS*. Le ***DES*** fut normalisé par l'*ANSI* (*American National Standard Institute-USA*) sous le nom de ***ANSI X3.92***. Adopté comme standard de chiffrement par le gouvernement des Etats-Unis pendant un bon nombre d'années, le ***DES*** a été cassé en Juin 1997 par l'attaque dite « *exhaustiv search attack* » menée pendant 3 semaines par une fédération de PC sur *Internet* opérant sur l'espace de ses 2^{56} clés possibles.

Pour améliorer la sécurité du ***DES,*** une solution à court terme a consisté à chaîner trois chiffrements ***DES*** à l'aide de deux clés de 56 bits (ce qui équivaut à une clé de 112 bits). Ce procédé est appelé ***Triple DES*** (***TDES***)**.**

Si le ***TDES*** est considéré aujourd'hui encore comme offrant une sécurité satisfaisante, il est toutefois trois fois plus lent que le ***DES***.

C'est pourquoi en Janvier 1997 le *NIST* lança un nouvel appel à projet pour élaborer l'***AES*** (*Advanced Encryption Standard*), un algorithme de chiffrement destiné à remplacer le ***DES***.

Le cahier des charges comportait les points suivants :

- L'algorithme doit être robuste, à blocs itérés, à clé symétrique, pour des utilisations commerciales et gouvernementales ;
- Il doit être plus efficace que le ***TDES***,
- Il doit présenter une large portabibilité : protocoles de réseau, cartes à puces, processeurs dédiés,…
- Il doit être très rapide.
- Une lecture facile de l'algorithme.
- L'algorithme doit supporter les combinaisons [longueur de clé]-[longueur de bloc] suivantes: ***AES 128*** :128-128, ***AES 192*** :192-128 et ***AES 256*** : 256-128 bits

Au 15 Juin 1998, date de la fin des candidatures, 21 projets ont été déposés : CAST-256, CRYPTON, DEAL, DFC, E2, FROG, HPC, LOKI97, MAGENTA, MARS, RC6, Rijndael, SAFER+, Serpent et Twofish.

Le 9 Août 1999, une « short list » de cinq nominés est annoncée regroupant les algorithmes ***Mars*** *(IBM),* ***RC6*** *(RSA),* ***Rijndael***, ***Serpent*** et ***Twofish.***

Le 2 Octobre 2000, le ***NIST*** choisi le ***Rijndael*** comme support pour l'***AES***, un algorithme mis au point par deux belges, *J. Daemen* et *V. Rijmen*.
L'***AES*** procède par blocs de 128 bits avec des clés de longueur 128, 192 ou 256 bits. Chaque bloc (*state*) subit une séquence de cinq transformations répétées 10 (***AES 128***), 12 (***AES 192***) ou 14 fois (***AES 256***) :

1 *Addition de la clé secrète* (par un *XOR* $\oplus$) .

2 *Substitution d'octets* : les 128 bits sont répartis en 16 blocs de 8 bits, eux-mêmes dispatchés dans un tableau 4×4. Chaque octet du bloc est remplacé par un octet d'une table de substitution définie au préalable: la «***S-Box***» construite avec une transformation non linéaire opérant sur le *corps de Galois* $\boldsymbol{GF(2^8)}$ à 256 éléments.

3 *Décalage de lignes* : les trois dernières lignes sont décalées cycliquement vers la gauche: la $2^{ème}$ ligne est décalée d'une colonne, la $3^{ème}$ ligne de 2 colonnes, et la $4^{ème}$ ligne de 3 colonnes .

4 *Brouillage des colonnes* : chaque colonne est transformée par combinaisons linéaires des différents éléments de la colonne (ce qui revient à multiplier la matrice 4×4 par une autre matrice 4×4). Les calculs sur les octets sont effectués dans le corps $\boldsymbol{GF(2^8).}$

5 *Addition de la clé de tour* : à chaque tour, une clé de tour est générée à partir de la clé secrète par un sous-algorithme (dit de cadencement). Cette clé de tour est ajoutée par un « *ou exclusif* » au dernier bloc (*state*) obtenu.

Trois critères principaux ont été respectés dans sa conception :

- Résistance à toutes les attaques connues alors,
- Rapidité du code sur une grande variété de plates-formes (logicielles et matérielles),
- Simplicité dans la conception.

Notre travail consiste à préciser les fondements mathématiques et le fonctionnement du *Standard de Chiffrement Avancé* (***AES***). Nous avons pour cela adopté la présentation

suivante :

- *Introduction* **,**
- *Chapitre 1*: consacré à une étude bibliographique,
- *Chapitre 2*: dans lequel nous développons les mathématiques utilisées par l'***AES,***
- *Chapitre 3* : relatif à la description et au fonctionnement de l'***AES,***
- *Chapitre 4* : une application clôturera notre étude,
- *Bibliographie*.

Chapitre 1- ETUDE BIBLIOGRAPHIQUE

Le but traditionnel de la cryptographie est d'élaborer des méthodes permettant de transmettre des données de manière confidentielle selon les principes énoncés par *A. Kerckhoffs* [Ker] :

Une information cryptée ne doit en aucun cas pouvoir être déchiffrée sans la connaissance de sa clé,
Les interlocuteurs ne doivent pas subir de dégâts au cas où le système de cryptage serait dévoilé,
La clé doit être simple et modifiable à souhait,
Les cryptogrammes doivent être transportables,
Le système doit être simple d'utilisation,
Le cryptosystème doit être au préalable examiné par des experts.

La cryptographie moderne s'intéresse plus généralement aux problèmes de sécurité des communications, *partant du principe que tout secret est un point de cassure possible*. Le but est d'offrir un certain nombre de services de sécurité comme :

- **La confidentialité,**
- **L'intégrité,**
- **L'authentification des données transmises,**
- **L'authentification d'un tiers,**
- **Le contrôle d'accès (autorisation ou non d'accès à des objets)**

On utilise à cet effet des techniques basées sur des algorithmes cryptographiques. Il existe deux grandes familles d'algorithmes de chiffrement : les algorithmes à clef secrète, ou algorithmes symétriques, et les algorithmes à clef publique, ou algorithmes asymétriques.

1.1- Chiffrement symétrique ou à clef secrète

Dans la cryptographie symétrique (*ou conventionnelle*), les *clefs de chiffrement et de déchiffrement sont identiques* : c'est la clef secrète, qui doit être connue des tiers communicants et d'eux seuls.

Les algorithmes symétriques sont de deux types :

- les algorithmes de *chiffrement en continu*, qui agissent sur le texte en clair un bit à la fois;
- les algorithmes de *chiffrement par blocs*, qui opèrent sur le texte en clair par groupes de bits appelés blocs .

1.2- Modes de chiffrement par blocs

Les algorithmes de chiffrement par blocs peuvent être utilisés suivant différents modes, dont les deux principaux sont le mode ***ECB*** (*Electronic CodeBook*) et le mode ***CBC*** (*Cipher Block Chaining*).

1.2.1- Le mode *ECB* (*Electronic CodeBook*)

Le mode « *Electronic CodeBook* » est la méthode la plus simple pour utiliser un algorithme de chiffrement par blocs : un bloc du texte en clair se chiffre, indépendamment de tout, en un bloc de texte chiffré. L'avantage de ce mode est qu'il permet le chiffrement en parallèle des différents blocs composant un message .

L'inconvénient de ce mode est qu'un même bloc de texte en clair sera toujours chiffré en un même bloc de texte chiffré. Or, dans le chiffrement sur un réseau par exemple, les données à chiffrer ont des structures régulières facilement repérables par un attaquant, qui pourra donc obtenir beaucoup d'informations. D'autre part, un attaquant actif pourra facilement manipuler les messages chiffrés en retirant, répétant ou interchangeant des blocs. Un autre inconvénient, qui s'applique au chiffrement par blocs, est l'amplification d'erreur: si un bit du texte chiffré est modifié pendant le transfert, tout le bloc de texte en clair correspondant sera altéré .

1.2.2- Le mode *CBC* (*Cipher Block Chaining*)

La solution aux problèmes posés par le mode ***ECB*** est d'utiliser une technique dite *de chaînage*, dans laquelle chaque bloc du cryptogramme dépend non seulement du bloc de texte en clair correspondant, mais aussi de tous les blocs précédents, ce qui assure une certaine résistance à l'égard des attaques standards [RSA Lab].

En mode « *Cipher Block Chaining* » chaque bloc de texte en clair est combiné par un *XOR* ($\oplus$) avec le bloc chiffré précédent, avant d'être chiffré. Le premier bloc du texte en clair est, quant à lui, combiné avec un bloc appelé *« **vecteur d'initialisation** »*. L'utilisation d'un *vecteur d'initialisation* différent pour chaque message permet de s'assurer que deux messages identiques (ou dont les premiers blocs sont identiques) donneront des cryptogrammes totalement différents.

Le gros avantage du mode ***CBC*** réside dans le fait que la structure du texte en clair est obscurcie par le chaînage. Un attaquant ne peut plus manipuler le cryptogramme, excepté en retirant des blocs au début ou à la fin. Un inconvénient est qu'il n'est plus possible de paralléliser le chiffrement des différents blocs.

« On pourrait craindre que le« *Cipher Block Chaining* » n'entraîne une propagation d'erreur importante. De fait, une erreur d'un bit sur le texte en clair affectera tous les blocs chiffrés suivants. Par contre, si un bit du texte chiffré est modifié au cours du transfert, seul le bloc de texte en clair correspondant et un bit du bloc de texte en clair suivant seront endommagés : le mode ***CBC*** est dit *auto-réparateur* ».

1.3- Chiffrement par blocs avec itération

Un algorithme de chiffrement par blocs avec itération est un algorithme qui chiffre les blocs par un processus comportant plusieurs rondes. *Dans chaque ronde, la même transformation est appliquée au bloc,* en utilisant une sous-clef dérivée de la clef de chiffrement. En général, un nombre de rondes élevé garantit une meilleure sécurité, au détriment bien évidemment des performances.

Un cas particulier d'algorithmes de chiffrement par blocs avec itération est la famille des « *chiffres de Feistel »***.** Dans un *chiffre de Feistel*, un bloc de texte en clair est découpé en deux ; la transformation de ronde est appliquée à une des deux moitiés, et le résultat est combiné avec l'autre moitié par un *XOR (*$\oplus$*)*. Les deux moitiés sont alors inversées pour

l'application de la ronde suivante. Un avantage de ce type d'algorithme est que chiffrement et déchiffrement sont structurellement identiques.

1.4- Exemples d'algorithmes symétriques

La méthode la plus employée pour concevoir un procédé de chiffrement est de chercher à réaliser une transformation suffisamment complexe et irrégulière pour que sa crypanalyse soit d'une complexité aussi élevée que possible. Cette méthode empirique, toutefois, ne fournit aucune garantie quant à la robustesse de l'algorithme résultant.

Des exemples d'algorithmes symétriques sont le ***DES*** (*Data Encryption Standard*), le ***DES triple à deux ou trois clefs***, l'***IDEA*** (*International Data Encryption Algorithm*), le ***RC5*** (*Rivest's Code 5*), etc… .

1.4.1- Masque jetable (One-Time-Pad)

Il existe un *algorithme de chiffrement prouvé inconditionnellement sûr* : le chiffre de ***Vernam*** ou *masque jetable* ou *One-time pad*, proposé en 1917 par *Gilbert Sandford Vernam* [Ver], ingénieur à *l'American Telephone and Telegraph Company*, et perfectionné par *Joseph O. Mauborgne*, chef du service cryptographique de l'armée américaine, qui introduisit la notion de clé aléatoire. Le *One-time pad* a été prouvé inconditionnellement sûr par *C. Shannon* [Sha]

Dans ce système, la "*clef*" a la même taille que le texte à chiffrer et est appelée *masque jetable:* "*masque*", car cette clef est combinée par un *XOR* ($\oplus$) avec le texte en clair pour obtenir le texte chiffré ; "*jetable*", car ***une clef ne doit servir qu'une seule fois***, sans quoi un attaquant pourrait retrouver le texte en clair. Le point important de ce système est que la ***clef est générée de façon totalement aléatoire*** **;** tout masque est alors équiprobable, si bien qu'un attaquant n'a aucune information pour baser sa cryptanalyse.

En effet, si M est le message (*plaintext*) à chiffrer, C le cryptogramme (*cyphertext*) correspondant et K le *masque jetable*, nous avons $C = M \oplus K$. …… Supposons que l'attaquant connaisse C ; quel que soit M', il existe K' tel que $C = M' \oplus K'$, c'est évidemment $K' = M' \oplus C$: tous les messages en clair sont équiprobables ! Sans information sur K, l'attaquant n'a donc aucun moyen de savoir quel est le bon texte en clair.

Les principaux inconvénients de ce système sont *la taille de la clef* et *la nécessité de posséder un générateur aléatoire pour sa création.* La conséquence en est que ce système est inutilisable dans le cadre du chiffrement d'un flux important de données et reste limité à des applications extrêmes, comme le chiffrement des communications avec le « *téléphone rouge* » entre la Maison Blanche et le Kremlin.

1.4.2- DES (Data Encryption Standard)

Le gouvernement américain a adopté, en Novembre 1976, le ***DES*** (*Data Encryption Standard*) comme algorithme de chiffrement standard officiel.

Le ***DES*** est un algorithme de chiffrement par blocs qui agit sur des **blocs** *de 64 bits*. C'est un chiffre de ***Feistel*** à 16 rondes. *La longueur de la clef est de 56 bits*. Généralement, celle-ci est représentée sous la forme d'un nombre de 64 bits, mais un bit par octet est utilisé pour le contrôle de parité. Les sous-clefs utilisées par chaque ronde ont une longueur de 48 bits.

Le ***DES*** a fait l'objet d'un cassage complet par l'attaque dite «*exhaustive search attack ».*

Le ***Triple DES*** (***TDES***) est une variante du ***DES*** qui consiste à appliquer l'algorithme trois fois à chaque bloc : chiffrement, déchiffrement puis de nouveau chiffrement. Les clefs utilisées pour chaque application du ***DES*** peuvent être toutes les trois distinctes, ou bien on peut utiliser seulement deux clefs distinctes : une pour le chiffrement et une pour le déchiffrement. Dans tous les cas, la longueur efficace de la clef du ***Triple DES*** ne dépasse pas 112 bits.

Un nouvel algorithme : *Advanced Encryption Standard (**AES**)*, a remplacé le ***DES***.

1.5- Chiffrement asymétrique ou à clef publique

Avec les algorithmes asymétriques, *les clefs de chiffrement et de déchiffrement sont distinctes et ne peuvent se déduire l'une de l'autre*. On peut donc rendre l'une des deux clés, publique tandis que l'autre reste privée (*secrète*). C'est pourquoi on parle de chiffrement à clef publique. Si la clef publique sert au chiffrement, tout le monde peut chiffrer un message, que seul le propriétaire de la *clef privée* pourra déchiffrer. On assure ainsi la *confidentialité*. Certains algorithmes permettent d'utiliser la clef privée pour chiffrer. Dans ce cas, n'importe qui pourra déchiffrer, mais seul le possesseur de la clef privée peut chiffrer. Cela permet donc la signature de messages.

Le concept de *cryptographie à clef publique* fut inventé par *Whitfield Diffie* et *Martin Hellman* [Dif] en 1976, dans le but de résoudre le problème de distribution des clefs posé par la cryptographie à clef secrète. De nombreux algorithmes permettant de réaliser un cryptosystème à clef publique ont été proposés depuis. Ils sont le plus souvent *basés sur des problèmes mathématiques difficiles à résoudre* : donc leur sécurité est conditionnée par ces problèmes, sur lesquels on a maintenant une vaste expertise. Mais, si quelqu'un trouve un jour le moyen de simplifier la résolution d'un de ces problèmes, l'algorithme correspondant s'écroulera.

Nombre des algorithmes proposés pour la cryptographie à clef publique se sont révélés rapidement non sûrs, ou non réalisables sur le plan pratique. Tous les algorithmes actuels présentent l'inconvénient d'être *bien plus lents que les algorithmes à clef secrète* ; de ce fait, *ils sont souvent utilisés non pour chiffrer directement des données, mais pour chiffrer une clef de session secrète*. Certains algorithmes asymétriques ne sont adaptés qu'au chiffrement, tandis que d'autres ne permettent que la signature. Trois algorithmes sont utilisables à la fois pour le chiffrement et pour la signature : ***RSA, ElGamal*** et ***Rabin***.

1.5.1- Algorithmes à empilement

Historiquement, le premier de ce type d'algorithme fut proposé par *Merkle* et *Hellman* [Mer] en 1978. Il repose sur le problème d'empilement: « *problème du sac à dos*» (*knapsack problem*), qui est un problème NP-complet utilisé notamment dans des protocoles de cryptographie, tels que le cryptosystème de *Merkle-Hellman* ou le cryptosystème de *Chor-Rivest* [Cho].

Cependant, les algorithmes de chiffrement basés sur le *problème du sac à dos* ont tous été cassés à ce jour, le dernier en date étant celui de *Chor-Rivest* complètement cryptanalysé en 1998 par *Vaudenay S.* [Vau] de l'E.N.S/CNRS-France.

Le problème du sac à dos est un exemple de méprise en ce qui concerne les liens entre la NP-complétude et la cryptographie.

1.5.2- RSA

Inventé par *Rivest, Shamir* et *Adleman* en 1978, ***RSA*** permet le chiffrement et la signature. Il est aujourd'hui encore très largement utilisé. Cet algorithme repose sur la difficulté de factoriser des grands nombres entiers.

Voici comment se fait la génération des paires de clefs :

1. On commence par choisir deux grands nombres premiers, p et q, et on calcule $N = pq$. N est rendu public ; p et q doivent rester secrets et peuvent être détruits une fois les clefs générées.
2. On choisit ensuite aléatoirement une clef publique e telle que e et $\varphi(N) = (p-1)(q-1)$ soient premiers entre eux, φ étant la fonction indicatrice d'Euler.
3. La clef privée d est obtenue grâce à l'algorithme d'*Euclide*: $ed \equiv 1 \bmod (p-1)(q-1)$.

Soit M le message en clair et C le chiffré. La fonction de chiffrement est, de façon simplifiée, $C = M^e \pmod N$ (si M est plus grand que N, il est séparé en morceaux de valeur inférieure à N et chaque morceau est chiffré séparément suivant cette formule). Du fait de la relation entre e et d, la fonction de déchiffrement correspondante est $M = C^d \bmod N$.

La signature se fait de manière similaire, en inversant e et d, c'est-à-dire en chiffrant avec une clef privée et en déchiffrant avec la clef publique correspondante : $S = M^d \pmod N$ et $M = S^e \pmod N$.

Pour un attaquant, retrouver la clef privée à partir de la clef publique nécessite de connaître $(p-1)(q-1) = N+1-p-q$, donc de connaître p et q. Pour cela, il doit factoriser le grand nombre entier N. Donc N doit être suffisamment grand pour que cela ne soit pas possible dans un temps acceptable par rapport au niveau de sécurité requis. Compte tenu de l'augmentation des capacités de calcul des ordinateurs (*loi de Moore*) et des avancées mathématiques en matière de factorisation des grands nombres entiers, la taille minimale des clefs devrait augmenter au cours du temps. Le schéma ci-après résume les processus :

Clefs			
Clef publique	$\boldsymbol{n = pq}$, où $\boldsymbol{p}$ et $\boldsymbol{q}$ sont deux grands nombres premiers tenus secrets, $\boldsymbol{e}$ choisi de manière que $\boldsymbol{e}$ et $\boldsymbol{(p-1)(q-1)}$ soient premiers entre eux		
Clef privée	$\boldsymbol{d \equiv e^{-1} \bmod (p-1)(q-1)}$		
Algorithmes			
Chiffrement	$\boldsymbol{c \equiv m^e \bmod n}$	Déchiffrement	$\boldsymbol{M \equiv c^d \bmod n}$
Signature	$\boldsymbol{s \equiv m^d \bmod n}$	Vérification	$\boldsymbol{M \equiv s^e \bmod n}$

1.6- Générateurs aléatoires et pseudo-aléatoires

La cryptographie a souvent recours à des nombres aléatoires. Ainsi, lorsqu'une personne génère une clef secrète (*ou privée*), elle doit faire intervenir le hasard de façon à empêcher un

adversaire de deviner la clef. De même, certains protocoles cryptographiques nécessitent, pour éviter la re-jouabilité par exemple, l'utilisation d'aléas imprévisibles par les attaquants.

Malheureusement, il est impossible de produire des suites aléatoires à l'aide uniquement d'un ordinateur : le générateur sera toujours périodique, donc prévisible. On a donc recours à des générateurs dits pseudo-aléatoires et cryptographiquement sûrs. Un tel générateur doit présenter les caractéristiques suivantes :

1. La période de la suite doit être suffisamment grande pour que les sous-suites finies utilisées avec l'algorithme ou le protocole cryptographique ne soient pas périodiques.
2. Ces sous-suites doivent, sur le plan statistique, être aléatoires.
3. Le générateur doit être imprévisible, au sens où il doit être impossible de prédire le prochain aléa à partir des aléas précédents.

La plupart des générateurs pseudo-aléatoires sont construits en utilisant des registres à décalage (*shift registers* en anglais) et, en particulier, les registres à décalage à rétroaction linéaire (*Linear Feedback Shift Registers, LFSR)*. Ces derniers présentent l'inconvénient de générer des suites linéaires, si bien que des grands nombres générés à partir de sous-suites sont fortement corrélés. C'est pourquoi les générateurs pseudo-aléatoires sont généralement construits en combinant, à l'aide d'une fonction non linéaire, plusieurs registres à décalage de tailles différentes. Ce type de générateur est très utilisé par les algorithmes de chiffrement par flots.

Si l'on veut vraiment générer des suites aléatoires, au sens où ces suites sont de plus non-reproductibles, on a généralement recours à des éléments extérieurs comme les déplacements de la souris, la vitesse de frappe, ...

1.7- Fonctions de hachage à sens unique, signature numérique et scellement

Aussi appelée fonction de condensation, une *fonction de hachage* est une fonction qui convertit une chaîne de longueur quelconque en une chaîne de taille inférieure et généralement fixe; la chaîne résultante est appelée *empreinte* (*digest* en anglais) ou condensé de la chaîne initiale.

Une *fonction à sens unique* est une fonction facile à calculer mais difficile à inverser. La cryptographie à clef publique repose sur l'utilisation de fonctions à sens unique à brèche secrète : pour qui connaît le secret (*i.e.* la clef privée), la fonction devient facile à inverser.

Une *fonction de hachage à sens unique* est une fonction de hachage qui est en plus une fonction à sens unique : il est aisé de calculer l'empreinte d'une chaîne donnée, mais il est difficile d'engendrer des chaînes qui ont une empreinte donnée, et donc de déduire la chaîne initiale à partir de l'empreinte. On demande généralement en plus à une telle fonction d'être *sans collision,* c'est-à-dire qu'il soit impossible de trouver deux messages ayant la même empreinte. On utilise souvent le terme fonction de hachage pour désigner, en fait, une fonction de hachage à sens unique sans collision.

La plupart des fonctions de hachage à sens unique sans collision sont construites par itération d'une fonction de compression : le message M est décomposé en n blocs $m_1,...,m_n$, puis une fonction de compression f est appliquée à chaque bloc et au résultat de la compression du bloc précédent ; l'empreinte $h(M)$ est le résultat de la dernière compression.

Des exemples de fonctions ainsi conçues sont ***MD5, SHA, RIPE-MD.***

1.7.1- MD5 (Message Digest 5)

Développé par *Rivest* en 1991, ***MD5*** produit une empreinte de 128 bits à partir d'un texte de taille arbitraire. ***MD5*** manipule le texte d'entrée par blocs de 512 bits.

1.7.2- SHA (Secure Hash Algorithm)

SHA est la fonction de hachage utilisée par ***SHS*** (*Secure Hash Standard*), la norme du Gouvernement Américain pour le hachage**.** ***SHA-1*** est une amélioration de ***SHA*** publiée en 1994**.** ***SHA-1*** produit une empreinte de 160 bits à partir d'un message de longueur maximale 2^{64} bits. Tout comme ***MD5***, ***SHA-1*** travaille sur des blocs de 512 bits.

1.7.3- RIPE-MD

Développée dans le cadre du projet ***RIPE*** (*RACE Integrity Primitives Evaluation*) de la Communauté Européenne, ***RIPE-MD*** fournit une empreinte de 128 bits. ***RIPE-MD-160*** est une version renforcée de ***RIPE-MD*** qui fournit une empreinte de 160 bits.

1.8- Intégrité et authentification de l'origine des données

Parmi les problèmes auxquels s'attaque la cryptographie, on trouve l'authentification de l'origine des données et l'intégrité : lorsque l'on communique avec une autre personne au travers d'un canal peu sûr, on cherche à ce que *le destinataire puisse s'assurer que le message émane bien de l'auteur auquel il est attribué et qu'il n'a pas été altéré pendant le transfert.*

Les fonctions de hachage à sens unique interviennent dans la résolution de ces problèmes. Si l'on dispose d'un canal sûr (mais plus coûteux) en parallèle du canal de communication normal, on peut communiquer l'empreinte des messages par l'intermédiaire de ce canal. On assure ainsi l'intégrité des données transférées.

Si on ne dispose pas d'un canal sûr, le problème se complique: si l'on transfère l'empreinte sur un canal de communication non sûr, un intercepteur peut modifier les données puis recalculer l'empreinte. Il convient donc de trouver une méthode pour s'assurer que seul l'expéditeur est capable de calculer l'empreinte. Pour cela, on peut utiliser, par exemple, une fonction de hachage à sens unique qui fonctionne de plus avec une clef privée. On remarquera que, ce faisant, on fournit également l'authentification de l'origine des données. Inversement, si on désire fournir l'authentification de l'origine des données et que l'on utilise pour cela un moyen qui ne garantit pas l'intégrité des données authentifiées, un intrus peut modifier le message et donc faire accepter comme authentifiées des données qu'il a choisies. C'est pourquoi *intégrité et authentification de l'origine des données sont généralement fournies conjointement par un même mécanisme.*

1.9- Signature numérique

La norme [ISO 7498-2] définit la signature numérique comme des "*données ajoutées à une unité de données, ou transformation cryptographique d'une unité de données, permettant à un destinataire de prouver la source et l'intégrité de l'unité de données et protégeant contre la*

contrefaçon (par le destinataire, par exemple)". La mention "*protégeant contre la contrefaçon*" implique que seul l'expéditeur doit être capable de générer la signature.

Une signature numérique fournit donc les services d'*authentification de l'origine des données, d'intégrité des données et de non-répudiation*. Ce dernier point la différencie des *codes d'authentification de message*, et a pour conséquence que la plupart des algorithmes de signature utilisent la cryptographie à clef publique.

Sur le plan conceptuel, la façon la plus simple de signer un message consiste à chiffrer celui-ci à l'aide d'une clef privée: seul le possesseur de cette clef est capable de générer la signature, mais toute personne ayant accès à la clef publique correspondante peut la vérifier. Dans la pratique, cette méthode est peu utilisée du fait de sa lenteur.

Une autre méthode utilisée pour signer consiste à *calculer une empreinte du message à signer et à ne chiffrer que cette empreinte*. Le calcul d'une empreinte par application d'une fonction de hachage étant rapide et la quantité de données à chiffrer étant fortement réduite, cette solution est bien plus rapide que la précédente.

1.9.1- DSS

Proposé par le ***NIST*** en 1991, puis finalement adopté en 1994, ***DSS*** (*Digital Signature Standard*) est la norme de signature numérique du Gouvernement Américain. Elle se base sur l'algorithme ***DSA*** (*Digital Signature Algorithm*), qui utilise ***SHA*** comme fonction de hachage à sens unique et ***ElGamal*** pour la génération et la vérification de la signature.

1.9.2- RSA

Si ***DSS*** est la norme officielle aux U.S.A., ***RSA*** est en revanche une norme de fait, qui est en pratique beaucoup plus utilisé pour la signature que ***DSA***.

1.9.3- Scellement

Tout comme la signature numérique, le scellement fournit les services d'*authentification de l'origine des données et d'intégrité des données*, mais il ne fournit pas la non-répudiation. Ceci permet l'utilisation de la cryptographie à clef secrète pour la génération du sceau ou *code d'authentification de message*.

*Un code d'authentification de message (Message Authentication Code, **MAC**) est le résultat d'une fonction de hachage à sens unique dépendant d'une clef secrète :* l'empreinte dépend à la fois de l'entrée et de la clef. On peut construire un ***MAC*** à partir d'une fonction de hachage ou d'un algorithme de chiffrement par blocs. Il existe aussi des fonctions spécialement conçues pour faire un ***MAC***.

Une façon courante de générer un code d'authentification de message consiste à appliquer un algorithme de chiffrement symétrique en mode ***CBC*** au message; le ***MAC*** est le dernier bloc du cryptogramme.

Un moyen simple de transformer une fonction de hachage à sens unique en un MAC consiste à chiffrer l'empreinte avec un algorithme à clef secrète. Une autre méthode consiste à

appliquer la fonction de hachage non pas simplement aux données à protéger, mais à un ensemble dépendant à la fois des données et d'un secret.

1.9.4- HMAC

Une méthode de calcul de ***MAC*** à base de fonction de hachage relativement sûre est ***HMAC*** qui peut être utilisée avec n'importe quelle fonction de hachage itérative telle que ***MD5***, ***SHA-1*** ou encore ***RIPE-MD.***

Soit ***H*** une telle fonction, ***K*** le secret et ***M*** le message à protéger.

H travaille sur des blocs de longueur disons ***b*** octets (64 en général) et génère une empreinte de longueur ***l*** octets (16 pour ***MD5***, 20 pour ***SHA*** et ***RIPE-MD-160***). Il est conseillé d'utiliser un secret de taille au moins égale à ***l*** octets.

On définit deux chaînes, ***ipad*** (*inner padding data*) et ***opad*** (*outer padding data*), de la façon suivante :
- ***ipad*** = l'octet 0*x*36 répété ***b*** fois, (0*x* indique la notation hexadécimale)
- ***opad*** = l'octet 0*x*5C répété ***b*** fois.

Le ***MAC*** se calcule selon la formule suivante : $HMAC_K(M) = H (K \oplus opad, H(K \oplus ipad, M))$.

Une pratique courante avec les fonctions de calcul de ***MAC*** est de tronquer la sortie pour ne garder comme ***MAC*** qu'un nombre réduit de bits. Avec ***HMAC***, on peut ainsi choisir de ne retenir que les ***t*** bits de gauche, où $t > (l/2, 80)$.
On désigne alors sous la forme **HMAC-H-*t*** l'utilisation de ***HMAC*** avec la fonction ***H***, tronqué à ***t*** bits (par exemple, ***HMAC-SHA1-96***).

1.10- Protocoles cryptographiques

Tout comme les protocoles de communication, les protocoles cryptographiques sont une série d'étapes prédéfinies, basées sur un langage commun, qui permettent à deux ou plusieurs participants d'accomplir une tâche. Dans le cas des protocoles cryptographiques, les tâches en question sont bien sûr liées à la cryptographie: ce peut être une authentification, un échange de clef,... Une particularité des protocoles cryptographiques est que les tiers en présence ne se font généralement pas confiance et que le protocole a donc pour but d'empêcher l'espionnage et la tricherie.

1.10.1- Protocoles à apport nul de connaissance

Une situation fréquente est celle où un tiers doit *prouver à un autre la connaissance d'un secret sans rien révéler sur ce secret.* Les protocoles qui permettent de répondre à ce problème sont appelés protocoles à apport nul de connaissance ou preuves à divulgation nulle (*0-knowledge* en anglais).

Les protocoles à apport nul de connaissance prennent la forme de protocoles interactifs au cours desquels le vérificateur *Bob*, pose à *Alice*, le plaideur, une série de questions. Si *Alice* connaît le secret, elle saura répondre correctement à toutes les questions ; sinon elle aura 50% de chances de répondre correctement à chaque question. La probabilité que *Alice* connaisse

effectivement le secret après n réponses correctes est donc de ($1-\frac{1}{2^n}$). Après 10 questions par exemple, cette probabilité sera de 0,999.

Une application de tels protocoles est leur utilisation pour prouver son identité. Ainsi, le schéma d'identification ***Feige-Fiat-Shamir*** [Fei] est un système de preuve d'identité à divulgation nulle. Soit N un module ***RSA*** aléatoire, produit de deux grands nombres premiers. La clef publique de *Alice* est v, un résidu quadratique modulo N (i.e. $x^2 \equiv v \mod N$ admet une solution et $v^{-1} \mod N$ existe). La clef privée correspondante est le plus petit s tel que $s = \sqrt{v^{-1}} \mod N$.

Le déroulement d'une ronde du protocole est le suivant :

1. *Alice* choisit un nombre aléatoire $r \leq N$, puis calcule $x = r^2 \mod N$ et envoie x à *Bob*.
2. *Bob* envoie un bit aléatoire $b = 0$ ou 1 à *Alice*.
3. *Alice envoie* $y = r \times s^b \bmod N$ *à Bob* .
   - Si $b = 0$, $y = r$, donc *Bob* vérifie que $y^2 \equiv x \mod N$; cela prouve que *Alice* connaît $r = \sqrt{x}$
   - Si $b = 1$, $y = r \times s$, donc Victor vérifie **que** $y^2 \times v \equiv x \mod N$; cela prouve que *Alice* connaît s.

Si *Alice* ne connaît pas s, pour tromper *Bob* lorsque $b = 1$, elle doit lui envoyer $x = v^{-1}$; mais cela implique que $r = s$, donc *Alice* ne saura pas répondre si $b = 0$. Inversement, si *Alice* envoie effectivement une valeur x générée par $x \equiv r^2 \pmod N$, elle saura répondre lorsque $b = 0$ ($y = r$), mais elle ne pourra pas répondre lorsque $b = 1$, car il lui faudrait connaître s. Dans chaque cas, *Alice* a donc une chance sur deux de savoir répondre à la question.

1.10.2- Le concept de tiers de confiance

Beaucoup de protocoles cryptographiques, notamment ceux visant à sécuriser des environnements distribués, ont recours à la notion de *tiers de confiance*. Dans un tel contexte, pour établir une communication sécurisée, *Alice et Bob se font aider de Bill le référant, qui est une personne en qui ils ont confiance*. *Bill* va jouer un rôle dans la sécurisation des échanges entre *Alice* et *Bob* en participant à la mise en œuvre de mécanismes de sécurité, notamment en intervenant dans les protocoles d'authentification et d'échange de clef. …… De nombreux protocoles et systèmes ayant recours à des tiers de confiance ont vu le jour, parmi lesquels ***Kerberos,*** un protocole d'authentification réseau créé au **MIT** qui utilise un système de tickets au lieu de mots de passe en texte clair, et ***Sesame*** (*Secure European System for Applications in a Multi-vendor Environment*) qui étend ***Kerberos*** en ajoutant des services d'autorisation et d'accès.

1.11 - Echange de clef

Les deux méthodes les plus courantes d'établissement de clef sont le *transport* de clef et la *génération* de clef. Un exemple de transport de clef est l'utilisation d'un algorithme à clef publique pour chiffrer une clef de session générée aléatoirement. Un exemple de génération

de clef est le protocole *Diffie–Hellman* [Dif], qui permet de générer un secret partagé à partir d'informations publiques.

Le seul protocole décrit ici est le principe de génération de clef de *Diffie–Hellman* et les façons de l'étendre pour contrer les attaques de l'intercepteur. *Diffie–Hellman* est à la base de nombreux protocoles d'échange de clef.

1.11. 1- Diffie–Hellman

Inventé en 1976 par *Diffie* et *Hellman* ce protocole permet à deux tiers de *générer un secret partagé sans avoir aucune information préalable l'un sur l'autre.* Il est basé sur la cryptologie à clef publique (dont il est d'ailleurs à l'origine), car il fait intervenir des valeurs publiques et des valeurs privées. Sa sécurité dépend de la difficulté de calculer des logarithmes discrets sur un corps fini. Le secret généré à l'aide de cet algorithme peut ensuite être utilisé pour dériver une ou plusieurs clefs (clef secrète, *clef de chiffrement de clefs*,...).

Le déroulement de l'algorithme se présente comme suit :

1. *Alice* et *Bob* se mettent d'accord sur un grand entier n tel que $(n-1)/2$ soit premier et sur un entier g primitif par rapport à n. Ces deux entiers sont publics.
2. *Alice* choisit de manière aléatoire un grand nombre entier a, qu'elle garde secret, et calcule sa valeur publique, $A = g^a \pmod n$.
   Bob fait de même et génère b et $B = g^b \pmod n$.
3. *Alice* envoie A à *Bob* ; *Bob* envoie B à *Alice*.
4. *Alice* calcule $K_{AB} = B^a \pmod n$; *Bob* calcule $K_{BA} = A^b \bmod n$. $K_{AB} = K_{BA} = g^{ab} \pmod n$ est le secret partagé par *Alice* et *Bob*.

Une personne qui écoute la communication connaît g, n, $A=g^a \bmod n$ et $B=g^b \pmod n$, ce qui ne lui permet pas de calculer $g^{ab} \pmod n$: il lui faudrait pour cela calculer le logarithme de A ou B pour retrouver a ou b.

Toutefois, *Diffie–Hellman est vulnérable à l'attaque active dite «attaque de l'intercepteur»*

Une façon de contourner le problème de l'*attaque de l'intercepteur* sur *Diffie–Hellman* est *d'authentifier les valeurs publiques utilisées pour la génération du secret partagé*. On parle alors de ***Diffie–Hellman authentifié***.

L'authentification peut se faire à deux niveaux :

- En utilisant des valeurs publiques authentifiées, à l'aide de certificats par exemple. Cette méthode est notamment à la base du protocole ***SKIP*** (*Simple Key for IP*);
- En authentifiant les valeurs publiques après les avoir échangées, en les signant par exemple. Cette méthode est utilisée entre autres par le protocole ***Photuris*** (protocole développé par *Phil Karn* et *William Simpson* [Kar], basé sur la génération de secret partagé *Diffie-Hellman,* puis authentification avec les clés publiques).

L'inconvénient, dans les deux cas, est que l'on perd un gros avantage de *Diffie–Hellman*, qui est la possibilité de générer un secret partagé sans aucune information préalable sur l'interlocuteur.

1.11.2- Échange de clef et authentification mutuelle

Pour établir une communication sécurisée, on procède généralement, en premier lieu, à une authentification à des fins de contrôle d'accès. Puis, un échange de clef permet l'utilisation d'un mécanisme de sécurisation des échanges : l'authentification est ainsi étendue à la suite de la communication. L'exemple de *Diffie–Hellman* montre comme il est important que l'échange de clef soit authentifié. On parle alors de *protocole d'authentification mutuelle avec échange de clef.*

1.11.3- Propriétés des protocoles d'échange de clefs

Diffie, ***Van Oorschot*** et ***Wiener*** [Oor] ont défini le concept de protocole d'authentification mutuelle avec échange de clef sûr. Un protocole est dit sûr si les deux conditions suivantes sont valables dans chaque instance du protocole où l'un des deux tiers, par exemple *Alice*, exécute le protocole honnêtement et accepte l'identité de l'autre tiers :

- Au moment où *Alice* accepte l'identité de *Bob*, les enregistrements des messages échangés par les deux tiers se correspondent (i.e. les messages n'ont pas été altérés en route).
- Il est matériellement impossible pour toute personne autre que les tiers en présence de retrouver la clef échangée.

La propriété évoquée ci-dessus représente le minimum requis pour tout protocole. Cependant, d'autres propriétés des protocoles d'authentification avec échange de clef peuvent être envisagées :

- La propriété dite de ***Perfect Forward Secrecy (PFS)*** est garantie si *la découverte par un adversaire du ou des secrets à long terme ne compromet pas les clefs de session* générées précédemment : les clefs de session passées ne pourront pas être retrouvées à partir des secrets à long terme. On considère généralement que cette propriété assure également que *la découverte d'une clef de session ne compromet ni les secrets à long terme ni les autres clefs de session*……
A ce sujet, on parle d'attaque de ***Denning-Sacco*** [Den] lorsqu'un attaquant récupère une clef de session et utilise celle-ci, soit pour se faire passer pour un utilisateur légitime, soit pour rechercher les secrets à long terme (par *attaque exhaustive* ou *attaque par dictionnaire*).

- La propriété dite de ***Back Traffic Protection*** est fournie si la génération de chaque clef de session se fait de manière indépendante : les nouvelles clefs ne dépendent pas des clefs précédentes et *la découverte d'une clef de session ne permet ni de retrouver les clefs de session passées ni d'en déduire les clefs à venir*.

- On dit qu'il y a *authentification directe* (*Direct Authentication*) si, *à la fin du protocole, les valeurs servant à générer le secret partagé sont authentifiées ou si chaque tiers a prouvé qu'il connaissait la clef de session.* Par opposition, l'authentification est dite indirecte (*Indirect authentication*) si elle n'est pas garantie à la fin du protocole, mais dépend de la capacité de chaque tiers à utiliser, dans la suite des échanges, la ou les clefs mises en place précédemment.

- On parle de protection de l'identité (*Identity Protection*) lorsque le protocole garantit qu'un attaquant espionnant les échanges ne pourra pas connaître les identités des tiers communicants.

- Enfin, l'utilisation du temps (*Timestamps*) afin d'éviter la « re-jouabilité » est très controversée du fait, essentiellement, de sa trop grande dépendance d'horloges synchronisées.

1.11.4- Hiérarchie des clefs

On peut définir plusieurs types de clefs suivant leurs rôles :

- **Clefs de chiffrement de clefs**

  Situées en haut de la hiérarchie, ces clefs servent exclusivement à chiffrer d'autres clefs et ont généralement une durée de vie longue. On notera que leur utilisation, restreinte au chiffrement de valeurs aléatoires que sont des clefs, rend les attaques par cryptanalyse plus difficiles à leur niveau.

- **Clefs maîtresses**

  Les clefs maîtresses sont des clefs qui ne servent pas à chiffrer mais uniquement à *générer d'autres clefs par dérivation*. Une clef maîtresse peut ainsi être utilisée, par exemple, pour générer deux clefs : une pour le chiffrement et une pour la signature.

- **Clefs de session ou de chiffrement de données**

  D'une *durée de vie généralement faible* (elle peut aller jusqu'à changer à chaque message), une telle clef, par opposition aux précédentes, sert à chiffrer des données et se situe donc au bas de la hiérarchie. Du fait de leur lenteur, les algorithmes asymétriques sont très peu utilisés en chiffrement de données, et *les clefs de session sont donc généralement des clefs secrètes.*

Il est à noter que des abus de langage ont souvent lieu avec ces termes. Ainsi, on parle parfois de clef de session pour désigner en fait une clef maîtresse de même durée de vie et servant à générer plusieurs clefs : une pour l'authentification, une pour le chiffrement,...

1.12- Cryptanalyse

Si le but de la cryptographie est d'élaborer des méthodes de protection, le but de la cryptanalyse est au contraire de casser ces protections. Une tentative de cryptanalyse d'un système est appelée une attaque, et elle peut conduire à différents résultats :

- *Cassage complet* : le cryptanalyste retrouve la clef de déchiffrement.
- *Obtention globale* : le cryptanalyste trouve un algorithme équivalent à l'algorithme de déchiffrement, mais qui ne nécessite pas la connaissance de la clef de déchiffrement.
- *Obtention locale* : le cryptanalyste retrouve le texte en clair correspondant à un message chiffré.
- *Obtention d'information* : le cryptanalyste obtient quelques indications sur le texte en clair ou la clef (certains bits de la clef, un renseignement sur la forme du texte en clair,...)

D'une manière générale, on suppose toujours que le cryptanalyste connaît le détail des algorithmes, fonctions mathématiques ou protocoles employés. Même si ce n'est pas toujours le cas en pratique, il serait risqué de se baser sur le secret des mécanismes utilisés pour assurer

la sécurité d'un système, d'autant plus que l'usage grandissant de l'informatique rend de plus en plus facile la reconstitution de l'algorithme à partir du programme.

1.12.1- Attaques des fonctions de chiffrement

Les fonctions de chiffrement sont supposées rendre impossible le décryptement, c'est-à-dire la récupération d'un message clair sans la clef. A fortiori, elles doivent protéger le secret des clefs.

1.12.1.1- Classement des attaques en fonction des données dont dispose le cryptanalyste

On distingue plusieurs types d'attaques suivant les informations que peut obtenir le cryptanalyste. Ce sont :

- **L'attaque à texte chiffré seulement**, où le cryptanalyste ne connaît qu'un ensemble de textes chiffrés ; il peut soit retrouver seulement les textes en clair, soit retrouver la clef. En pratique, il est très souvent possible de deviner certaines propriétés du texte en clair (format ASCII, présence d'un mot particulier,...), ce qui permet de valider ou non le décryptement.

- **L'attaque à texte en clair connu**, où le cryptanalyste connaît non seulement les textes chiffrés, mais aussi les textes en clair correspondants ; son but est alors de retrouver la clef. Du fait de la présence, dans la plupart des textes chiffrés, de parties connues (en-têtes de paquets, champs communs à tous les fichiers d'un type donné,...), ce type d'attaques est très courant.

- **L'attaque à texte en clair choisi**, où le cryptanalyste peut, de plus, choisir des textes en clair à chiffrer et donc utiliser des textes apportant plus d'informations sur la clef. Si le cryptanalyste peut de plus adapter ses choix en fonction des textes chiffrés précédents, on parle d'**attaque adaptative**.

- **L'attaque à texte chiffré choisi**, qui est l'inverse de la précédente : le cryptanalyste peut choisir des textes chiffrés pour lesquels il connaîtra le texte en clair correspondant ; sa tâche est alors de retrouver la clef. Ce type d'attaques est principalement utilisé contre les systèmes à clef publique, pour retrouver la clef privée.

1.12.1.2- Attaques sur les algorithmes symétriques

Attaques au niveau des clefs

L'attaque la plus simple sur le plan conceptuel est l'**attaque exhaustive** (*exhaustiv search attack*) ou attaque en force brute, qui consiste à essayer toutes les clefs possibles. Avec une clef de 56 bits par exemple, cela nécessite 2^{56} tests.

Si la clef recherchée est en fait un mot de passe ou si elle dérive d'un mot de passe, on a des chances de trouver la clef en testant des mots susceptibles d'avoir été choisis comme mot de passe. On parle alors d'**attaque par dictionnaire**, car le cryptanalyste se constitue un dictionnaire de mots à tester (ce peut être une liste de prénoms, l'ensemble des mots d'une langue donnée,...). Ce type d'attaques est bien sûr beaucoup plus rapide qu'une attaque exhaustive.

Cryptanalyse différentielle

La **cryptanalyse différentielle** est un type d'attaques qui peut-être utilisé contre les algorithmes de chiffrement par blocs itératifs. Elle utilise une attaque à texte en clair choisi et se base sur l'observation de l'évolution des différences entre deux textes lorsqu'ils sont chiffrés avec la même clef. En analysant ces différences entre paires de textes, il est possible d'attribuer des probabilités à chaque clef possible. A force d'analyser des paires de textes, on finit soit par trouver la clef recherchée, soit par réduire suffisamment le nombre de clefs possibles pour pouvoir mener une attaque exhaustive rapide. La cryptanalyse différentielle peut également être utilisée pour attaquer d'autres types d'algorithmes comme les fonctions de hachage.

Ce type d'attaques a été utilisé pour la première fois par *Murphy* [Mur] en 1990 contre l'algorithme ***FEAL-4***. *Biham* et *Shamir* [Bih] ont mené des attaques différentielles sur le ***DES*** mais leur efficacité reste limitée du fait de la conception des ***S-Box***, qui avaient été optimisées contre ce type d'attaques. La meilleure attaque différentielle contre le ***DES*** complet à 16 rondes nécessite 2^{47} textes en clair choisis.

Cryptanalyse linéaire

La **cryptanalyse linéaire** utilise une attaque à texte en clair connu et consiste à modéliser l'algorithme de chiffrement par blocs par une approximation linéaire. Avec un nombre suffisant de paires (texte en clair, texte chiffré), on peut deviner certains bits de la clef.

La cryptanalyse linéaire fut utilisée pour la première fois en 1992 par *Matsui* et *Yamagishi* [Mat] pour attaquer, avec succès, ***FEAL*** (***Fast data Encipherment ALgorithm,*** proposé comme alternative au ***DES***). Elle fut étendue par *Matsui* en 1993 pour attaquer ***DES***. Les *S-Box* du ***DES*** ne sont pas optimisées pour contrer la cryptanalyse linéaire, et la meilleure attaque linéaire actuelle contre le ***DES*** nécessite 2^{43} textes en clair connus en moyenne. Une réalisation logicielle de cette attaque a trouver une clef ***DES*** en 50 jours avec 12 stations de travail.

En 1994, *Langford* et *Hellman* [Lan] introduisirent une attaque appelée cryptanalyse différentielle linéaire qui combine des éléments des deux méthodes précédentes.

1.12.2.3- Attaques sur les algorithmes asymétriques

Attaques au niveau des clefs

Avec les algorithmes asymétriques, le problème n'est pas de trouver la bonne clef par attaque exhaustive, mais *de dériver la clef secrète à partir de la clef publique*. La plupart des algorithmes asymétriques reposant sur des problèmes mathématiques difficiles à résoudre, *cela revient généralement à résoudre ce problème*. C'est pourquoi les paramètres qui influencent la difficulté de résolution du problème sont les éléments déterminant la sécurité. Généralement, cela se traduit par la nécessité d'utiliser de grands nombres, la taille de ces nombres ayant une répercussion sur la longueur des clefs. Cela explique que les clefs utilisées par la cryptographie à clef publique soient généralement bien plus longues que celles utilisées par la cryptographie à clef secrète.

Par exemple, dans le cas de ***RSA***, l'élément déterminant est la taille du module. La factorisation d'un module de 512 bits est à la portée d'une agence gouvernementale, 1024 bits est considéré comme moyennement sûr actuellement, et 2048 bits garantirait une sécurité à moyen terme.

Attaque à texte en clair deviné et problème de la faible entropie

Un point faible des algorithmes à clef publique est le caractère public de la clef de chiffrement : le cryptanalyste ayant connaissance de cette clef, il peut mener une *attaque à texte en clair deviné*, qui consiste à tenter de deviner le texte en clair et à le chiffrer pour vérifier son exactitude. Cette particularité implique donc une *restriction sur l'utilisation des algorithmes asymétriques :* il faut éviter de les utiliser avec un ensemble de textes en clair possibles restreint (*i.e.* de faible entropie). En effet, si tel était le cas, un attaquant pourrait aisément se constituer une liste exhaustive des textes en clair possibles et des textes chiffrés correspondants. Il n'aurait alors aucun mal à retrouver le texte en clair correspondant à un cryptogramme donné.

Attaque à texte chiffré choisi

D'autre part, la plupart des algorithmes asymétriques sont sensibles aux *attaques à texte chiffré choisi.* Il convient donc, si l'on utilise le même algorithme pour le chiffrement et pour la signature, de s'assurer que les protocoles employés ne permettent pas à un adversaire de faire signer n'importe quel texte. Mieux, on peut avoir recours à des couples (clef publique, clef privée) distincts pour ces deux opérations.

Attaque temporelle

Un type d'attaque apparu récemment est *l'attaque temporelle*, qui utilise la mesure du temps pris par des opérations cryptographiques pour retrouver les clefs privées utilisées. Cette attaque a été réalisée avec succès contre des cartes à microcircuits, des calculettes de sécurité et contre des serveurs de commerce électronique à travers l'*Internet*. La société *Counterpane Systems*, entre autres, a généralisé ces méthodes pour y inclure des *attaques sur des systèmes en mesurant la consommation, les émissions radioélectriques, et autres "canaux latéraux"* ; ils les ont mises en œuvre contre plusieurs types d'algorithmes à clef publique ou à clef secrète utilisés dans des calculettes. Une recherche voisine s'est intéressée à l'introduction délibérée d'erreurs dans les processeurs cryptographiques pour tenter d'obtenir des informations sur la clef.

Attaques des générateurs pseudo-aléatoires

Les générateurs pseudo-aléatoires sont supposés être imprévisibles. Un modèle d'attaque courant consiste à observer les aléas engendrés pendant un certain temps pour deviner la suite, voire l'état du générateur.

Attaques des fonctions de hachage à sens unique

Les deux attaques exhaustives de base que l'on peut mener contre une fonction de hachage à sens unique consistent à tester des entrées aléatoires avec la fonction jusqu'à :

- trouver une entrée qui donne en sortie une empreinte donnée (contredisant de ce fait la propriété "à sens unique") ;
- trouver deux entrées qui produisent la même sortie (contredisant de ce fait la propriété "sans collision").

Supposons que la fonction de hachage génère une empreinte de n bits. Si l'on veut trouver une entrée qui produira une valeur de sortie h donnée, alors, chaque sortie étant équiprobable, il faudra essayer 2^n valeurs possibles d'entrée pour avoir une bonne chance de trouver.

Attaque des anniversaires

Le paradoxe des anniversaires est un résultat célèbre du problème de probabilité suivant : combien faut-il de personnes dans une pièce pour qu'il y ait plus d'une chance sur deux pour que deux personnes au moins soient nées le même jour de l'année. La réponse est surprenante : 23 personnes suffisent ! D'une façon plus générale, considérons une fonction qui, lorsqu'on lui fournit une entrée aléatoire, retourne, de façon équiprobable, une valeur parmi k valeurs possibles (k grand). Si l'on applique cette fonction à différentes valeurs d'entrée, on a plus d'une chance sur deux de retomber sur une sortie déjà obtenue après avoir effectué un nombre d'essais de l'ordre de $\sqrt{k}$.

Si l'on applique ce résultat à une fonction de hachage générant une empreinte de n bits, il y a $k=2^n$ valeurs possibles en sortie, donc il faudra essayer seulement de l'ordre de $2^{n/2}$ valeurs d'entrées. Au lieu d'être en $O(2^n)$, la complexité de l'attaque n'est plus qu'en $O(2^{n/2})$!

L'attaque des anniversaires est surtout intéressante avec les algorithmes de signature basés sur une fonction de hachage : elle permet de faire signer à un tiers un message correct, mais auquel correspond un message frauduleux ayant la même empreinte. Il a été proposé la stratégie suivante :

- L'adversaire choisit un message frauduleux qu'il désire faire signer, et un message inoffensif qu'*Alice* acceptera sûrement de signer.
- L'adversaire génère $2^{n/2}$ variantes du message inoffensif (en faisant, par exemple, des modifications rédactionnelles mineures) et les empreintes correspondantes. Il génère ensuite un nombre égal de variantes du message frauduleux.
- La probabilité qu'une des variantes du message inoffensif et une des variantes du message frauduleux donnent la même empreinte est supérieure à 1/2 selon le paradoxe des anniversaires.
- L'adversaire fait alors signer la variante du message inoffensif à *Alice.*
- La signature du message inoffensif est alors retirée et attachée à la variante du message frauduleux qui donne la même empreinte. L'adversaire a donc forgé un faux message signé sans avoir besoin de connaître les clefs de chiffrement.

Pour éviter de telles attaques, il convient de choisir une longueur d'empreinte assez élevée. Les empreintes de 128 bits générées par ***MD5*** sont désormais trop justes ; les 160 bits de ***SHA-1*** et ***RIPE-MD-160*** sont plus sûrs.

1.12.2.4- Attaques des protocoles cryptographiques

On distingue deux types d'attaques sur les protocoles : les *attaques passives* et les *attaques actives*. Dans le premier cas, l'attaquant ne peut qu'espionner les données échangées par les tiers communicants, alors que dans le second cas il peut modifier ou supprimer des messages, en ajouter des nouveaux ou des anciens qu'il rejoue.

L'attaque par mascarade

On parle d'attaque par mascarade (*spoofing attack*) lorsqu'un attaquant **essaye de se faire passer pour un utilisateur légitime** en exécutant un protocole à la place de celui-ci. La conception d'un protocole a bien sûr pour but principal de contrecarrer ce type d'attaques.

La « replay attack »

Une attaque par « rejeu » (*replay attack*) consiste à *envoyer, dans une communication, des messages interceptés au cours d'une autre communication ou plus tôt dans la communication.* Ce type d'attaques permet de contourner des protocoles simples comme, par exemple, une authentification par mot de passe : il suffit à un adversaire d'avoir espionné un échange pour connaître le mot de passe et donc pour pouvoir se faire passer pour un utilisateur légitime. Les méthodes pour se prémunir contre ce type d'attaques sont l'utilisation de marquages temporels (*timestamps*) ou de défis imprévisibles et à usage unique.

Attaque par réflexion

Une attaque par réflexion (*reflection attack*) est une attaque pour laquelle *l'adversaire exploite le caractère symétrique d'un protocole* pour répondre aux défis de son interlocuteur en utilisant des réponses fournies par l'interlocuteur lui-même. Considérons par exemple le protocole d'authentification mutuelle suivant :

Pour se faire passer pour *Alice* auprès de *Bob*, il suffit à un adversaire d'établir deux connexions en parallèle avec *Bob*, et d'envoyer $défi_{A'} = défi_B$ comme second défi à *Bob*. Celui-ci fournira alors la réponse attendue à son premier défi :

Attaque dite de l'intercepteur

L'attaque de l'intercepteur (***man in the middle attack -MITM***) est une attaque qui a pour but d'intercepter les communications entre deux parties, sans que ni l'une ni l'autre ne puisse se douter que le canal de communication entre elles a été compromis. L'attaquant doit d'abord être capable d'observer et d'intercepter les messages d'une victime à l'autre. Cette attaque est particulièrement efficace à l'égard du protocole d'échange de clés *Diffie-Hellman*, quand il est utilisé sans authentification.

Le principe de cette attaque est que, pendant que deux tiers procèdent à un échange de clefs, *un adversaire se positionne entre les deux tiers et intercepte les échanges*. De cette façon, il procède à un échange de clef avec chaque tiers. A la fin du protocole, chaque tiers utilisera donc une clef différente, chacune de ces clefs étant connue de l'intercepteur. Pour chaque message transmis par la suite, l'intercepteur procédera à son déchiffrement avec la clef correspondante puis le re-chiffrera avec l'autre clef avant de l'envoyer à son destinataire: les deux tiers croiront communiquer de façon sûre alors que l'intercepteur pourra en fait lire tous les messages, voire même forger de faux messages.

Voici comment se déroule cette attaque dans le cas de *Diffie–Hellman :*

1. *Alice* envoie sa valeur publique $A = g^a \bmod n$ à Bob ; Ingrid l'intercepteur remplace cette valeur publique par la sienne. *Bob* reçoit donc $I = g^i \bmod n$.
2. *Bob* envoie sa valeur publique à *Alice* ; *Ingrid* remplace là aussi cette valeur par la sienne.
3. *Alice* génère le secret $K_{AI} = I^a \bmod n$; *Ingrid* génère le même secret en calculant $A^i \bmod n$
4. *Bob* génère le secret $K_{BI} = I^b \bmod n$; *Ingrid* génère le même secret en calculant $B^i \bmod n$

Alice et *Bob* croient tous les deux être en possession d'un secret partagé, mais en fait chacun d'eux partage un secret différent avec *Ingrid*.

Chapitre 2- FONDEMENTS MATHEMATIQUES DE L'AES

2.1- Introduction à la théorie des corps finis

2.1.1- Historique

L'étude des corps finis commence au début du 20ème siècle, avec les travaux de *Leonard Dickson* (1874-1954) puis de *Joseph Wedderburn* (1882-1948). *Dickson* publie la première classification des corps finis commutatifs. La question du cas non commutatif est alors l'objet d'une conjecture: *il n'existe pas de corps fini non commutatif*……
En 1905, *Wedderburn* démontre la conjecture qui devient le « théorème de *Wedderburn* ».

2.1.2- Principales propriétés

Comme **Z** est un anneau euclidien, a fortiori principal, tout idéal I s'écrit de manière unique sous la forme $I = n.Z$ où n est un entier appelé le générateur positif de I. Par définition, c'est le plus petit entier positif non nul appartenant à I. Les seules structures quotients obtenues à partir de Z sont donc les anneaux commutatifs unitaires $\mathbf{Z}/n.\mathbf{Z}$. Ces anneaux donnent les premiers exemples de corps finis. On a en effet le résultat suivant :

Théorème 2.1.2.1

L'anneau $\mathbf{Z}/n\mathbf{Z}$ *est un corps si, et seulement si,* n *est un nombre premier.*

Démonstration

Cette proposition est une conséquence directe de l'identité de *Bézout*.

Supposons n premier : alors si a est un entier premier avec n (c'est-à-dire non multiple de n), il existe deux entiers b et c tel que : $a\,b + n\,c = 1 \Leftrightarrow a\,b \equiv 1 \ (mod\ n)$ (Identité de *Bezout*)
Ce qui signifie que la classe de a est inversible, d'inverse la classe de b.

Réciproquement : si $n \in N^*$ n'est pas premier, il existerait alors deux entiers a et b différents de n et de 1 tels que $a.b = n$ i.e $a.b \equiv 0$ (mod n). La classe de a ainsi que la classe de b seraient des diviseurs de zéro, ce qui est impossible dans un corps. □

Soit **K** un corps fini et p sa *caractéristique*, c'est-à-dire le plus petit entier tel que l'addition de 1 itérée p fois soit égale à 0. Un tel nombre existe car sinon, le groupe additif engendré par 1 serait de cardinal infini alors que **K** ne contient pas de sous-ensemble infini.
De plus, p est un nombre premier. En effet, si a et b sont des entiers tels que $a.b = p$, alors le produit de a par b est nul dans **K**, ce qui montre que soit a soit b est un multiple de p, car il n'existe pas de diviseur de 0 dans un corps……
L'unique homomorphisme d'anneaux unitaires de Z dans **K** induit une injection de $Z/p.Z$ dans **K** , son image est le sous-corps de **K** formé des éléments du sous-groupe additif engendré par 1. C'est un sous-corps de **K** contenant p éléments et isomorphe à Z/pZ noté F_p .

Le corps **K** hérite donc d'une structure d'espace vectoriel sur F_p. Sa dimension, notée n, étant nécessairement finie son cardinal est alors p^n, on a ainsi établi que ;

Théorème 2.1.2.2

Le cardinal d'un corps fini est une puissance d'un nombre premier qui en est sa caractéristique. Plus exactement, tout corps fini est une extension finie d'un corps premier F_p.

Il existe des polynômes irréductibles $P(X)$ de $\boldsymbol{F_p}$ $[X]$ tel que le quotient $\boldsymbol{F_p}$ $[X]/(P(X))$ de l'anneau des polynômes $\boldsymbol{F_p}$ $[X]$ par l'idéal principal engendré par un tel polynôme irréductible $P(X)$ soit un corps fini (commutatif). Il contient strictement le corps premier d'ordre $\boldsymbol{p}$, si le polynôme n'est pas de degré un. Ce corps n'est autre qu'un corps de rupture du polynôme: la classe de X fournit une racine de P. Ici X désigne une indéterminée et $P(X)$ un polynôme formel.

Automorphisme de Frobenius

On suppose ici donné un corps **K** fini de caractéristique $\boldsymbol{p}$, de cardinal $\boldsymbol{q = p^k}$.

Il est bien connu que : $\boldsymbol{(x+y)^p = x^p + y^p}$

- L'application $\boldsymbol{Frob : x \in K \rightarrow x^p \in K}$ est un endomorphisme bijectif, appelé *endomorphisme de Frobenius* et est noté : ***Frob***.

L'étude de l'automorphisme de *Frobenius* permet aussi de déterminer les polynômes irréductibles sur le corps fini **K** : il est connu en effet que :

Proposition 2.1.2.3

Les polynômes irréductibles de $\boldsymbol{K}$ *de degré* $\boldsymbol{n}$ *sont les diviseurs premiers de degré* $\boldsymbol{n}$ *du polynôme suivant et sont donc tous cyclotomiques :* $\boldsymbol{X^{p^n} - X}$

Existence d'un corps fini de cardinal p^n

Soit n un entier strictement positif et $\boldsymbol{q}$ l'entier égal à $\boldsymbol{p^n}$.
Soit aussi $\boldsymbol{L}$ le corps de décomposition du polynôme $P(X) = X^q - X$ sur le corps $\boldsymbol{F_p}$.
L'ensemble des éléments invariants par l'automorphisme $\boldsymbol{Frob^n}$ est une extension $\boldsymbol{K}$ de $\boldsymbol{F_p}$;
$\boldsymbol{K}$ est exactement l'ensemble des racines de $P(X)$. Or $P(X)$ est scindé sur $\boldsymbol{K}$, il est de plus séparable car premier avec son polynôme dérivé égal à $\boldsymbol{1}$. $\boldsymbol{K}$ contient donc autant de racines que son degré, c'est-à-dire $\boldsymbol{p^n}$. □

Une question importante sera celle de la structure du groupe multiplicatif des corps finis.

Groupe multiplicatif et conséquences

Soit $\boldsymbol{K}$ un corps fini de cardinal $\boldsymbol{q = p^n}$. Alors, le groupe multiplicatif de ce corps est de cardinal $\boldsymbol{q - 1}$; Soit $\boldsymbol{e}$ son exposant, c'est un diviseur de $(\boldsymbol{q - 1})$. Il vient alors que le polynôme $\boldsymbol{X^e - 1}$ à coefficients dans $\boldsymbol{K}$, possède $(\boldsymbol{q - 1})$ racines distinctes et est de degré $\boldsymbol{e}$, ce qui implique que $\boldsymbol{e}$ est supérieur à $\boldsymbol{q - 1}$, puis l'égalité. L'exposant du groupe multiplicatif étant égal à son ordre, et comme dans un groupe fini commutatif l'exposant est toujours l'ordre d'un élément du groupe, on peut affirmer que :

Proposition 2.1.2.4

- *Le groupe multiplicatif d'un corps fini :* $\boldsymbol{F_q = Z/qZ^*}$ $(\boldsymbol{q=p^n}$, $\boldsymbol{p}$ premier) *est cyclique d'ordre* $(\boldsymbol{q-1})$

En conséquence, tout corps de cardinal q est un corps de décomposition du polynôme $X^q - X$ sur F_p. Deux tels corps de décomposition sont isomorphes, donc si on fixe une clôture algébrique de F_p, alors :

- Il existe un et un seul corps de cardinal q, noté F_q, contenu dans la clôture algébrique fixée.

Soit K un corps commutatif et $K[X]$ l'anneau des polynômes à coefficients dans K. Un des nombreux objectifs de la théorie de Galois est l'étude de l'équation polynomiale $P(X) = 0$. Si P est un polynôme irréductible, on recherche des solutions dans une extension algébrique L de K. Un cas particulier largement utilisé est l'anneau $K[X]/(P)$ des polynômes quotientés par l'idéal engendré par $P(X)$. Comme $P(X)$ est irréductible, l'idéal engendré par $P(X)$ est maximal, l'anneau quotient est bien un corps.

Cette technique permet de construire tous les corps finis. Soit K un corps fini, il existe toujours p un nombre premier et n un entier positif tel que le cardinal de K soit égal à p^n. La valeur p correspond à la caractéristique de K. Si $P(X)$ est un polynôme irréductible à coefficients dans l'anneau Z/pZ qui est aussi un corps noté F_p, alors le corps L est isomorphe au quotient de l'anneau $F_p[X]$ par l'idéal engendré par $P(X)$.

Remarque *:* Soit A un anneau commutatif. Il est bien connu que l'application h : $h : A \to A/I$ définie par : $h(x) = x + I$ est un homomorphisme surjectif d'anneau dont le noyau est l'idéal I *(bilatère) ;*

Alors :

- I est premier si et seulement si A / I est un anneau intègre.
- I est maximal si et seulement si A / I est un corps.

On peut démontrer directement les assertions développées plus haut :

Proposition 2.1.2.5

$K[X]/P[X]$ est un corps si et seulement si P est irréductible dans $K[X]$

Démonstration

$\Rightarrow$ Supposons que $K[X]/P[X]$ soit un corps : si $P=AB$, alors $\overline{A}\,\overline{B} = \overline{AB} = \overline{P} = \overline{0}$, et puisque tout corps est intègre, alors $\overline{A}$ ou $\overline{B}$ vaut $\overline{0}$: par exemple si $\overline{A} = \overline{0}$, alors $P \mid A$. Puisque on a pris au départ deg $P \geq 1$, P est non nul et donc A non plus (car $P=AB$), ce qui implique que deg $A \geq$ deg P = deg A + deg B, d'où deg $B \leq 0$, c'est-à-dire deg $B = 0$ (car $B \neq 0$ puisque $P = AB$), et nécessairement B est constant : P est donc bien irréductible dans $K[X]$.

Réciproquement, supposons que P soit irréductible dans $K[X]$. D'abord $\overline{1} \neq \overline{0}$ car sinon, $\overline{P} = \overline{0} = \overline{1} \Rightarrow P \mid 1 \Rightarrow P$ constant, ce qui est absurde. Maintenant si $\overline{A}$ est un élément non nul de $K[X]/P[X]$, alors $\overline{A} \neq \overline{0}$ ce qui veut dire que P ne divise pas A ou que P est premier avec A: il existe donc 2 polynômes U et V dans $K[X]$ tels que: $AU + PV = 1$ en sorte que : $\overline{AU + PV} = \overline{1} = \overline{A}\,\overline{U} + \overline{P}\,\overline{V} = \overline{A}\,\overline{U} + \overline{0}\,\overline{V} = \overline{A}\,\overline{U}$, ce qui signifie que A est inversible et donc $K[X]/P[X]$ est bien un corps

□

2.2 - Les Mathématiques de l'AES

2.2.1- Rappels: construction d'un corps fini à p^m éléments

On considère l'anneau $Z_2[x]$ des polynômes à une indéterminée à coefficients dans $Z_2 = Z/2Z$. On rappelle que $Z_2 \sim \{0, 1\}$ est muni des deux opérations suivantes :

+	0	1
0	0	1
1	1	0

×	0	1
0	0	0
1	0	1

Proposition 2.1.2.6 (voire l'annexe)

Le polynôme $m(x) = x^8 + x^4 + x^3 + x + 1$ est irréductible dans $Z_2[x]$, car $m(0)=1$ et $m(1)=1$, et donc l'ensemble des classes résiduelles modulo $m(x)$ des éléments de $Z_2[x]$, que l'on note :

$\frac{Z_2[x]}{m(x)\,Z_2[x]}$, est un corps: c'est le *corps de* **Galois** $GF(2^8)$.

On rappelle que si $A(x)$ est un polynôme de $Z_2[x]$, il existe deux polynômes $B(x) \in Z_2[x]$ et $R(x) \in Z_2[x]$, *uniques*, tels que : $A(x) = B(x)\,m(x) + R(x)$, avec $d°R < d°m$, ce qui équivaut à dire que : $A(x) \equiv R(x)$ **mod.** $m(x)$ *:* $R(x)$ est la classe résiduelle modulo $m(x)$ de $A(x)$

Comme $m(x)$ est de degré 8, $\frac{Z_2[x]}{m(x)\,Z_2[x]} \cong GF(2^8)$ est donc constitué des polynômes de $Z_2[x]$ de degré au plus égal à 7, et à coefficients dans $Z/2Z=F_2$:

$$GF(2^8) \cong \left\{ \sum_{i\geq 0} a_i X^i \bmod m(X),\ a_i \in \{0,1\} \right\} \cong \left\{ \sum_{i=0}^{i=7} a_i X^i,\ a_i \in \{0,1\} \right\}$$

En pratique, on représentera le polynôme $A(X) = a_7 X^7 + \ldots + a_1 X + a_0$ par l'octet A={ $a_7 a_6 \ldots a_1 a_0$ } : on dira qu'on utilise une notation polynomiale des éléments de $GF(2^8)$.

Exemple

1) Le polynôme $X_Z = x^6 + x^4 + x^3 + x \in GF(2^8)$ peut être identifié à l'octet : $(\mathbf{01011010})_2$ en notation binaire, qui représente la lettre Z selon la table de correspondance **ASCII**, ***8-bits*** «***The extended ASCII table*** » (*cf Annexe n° 1*).

2) Le polynôme $X_B = x^6 + x$ peut être identifié à l'octet: $(\mathbf{01000010})_2$ qui représente la lettre ***B***, selon toujours la table de correspondance ASCII 8-bits.

En fait, l'***AES*** fonctionne essentiellement avec le système ***Hexadécimal***: ***0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.***

Ainsi : $(\mathbf{01011010})_2$ s'écrit *0x*5A, *0x* n'étant qu'un préfixe qui indique qu'on est dans le système *Hexadécimal*: de cette façon on utilise seulement 2 bits (*pour 5A*) au lieu de 8 bits (*pour 01011010*), soit 4 fois moins, ce qui est plus économique.

2.2.1.1-Les opérations de $GF(2^8)$

Addition dans $GF(2^8)$

L'addition de deux éléments de $F_2 = Z/2Z$ correspond à l'opération *XOR* $(\oplus)$. Ainsi, l'addition de deux éléments ***A(X)*** et ***B(X)*** de $GF(2^8)$ correspond à l'addition de deux polynômes à coefficients dans $F_2 = Z/2Z$:

$$A(x) = a_7 x^7 + a_6 x^6 + ... + a_1 x + a_0$$
$$B(x) = b_7 X^7 + b_6 X^6 + ... + b_1 X + b_0$$
$$\Rightarrow \quad A(x) \oplus B(x) = (a_7 \oplus b_7)x^7 + (a_6 \oplus b_6)x^6 + ... + (a_1 \oplus b_1)x + (a_0 \oplus b_0)$$

Exemple

Soient les polynômes de $GF(2^8)$: $X_Z(x) = x^6 + x^4 + x^3 + x$ et $X_B(x) = x^6 + x$;
On définit l'addition $\oplus$ dans $GF(2^8)$, en posant:
$\Rightarrow$ $X_Z(x) \oplus X_B(x) = (x^6 + x^4 + x^3 + x) + (x^6 + x) = x^4 + x^3$, + étant l'addition dans $Z_2[x]$.

En identifiant X_Z et X_B aux octets correspondants, on peut écrire:

$(x^6 + x^4 + x^3 + x) + (x^6 + x) = x^4 + x^3$	(notation polynomiale)
$(01011010)_2 + (01000010)_2 = (00011000)_2$	(notation binaire : *ASCII 8 bits)*
$\{0x5A\} + \{0x42\} = \{0x18\}$	(notation hexadécimale)

Multiplication dans $GF(2^8)$

Soient ***A(x)*** et ***B(x)*** 2 polynômes de $GF(2^8)$: on note $\otimes$ l'opération de multiplication du *corps de* ***Galois*** $GF(2^8)$
Nous explicitons le produit $A(x) \otimes B(x) \in GF(2^8)$; deux cas sont à considérer :

- **Si** $deg(A) + deg(B) \leq 7$: alors $A(x) \otimes B(x) = A(x)$ x $B(x)$, x étant la multiplication dans $Z_2[x]$

- **Si** $deg(A) + deg(B) \geq 8$: alors $A(x) \otimes B(x) = R(x)$,

  où $R(x)$ est la classe résiduelle du produit $A(x)$ x $B(x)$ (dans $Z_2[x]$) modulo $m(x)$

Exemple **:** calculons $X_Z(x) \otimes X_B(x)$; on a successivement :

$X_Z(x) \otimes X_B(x) = (x^6 + x^4 + x^3 + x)(x^6 + x)$, et en développant dans $Z_2[x]$

$X_Z(x) \times X_B(x) = x^{12} + x^{10} + x^9 + x^7 + x^7 + x^5 + x^4 + x^2 = x^{12} + x^{10} + x^9 + x^5 + x^4 + x^2 \notin GF(2^8)$

On écrit alors que :

$x^{12} + x^{10} + x^9 + x^5 + x^4 + x^2 = (x^4 + x^2 + x + 1) \times (x^8 + x^4 + x^3 + x + 1) + (x^7 + x^6 + x^2 + 1)$ et

donc : $X_Z(x) \times X_B(x) \equiv x^7 + x^6 + x^2 + 1 \pmod{m(x)}$, et par définition même :

$X_Z(x) \otimes X_B(x) = x^7 + x^6 + x^2 + 1 \in GF(2^8)$ s'écrit ‹ 11000101 ›$_2$ s'écrit Å s'écrit *0xC5* (*ASCII 8 bits*)

Remarque : En notation hexadécimale, le polynôme ***m(X)*** s'écrit **0*x*11B**.

Calcul de l'inverse d'un élément de $GF(2^8)$

Soit $X(x) = a_7x^7 + a_6x^6 + a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0$ s'écrit $(a_7a_6a_5a_4a_3a_2a_1a_0)_2$, $a_i = 0$ ou $\mathbf{1}$

Il s'agit de calculer $X^{-1}(x)$ au sens de la loi du groupe multiplicatif $\otimes$ du corps $GF(2^8)$.

On utilise l'***algorithme étendu d'Euclide***, en calculant $U(x)$ et $V(x)$ deux polynômes de $GF(2^8)$ tels que : $X(x) \times U(x) + m(X) \times V(x) = 1$ *(E)*

Les polynômes $U(x)$ et $V(x)$ existent toujours, car $m(x)$ est premier à tout polynôme de $GF(2^8)$

Alors : $X(x) \times U(x) \equiv 1 \pmod{m(x)}$ ce qui implique : $X(x) \otimes U(x) = 1$, *i.e.* $X^{-1}(x) = U(x)$ dans $GF(2^8)$.

Algorithme pour la résolution de l'équation (*E*)

Si ***A*** *et* ***B*** *sont deux polynômes premiers entre eux i.e. PGCD (A, B)=1, et si* $q_1, q_2 \ldots q_t$ *est la suite des quotients obtenus en utilisant l'algorithme d'***Euclide** *:*

$A = B\,q_k + r_k$, $0 \leq \deg r_k < \deg B$, *on construit la matrice :*

		q_1	q_2	q_3	…… …	q_{t-1}	q_t
0	1	P_1	P_2	P_3	…… …	P_{t-1}	A
1	0	Q_1	Q_2	Q_3	……	Q_{t-1}	B

Avec : $P_1 = q_1$, $Q_1 = 1$; $P_2 = q_2$ $P_1 + 1$, $Q_2 = q_2 Q_1$ *; et pour* $k \geq 3$ *:*

$P_k = q_k P_{k-1} + P_{k-2}$ *et* $Q_k = q_k Q_{k-1} + Q_{k-2}$; *alors :*

$A\,Q_{t-1} - B\,P_{t-1} = (-1)^t$ *, d'où la solution de l'équation (E) :*

$$\begin{cases} U : (-1)^t Q_{t-1} \\ V : (-1)^{t+1} P_{t-1} \end{cases}$$

□

illustrer cette procédure à travers un exemple :

Calculer l'inverse de $X_o = x^6 + x^5 + x^3 + x^2 + x + 1$ *s'écrit* $(01101111)_2$ *s'écrit* $0x\ 6F$

On initialise la procédure comme pour le calcul du PGCD (m(x), X_o) :

$$x^8 + x^4 + x^3 + x + 1 = (x^6 + x^5 + x^3 + x^2 + x + 1)(x^2 + x + 1) + (x^4 + x^2 + x)$$
$$x^6 + x^5 + x^3 + x^2 + x + 1 = (x^4 + x^2 + x)(x^2 + x + 1) + (x^3 + x^2 + 1)$$
$$x^4 + x^2 + x = (x^3 + x^2 + 1)(x + 1) + 1 \rightarrow PGCD\,(X_O, m(x))$$
$$x^3 + x^2 + 1 = 1(x^3 + x^2 + 1) + 0$$

On utilise ensuite l'algorithme précédent :

		x^2+x+1	X^2+x+1	$x+1$	x^3+x^2+1
0	*1*	x^2+x+1	x^4+x^2	$X^5+x^4+x^3+x+1 = U(x)$	$X^8+x^4+x^3+x+1 = m(x)$
1	*0*	*0*	x^2+x+1	$X^3 = -V(x)$	$X^6+x^5+x^3+x^2+x+1 = X_o$

On vérifie que :

$(x^6 + x^5 + x^3 + x^2 + x + 1)(x^5 + x^4 + x^3 + x + 1) - (x^8 + x^4 + x^3 + x + 1)x^3 = 1$, et donc :
$(x^6 + x^5 + x^3 + x^2 + x + 1)(x^5 + x^4 + x^3 + x + 1) \equiv 1 \mod m(x)$,
en sorte que, au sens de la structure de corps de $GF(2^8)$, on a :

$(x^6 + x^5 + x^3 + x^2 + x + 1)^{-1} = (x^5 + x^4 + x^3 + x + 1)$ *s'écrit* $(00111011)_2$ *s'écrit* $0x3B$

2 .3- Méthodes pour multiplier deux éléments de $GF(2^8)$

En pratique, pour réaliser cette multiplication, on dispose de deux méthodes.

2.3.1- Méthode "lente" : multiplication par x^i .

Cette méthode consiste à réaliser effectivement la multiplication polynomiale.

Soit $A \in GF(2^8)$. En notation polynomiale : $A(x) = a_7 x^7 + a_6 x^6 + ... + a_1 x + a_0$, $a_i \in F_2$

Ainsi, $x\,A(x) = a_7 x^8 + a_6 x^7 ... + a_1 x^2 + a_0 x$. Il reste à effectuer la réduction modulo $m(x)$.

- Si $a_7 = 0$, on obtient l'expression réduite: $x\,A(x) = a_6 x^7 + ... + a_1 x^2 + a_0 x \in GF(2^8)$

- Si $a_7 = 1$: et dans ce cas : $x\,A(x) = x^8 + a_6 x^7 + ... + a_1 x^2 + a_0 x$. Comme $m(x) = x^8 + x^4 + x^3 + x + 1 \equiv 0 \mod m(x)$, alors : $x^8 \equiv x^4 + x^3 + x + 1 \mod m(x)$ et donc :

$$x^8 + a_6 x^7 + \ldots + a_1 x^2 + a_0 x = (a_6 x^7 + \ldots + a_1 x^2 + a_0 x) \oplus (x^4 + x^3 + x + 1)$$

$$= a_6 x^7 + a_5 x^6 + a_4 x^5 + (a_3 \oplus 1) x^4 + (a_2 \oplus 1) x^3 + a_1 x^2 + (a_0 \oplus 1) x + 1$$

En notation polynomiale, cette opération consiste donc à un décalage à gauche suivi éventuellement d'un *XOR* bit-à-bit avec {1B}.

Soit $x\,time$ la fonction qui réalise l'opération $x.\,A(x) \bmod m(x)$; $x\,time$ peut être utilisée pour définir la fonction $x^i\,time$ qui réalise l'opération $x^i . A(x) \bmod m(x)$. Enfin, à partir de cette dernière fonction, il est possible de réaliser la multiplication complète de deux éléments de $GF(2^8)$.

2.3. 2- Méthode "rapide" à partir d'une représentation exponentielle.

Soit $w(x)$ un générateur de $GF(2^8) = \dfrac{F_2[x]}{m(x) F_2(x)}$: dans ce cas, tout élément non nul de $GF(2^8)$ se représente de manière unique par $(w(x))^i \bmod m(x)$, $0 \le i < 255$.

On obtient ainsi une autre écriture pour $GF(2^8)$:

$$GF(2^8) = \{0\} \cup \{(w(x))^i \bmod m(x)\}_{0 \le i < 255}$$

Avec cette représentation (dite cyclique ou exponentielle), la multiplication de deux éléments A et B non nuls devient facile:

$A(x) = (w(x))^i$

$B(x) = (w(x))^j$

Alors : $A(x) . B(x) = (w(x))^{i+j \bmod 255}$

L'idée consiste alors à passer par cette représentation pour effectuer la multiplication. On utilise pour cela des tables de correspondance entre les éléments $A(x)$ de $GF(2^8)$ et l'exposant i correspondant à sa représentation exponentielle.

Le générateur le plus simple pour le polynôme irréductible $m(x)$ ⇔ **0x11B** de $\mathbf{Z_2[x]}$ est :

$w(x) = x+1$ ⇔ **0x03** (en notation hexadécimale)

Ainsi, à partir de la fonction de multiplication lente, on génère une table notée *ExpoToPoly* de 256 entrées telle que *ExpoToPoly[k]* donne la représentation polynomiale de $w(x)^k$ mod $m(x)$ (On représentera l'élément 0 par $w(x)^{255}$ bien que mathématiquement, $w(x)^{255} = 1$) La table *PolyToExpo* correspond à la table inverse. Ces tables sont fournies en annexe.

On utilise ces tables pour effectuer efficacement la multiplication de deux éléments A et B de $GF(2^8)$.

Les polynômes à coefficients dans GF(2⁸)

Tout comme un octet peut être représenté par un polynôme de degré 7, un mot de 32 bits (4 octets) peut l'être par un polynôme de degré 3 à coefficients dans $GF(2^8)$, chaque coefficient représentant un octet du mot.

Exemple:

$$
\begin{array}{ll}
 & \texttt{0x5b 0x99 0xb3 0xe7} \\
\Leftrightarrow & 01011011_2\ 10011001_2\ 10110011_2\ 11100111_2 \\
\Rightarrow & (01011011_2)X^3 + (10011001_2)X^2 + (10110011_2)X + (11100111_2) \\
\Leftrightarrow & a_1X^3 + a_2X^2 + a_1X + a_0 = a(X)
\end{array}
$$

L'addition de deux mots est alors égale à l'addition des polynômes les représentants, soit un ou-exclusif entre les coefficients de même degré.

$$a(X) + b(X) = (a_3 \oplus b_3)X^3 + (a_2 \oplus b_2)X^2 + (a_1 \oplus b_1)X + (a_0 \oplus b_0)$$

La multiplication de deux mots donne un polynôme de degré 3+3 dont on peut calculer les coefficients par la définition précédente et la définition générale du produit de polynômes.

$$a(X) \times b(X) = c(X) = c_6X^6 + c_5X^5 + c_4X^4 + c_3X^3 + c_2X^2 + c_1X + c_0$$

$$
\begin{array}{lcl}
c_0 & = & a_0 \otimes b_0 \\
c_1 & = & a_1 \otimes b_0 \oplus a_0 \otimes b_1 \\
c_2 & = & a_2 \otimes b_0 \oplus a_1 \otimes b_1 \oplus a_0 \otimes b_2 \\
c_3 & = & a_3 \otimes b_0 \oplus a_2 \otimes b_1 \oplus a_1 \otimes b_2 \oplus a_0 \otimes b_3 \\
c_4 & = & a_3 \otimes b_1 \oplus a_2 \otimes b_2 \oplus a_1 \otimes b_3 \\
c_5 & = & a_3 \otimes b_2 \oplus a_2 \otimes b_3 \\
c_6 & = & a_3 \otimes b_3
\end{array}
$$

Afin de rester dans l'espace des mots de 32 bits, on considère les polynômes **c(X)** *modulo* un polynôme de degré ***4*** qui vaut pour l'**AES**: $M(X) = X^4+1$, formant un groupe que l'on pourrait noter : $A = \frac{GF(2^8)[X]}{(X^4+1)GF(2^8)[X]}$

Notons alors : $d(X) = d_3X^3 + d_2X^2 + d_1X + d_0 = c(X) \bmod (X^4+1) = a(X) \otimes b(X)$, le produit modulaire dans $\mathcal{A}$ des polynômes $a(X)$ et de $b(X)$ à coefficients dans $GF(2^8)$.

On sait par ailleurs que : $X^j \bmod (X^4+1) = X^{j \bmod 4}$, et donc :

$c(X) \bmod (X^4+1) = [c_6X^6 + c_5X^6 + c_4X^4 + c_3X^3 + c_2X^2 + c_1X + c_0] \bmod (X^4+1)$ ou :

$$d(X) = c(X) \bmod (X^4+1) = \sum_{j=0}^{j=6} c_j X^j \bmod (X^4+1) = \sum_{j=0}^{j=6} c_j X^{j \bmod 4}$$

$$d(X) = c_6X^2 + c_5X + c_4 + c_3X^3 + c_2X^2 + c_1X + c_0$$

$$d(X) = c_3X^3 + (c_2 + c_6)X^2 + (c_1 + c_5)X + (c_0 + c_4)$$

$$= d_3X^3 + d_2X^2 + d_1X + d_0$$, et par simple identification et compte tenu des formules précédentes donnant les c_j , on obtient finalement :

$$\begin{aligned} d_0 &= a_0 \otimes b_0 \oplus a_3 \otimes b_1 \oplus a_2 \otimes b_2 \oplus a_1 \otimes b_3 \\ d_1 &= a_1 \otimes b_0 \oplus a_0 \otimes b_1 \oplus a_3 \otimes b_2 \oplus a_2 \otimes b_3 \\ d_2 &= a_2 \otimes b_0 \oplus a_1 \otimes b_1 \oplus a_0 \otimes b_2 \oplus a_3 \otimes b_3 \\ d_3 &= a_3 \otimes b_0 \oplus a_2 \otimes b_1 \oplus a_1 \otimes b_2 \oplus a_0 \otimes b_3 \end{aligned}$$

Ces opérations peuvent être mises sous forme matricielle.

$$\begin{pmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{pmatrix} = \begin{pmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix}$$

Comme X^4+1 n'est pas un polynôme irréductible, $\mathcal{A}$ n'est pas un corps et ses éléments ne sont pas forcément inversibles. L'***AES*** utilise alors pour ses calculs sur les mots le polynôme ***a(X)*** inversible dans $\mathcal{A}$.

$$\begin{aligned} a(X) &= (0\text{x}03)X^3 + (0\text{x}01)X^2 + (0\text{x}02)X + (0\text{x}02) \\ a^{-1}(X) &= (0\text{x}0\text{b})X^3 + (0\text{x}0\text{d})X^2 + (0\text{x}09)X + (0\text{x}0\text{e}) \end{aligned}$$

Multiplication par x

Soit $b(x) = b_3x^3 + b_2x^2 + b_1x + b_0 \Rightarrow xb(x) = b_3x^4 + b_2x^3 + b_1x^2 + b_0x$,

Alors : $x \otimes b(x)$ est obtenu en réduisant $xb(x)$ modulo $(x^4 + 1)$, ce qui donne :

$c(x) = x \otimes b(x) = b_2x^3 + b_1x^2 + b_0x + b_3$

Que l'on peut écrire sous forme matricielle :

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 00 & 00 & 00 & 01 \\ 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Cette procédure simplifie notablement l'opération de multiplication par *x*.

Chapitre 3- DESCRIPTION ET FONCTIONNEMENT DE L' ALGORITHME AES

3.1- Entrées et Sorties

Les entrées et les sorties de l'***AES*** consistent en des séquences de 128 bits. La clé secrète de chiffrement est une suite de 128, 192 ou 256 bits.
Un bloc peut être représenté comme une matrice d'octets 4x**Nb** (*State*). Les octets lus en entrée y sont copiés colonne après colonne (chaque colonne représente un mot lu), selon la matrice suivante :

$$\begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix}$$

$\{a_{0,j}, a_{1,j}, a_{2,j}, a_{3,j}\}$ représente un mot w_j, $j=0,1,2$ ou 3 ; plus précisément :

$$w_0 = \begin{Bmatrix} a_{0,0} \\ a_{1,0} \\ a_{2,0} \\ a_{3,0} \end{Bmatrix} ; \quad w_1 = \begin{Bmatrix} a_{0,1} \\ a_{1,1} \\ a_{2,1} \\ a_{3,1} \end{Bmatrix} ; \quad w_2 = \begin{Bmatrix} a_{0,2} \\ a_{1,2} \\ a_{2,2} \\ a_{3,2} \end{Bmatrix} ; \quad w_3 = \begin{Bmatrix} a_{0,3} \\ a_{1,3} \\ a_{2,3} \\ a_{3,3} \end{Bmatrix}$$

Figure n° 1 : Structure d'un bloc d'octets

3.2- Notation, structure de données

Les conventions de notation issues de la spécification du *NIST* sont les suivantes :

AddRoundKey() : transformation (pour le chiffrement et le déchiffrement) qui ajoute une clé de tour (*Round key*) au bloc (*State*) courant.

MixColumns() : transformation qui permute les colonnes d'un bloc lors du chiffrement.

RotWord() : fonction utilisée par la routine d'expansion de la clé qui applique à un mot de 4 octets une permutation circulaire.

ShiftRows() : transformation qui applique des permutations circulaires aux trois dernières lignes du bloc lors du chiffrement.

SubBytes() : transformation qui opère une substitution non linéaire de chaque octet du bloc en utilisant une table (***S-box***) lors du chiffrement.

SubWord()	:	fonction utilisée par la routine d'expansion de la clé qui prend un mot en entrée et applique à ses 4 octets une substitution non linéaire (***S-box***)
InvMixColumns()	:	transformation du déchiffrement qui est l'inverse de la transformation **MixColumns()**.
InvShiftRows()	:	transformation du déchiffrement qui est l'inverse de la transformation **ShiftRows()**.
InvSubBytes()	:	transformation du déchiffrement qui est l'inverse de la transformation **SubBytes()**.
K	:	la clé de chiffrement.
Nb	:	Nombre de colonnes (mots de 32 bits) du bloc. Pour l'***AES*** : **Nb=4**.
Nk	:	nombre de mots de 32 bits dans la clé de chiffrement. Pour ***AES:*** **Nk= 4,6 ou 8**.
Nr	:	Nombre de tours, fonction de **Nb** et **Nk**. Pour l'***AES*** **Nr=10,12 ou 14**.
Rcon[]	:	table constante.

3.3- Description de l'algorithme de chiffrement *A.E.S*

L'algorithme ***AES*** n'est pas exactement celui de *Rijndael* dans la mesure où ce dernier supporte des tailles de blocs plus nombreux qu'***AES***. L'*Advanced Encryption Standard* fixe la taille de blocs à 128 bits: **Nb=4** reflète le nombre de mots de 32 bits dans un tel bloc (c'est aussi le nombre de colonnes nécessaire pour une représentation matricielle).

AES utilise des **clés de 128, 192 ou 256 bits**. La longueur de clé est caractérisée de façon similaire par **Nk=4, 6** ou **8**.
AES exécute une séquence de rondes qui seront détaillés dans la suite. On note **Nr** le nombre de rondes qui doivent être effectuées. Ce nombre dépend des valeurs de **Nb** et de **Nk**. Les différentes configurations possibles sont détaillées dans le tableau suivant :

Détail des configurations possibles

	Nk	**Nb**	**Nr**
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Comme on l'a déjà indiqué, ***AES*** opère sur une matrice 4 x **Nb** d'éléments de $GF(2^8)$, notée « *State* ». Le chiffrement ***AES*** consiste en une addition initiale de clé, notée *AddRoundKey*, suivie par (**Nr-1**) rondes, chacune constitué de quatre étapes :

1. ***SubBytes*** : il s'agit d'une substitution non-linéaire lors de laquelle chaque octet est remplacé par un autre octet choisi dans une table particulière (***S-Box***).
2. ***ShiftRows*** est une étape de transposition ou chaque élément de la matrice est décalée cycliquement à gauche d'un certain nombre de colonnes.
3. ***MixColumns*** effectue un produit matriciel en opérant sur chaque colonne (vu alors comme un vecteur) de la matrice.

4. ***AddRoundKey*** qui combine par addition chaque octet avec l'octet correspondant dans une clé de ronde obtenue par diversification de la clé de chiffrement.

Enfin, une ronde finale *FinalRound* est appliquée (elle correspond à une ronde dans laquelle l'étape M*ixColumns* est omise).

La clé de ronde pour l'étape i sera notée *RoundKeys[i]*, et *RoundKeys[0]* désignera un des paramètres de l'addition initiale de clé. La dérivation de la clé de chiffrement **K** dans le tableau *RoundKeys[]* est notée *KeyExpansion*.

Les sections suivantes détaillent chaque étape.

Etape d'initialisation

On part du premier bloc :

du message : $\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix}$ et de la clé : $\begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix}$

que l'on additionne par un « ou exclusif » :

$$\begin{bmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \oplus \begin{bmatrix} k_{0,0} & k_{0,1} & k_{0,2} & k_{0,3} \\ k_{1,0} & k_{1,1} & k_{1,2} & k_{1,3} \\ k_{2,0} & k_{2,1} & k_{2,2} & k_{2,3} \\ k_{3,0} & k_{3,1} & k_{3,2} & k_{3,3} \end{bmatrix} = \begin{bmatrix} b_{0,0} & b_{0,1} & b_{0,2} & b_{0,3} \\ b_{1,0} & b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,0} & b_{2,1} & b_{2,2} & b_{2,3} \\ b_{3,0} & b_{3,1} & b_{3,2} & b_{3,3} \end{bmatrix}$$

Le résultat en est la matrice « *State* » $[b_{i,j} = a_{i,j} \oplus k_{i,j}]_{0 \le i,j \le 3}$

Etape S*ubBytes*

L'étape SubBytes correspond à une transformation non-linéaire de l'algorithme. Dans cette étape, chaque élément de la matrice « *State* » est permuté selon une table de substitution inversible notée « ***S-Box*** ». Cette table ainsi que son inverse sont présentées en annexe.

La table ***S-Box*** dérive de la fonction : $g : a \in GF(2^8) \rightarrow a^{-1} \in GF(2^8)$, 0 étant son propre inverse. Cette fonction est connue pour ses bonnes propriétés de non-linéarité. Afin d'éviter des attaques basées sur de simples propriétés algébriques, la « ***S-Box*** » est construite en combinant cette fonction inverse avec une transformation affine inversible ***f***. On a donc :

$$S\text{-}Box[a] = f(g(a)), \quad \forall a \in GF\ [2^8]$$

Les concepteurs ont également fait en sorte que cette application ***SBox*** n'admette pas de point fixe, ni de point fixe opposé:

$$\mathrm{SBox}[a] + a \neq 00, \quad \forall a \in \mathbb{F}_{256}$$

$$\mathtt{SBox}[a] + a \neq \mathtt{FF}, \quad \forall a \in \mathbb{F}_{256}$$

Enfin, la fonction affine f : $GF(2^8) \rightarrow GF(2^8)$ est définie par :

$$b = f(a) \Longleftrightarrow \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

Opération inverse

L'opération inverse de *SubBytes* est notée *InvSubBytes* et consiste à effectuer la même manipulation mais à partir de la ***S-Box*** inverse, notée *InvSBox*. Cette table est également fournie en annexe.

Remarque : mathématiquement, comme la fonction g est son propre inverse, on a :

$$InvSBox(a) = SBox^{-1}(a) = g^{-1}(f^{-1}(a)) = g(f^{-1}(a)), \quad \forall a \in GF(2^8)$$

La fonction affine inverse f^{-1} est définie par :

$$b = f^{-1}(a) \Longleftrightarrow \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 1 \end{bmatrix}$$

L'étape *Shiftrows*

L'étape *Shiftrows* opère sur les lignes de la matrice *State* et effectue un décalage cyclique des lignes de l'état selon différents offsets. La ligne 0 n'est pas décalée, la ligne 1 l'est de C_1 octets, la ligne 2 de C_2 octets, et la ligne 3 de C_3 octets. Les valeurs de C_1, C_2 et C_3 dépendent de la longueur du bloc, selon la table suivante:

m	n	o	p	...		→	pas de décalage	→	m	n	o	p	...	
j	k	l	...			→	décalage de 1	→	k	l	...			j
d	e	f	...			→	décalage de 2	→	f	...			d	e
						→	décalage de 3	→						

w	x	y	z	...	

z	...		w	x	y

Shiftrows: décalage des lignes en fonction de N_b dans ***Rijndael*** (d'après [Dae])

Evidemment, cette table n'est fournie qu'a titre indicatif dans la mesure où l'***AES*** fixe la taille de bloc à $N_b=4$.

Opération inverse

L'opération inverse de *Shiftrows* est notée *InvShiftrows* et consiste à effectuer au niveau le la ligne i un décalage cyclique à droite de C_i éléments.

Etape MixColumns

La transformation *MixColumns* opère sur les colonnes c de la matrice *State* en les traitant comme un polynôme $a(x)$ de degré 3 à coefficients dans $GF(2^8)$.

L'étape *MixColumns* consiste alors à effectuer pour chaque colonne une multiplication par un polynôme ***c(x)*** fixé suivi d'une réduction modulo le polynôme x^4+1. Dans *MixColumns*, on réalise donc l'opération :

$$(03x^3 + x^2 + x + 02) \times a(x) \mod (x^4 + 1)$$

Matriciellement, cette opération s'ecrit :

$$b(x) = c(x) \times a(x) \mod (x^4 + 1) \Longleftrightarrow \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Opération inverse

L'opération inverse de *MixColumns* est notée *InvMixColumns* et consiste à effectuer la même opération mais à partir d'une multiplication par le polynôme $d(x) = c^{-1}(x)$ donné par la relation :

$$(03x^3 + x^2 + x + 02) \times d(x) \equiv 01 \mod (x^4 + 1)$$

On obtient ainsi :

$$d(x) = 0Bx^3 + 0Dx^2 + 09x + 0E$$

Matriciellement, cette étape revient à effectuer le calcul suivant :

$$b(x) = d(x) \times a(x) \mod (x^4+1) \Longleftrightarrow \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

AddRoundKey

Lors de l'étape *AddRoundKey*, la matrice *State* est modifiée en l'additionnant (au sens de l'addition termes à termes dans $GF(2^8)$) avec une clé de ronde.

Détails de la diversification de la clef dans l'*AES*

Cette étape, noté *KeyExpansion*, permet de diversifier la clé de chiffrement ***K*** (de 4 ***Nk*** octets) dans une clé étendue ***W*** de 4***Nb*** (***Nr*** +1) octets. On disposera ainsi de ***Nr*** +1 clés de rondes (chacune de 4***Nb*** octets).

Les matrices ***K*** et ***W*** peuvent être vues comme une succession de colonnes, chacune constituée de 4 octets. Dans la suite, on notera c[i] (resp. k[i]) la (***i*** +1)^{ème} colonne de ***W*** (resp. de ***K***).

L'algorithme utilisé pour la diversification de clé diffère légèrement selon que $N_k \leq 6$ ou $N_k > 6$. Dans tous les cas, les N_k premières colonnes de ***K*** sont recopiées sans modifications aux N_k premières colonnes de ***W***. Les colonnes suivantes sont définies récursivement à partir des colonnes précédentes. *KeyExpansion* y utilise notamment les éléments suivants :

- *SubWord* qui est une fonction prenant en entrée un mot de 4 octets et applique la boîte-S ***S-box*** sur chacun des octets.
- La fonction *RotWord* qui prend en entrée un mot de 4 octets $\boldsymbol{a} = [\boldsymbol{a_0}, \boldsymbol{a_1}, \boldsymbol{a_2}, \boldsymbol{a_3}]$ et effectue une permutation circulaire de façon à renvoyer le mot $[\boldsymbol{a_1}, \boldsymbol{a_2}, \boldsymbol{a_3}, \boldsymbol{a_0}]$.
- Le tableau de constantes de rondes *Rcon[i],* indépendant de N_k, qui est défini récursivement par :

$$\mathrm{Rcon}[i] = [x^{i-1}, 00, 00, 00], \forall i \geq 1$$

On pourra utiliser la fonction ***x^i-time*** pour calculer la valeur de *Rcon[i]*.

Lars Knudsen et *John Mathiassen* [Knu] ont montré en 2004 que le « *key schedule* » joue un rôle primordial dans la résistance aux cryptanalyses de type **linéaire** ou **différentielle**. Les générations de sous-clés, lorsqu'elles sont bien conçues, permettent d'arriver rapidement à l'absence de biais statistiques potentiellement exploitables par la cryptanalyse.

La clé étendue

La clé étendue est un vecteur ***W*** composé de mots de 4 octets (***W*[*i*]** = (***a, b, c, d***)), de longueur $N_b (N_r + 1)$:

- Les N_k premiers mots ($W[1]$, ….$W[N_k]$) contiennent la clé.
- Les mots suivants sont calculés en faisant un *XOR* du mot précédent ($W[i-1]$) et du mot situé N_k positions avant ($W[i - N_k]$).
- Pour les mots situés sur une position qui est un multiple de N_k, une transformation est appliquée à $W[i - 1]$ avant le *XOR*.
  Il s'agit d'une permutation cyclique d'un cran vers la gauche $(a, b, c, d) \rightarrow (b, c, d, a)$, suivie d'une application de la ***S-Box*** séparément sur chaque octet du mot et d'un *XOR* avec un certain vecteur dépendant du tour.
- Si la clé est de 192 bits, et que $(i - 4)$ est un multiple de 4, on applique encore la ***S-Box*** séparément sur chaque octet du mot $W[i - 1]$ déjà modifié, avant de faire le *XOR* final.

Au fur et à mesure des tours, on prend les sous-clés nécessaires les unes à la suite des autres (pour le tour i, on prend les mots de $W[i . N_b]$ à $W[(i+1) . N_b]$)

Déchiffrement dans AES

La routine de chiffrement peut être inversée et réordonnée pour produire un algorithme de déchiffrement utilisant les transformations *InvSubBytes*, *InvShiftRows*, *InvMixColumns*, et *AddRoundKey*.

Dans cette version du déchiffrement, la séquence des transformations diffère de celle du chiffrement, ***le traitement de la clé restant inchangé***.

Attaques

L'***AES*** n'a pour l'instant pas été cassé et la recherche exhaustive (« *force brute* ») demeure la seule solution. ***Rijndael*** a été conçu de telle manière à rendre des méthodes classiques comme la cryptanalyse linéaire ou différentielle impraticables.

Attaques sur des versions simplifiées

Des attaques existent sur des versions simplifiées d'***AES***. *Niels Ferguson* [Fer] et son équipe ont proposé en 2000 une attaque sur une version à 7 tours de l'***AES 128***. Une attaque similaire casse un ***AES 192*** ou ***AES 256*** contenant 8 tours. Un ***AES 256*** peut être cassé s'il est réduit à 9 tours avec une contrainte supplémentaire. En effet, cette dernière attaque repose sur le principe des « related-keys » (clés apparentées). Dans une telle attaque, la clé demeure secrète mais l'attaquant peut spécifier des transformations sur la clé et chiffrer des textes à sa guise. Il peut donc légèrement modifier la clé et regarder comment la sortie de l'***AES*** se comporte.

Attaques sur la version complète

Certains groupes ont affirmé avoir cassé l'***AES*** complet mais après vérification par la communauté scientifique, il s'avérait que toutes ces méthodes étaient erronées. Cependant, plusieurs chercheurs ont mis en évidence des possibilités d'attaques algébriques, notamment l'attaque *XL* et une version améliorée, la *XSL*.

L'attaque XSL

*« L'attaque XSL est basée sur la résolution d'un système d'équations quadratiques qui symbolisent la structure du chiffrement. Le système est typiquement très gros, plus de 8000 équations avec 1600 variables pour un **AES** de 128 bits. Plusieurs méthodes pour résoudre de tels systèmes sont connues (en particulier les bases de Gröbner) et une nouvelle technique a été proposée sous le nom de eXtended Sparse Linearisation (XSL). Malgré ces découvertes, l'attaque reste purement théorique car elle demande une trop grande puissance de calcul.*

*Cette méthode a causé une controverse quant à sa réelle efficacité : les auteurs prétendaient que XSL pouvait potentiellement casser **Rijndael** . Si la communauté des cryptologues est restée sceptique face à cette attaque, elle a eu le mérite de relancer les doutes au sujet de la simplicité algébrique de **AES** »*

Don Coppersmith affirma au sujet de cette attaque :

« Je crois que le travail de Courtois-Pierprzyk [Cou] *a un défaut. Les auteurs surestiment le nombre d'équations linéairement indépendantes. En conséquence, ils n'ont pas assez d'équations linéaires à disposition pour résoudre le système et la méthode ne casse pas **Rijndael.** La méthode a un certain mérite et il est intéressant de l'examiner plus en avant mais elle ne casse pas **Rijndael** en l'état actuel ».*

Chapitre 4 –APPLICATION UTILISANT L'AES - 128

Soit à crypter le message suivant :

Texte clair :**Bekhouche** *(blanc de séparation)* **Saïda** *(blanc de fin de message)*

Clé : **Post-graduation** *(blanc)*

4.1-Chiffrement du Message

4.1.1- Etape d'initialisation

Les octets lus en entrée sont copiés colonne après colonne :

Message

B	o	e	ï
e	u		d
k	c	S	a
h	h	a	

Clé

P	-	d	i
o	g	u	o
s	r	a	n
t	a	t	

↓

4.1.1.1-Codage ASCII 8 bits (hexadécimale)

Le codage du message et de la clé en Hexadécimal, selon le code *ASCII 8 bits* donne:

0x42	*0x6F*	*0x65*	*0xEF*
0x65	*0x75*	*0x20*	*0x64*
0x6B	*0x63*	*0x53*	*0x61*
0x68	*0x68*	*0x61*	*0x20*

0x50	*0x5F*	*0x64*	*0x69*
0x6F	*0x67*	*0x75*	*0x6F*
0x73	*0x72*	*0x61*	*0x6E*
0x74	*0x61*	*0x74*	*0x20*

En notation « pseudo C », les étapes du chiffrement se déroulent de la manière suivante (cf [Dae], page 10/45) :

```
Round(state,RoundKey)
{
ByteSub(State) ;
ShiftRow(State) ;
MixColumn(State)
AddRoundKey(State,RoundKey) ;
}
```

La ronde finale (la 10ème pour le AES-128) est définie par:

```
FinalRound(State,RoundKey)
{
ByteSub(State) ;
ShiftRow(State) ;
AddRoundKey(State,RoundKey) ;
}
```

Nous présentons le détail des différentes étapes de la 1ère ronde :

4.1.2-Ronde n°1

4.1.2.1- Etape « AddRoundkey » :

On réalise l'opération **XOR,** bit à bit entre le message et la clé ; on obtient l'état:

↓

0x12	*0x30*	*0x01*	*0x86*
0x0A	*0x12*	*0x55*	*0x0B*
0x18	*0x11*	*0x32*	*0x0F*
0x1C	*0x09*	*0x15*	*0x00*

(E_{111}) :

4.1.2.2- Etape « SubBytes » :

On substitue à chaque élément de l'état (state) précédent, l'élément correspondant de la ***S-Box*** **;** on obtient l'état suivant :

↓

0xc9	*0x04*	*0x7c*	*0x44*
0x67	*0xc9*	*0xfc*	*0x2b*
0xad	*0x82*	*0x23*	*0x76*
0x9c	*0x01*	*0x59*	*0x63*

(E_{112}) :

4.1.2.3-Etape *« Shift Rows » :*

La 1ère Ligne de l'état précédent est maintenue.
La 2ère Ligne de l'état est décalée d'un cran
La 3ère Ligne de l'état est décalée vers la gauche de deux crans.
La 4ère Ligne de l'état est décalée vers la gauche de trois crans ; on obtient l'état suivant:

↓

0xc9	*0x04*	*0x7 c*	*0x44*
0xc9	*0xfc*	*0x2 b*	*0x67*
0x23	*0x76*	*0xa d*	*0x82*
0x63	*0x9c*	*0x0 1*	*0x59*

(E_{113}) :

4.1.2.4- Etape « *MixColums* »

L'étape MixColums consiste à effectuer, pour chaque colonne, une multiplication par un polynôme c(x) fixé, avec: $c(x) = 03\ X^3 + 01\ X^2 + 01\ X + 02$; on obtient l'état :

On part de la matrice imposée par l'AES

$$(C) : \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix}$$ et on fait les opérations suivantes :

1er : On multiplie la matrice (C) par la 1ère colonne de la matrice (E_{113}) ; on obtient :

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \times \begin{pmatrix} c9 \\ c9 \\ 23 \\ 63 \end{pmatrix} = \begin{pmatrix} 89 \\ 46 \\ e3 \\ 6c \end{pmatrix}$$

2ème : On multiplie (C) par la 2ème colonne de (E_{113}) ; on obtient :

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \times \begin{pmatrix} 04 \\ fc \\ 76 \\ 9c \end{pmatrix} = \begin{pmatrix} fd \\ e1 \\ ad \\ a5 \end{pmatrix}$$

3ème : On multiplie (C) par la 3ème colonne de (E_{113}) ; on obtient :

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \times \begin{pmatrix} 7c \\ 2b \\ ad \\ 01 \end{pmatrix} = \begin{pmatrix} 29 \\ c7 \\ 15 \\ 00 \end{pmatrix}$$

4ème : On multiplie (C) par la 4ème colonne de (E_{113}); on obtient :

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \mathbf{x} \begin{pmatrix} 44 \\ 67 \\ 82 \\ 59 \end{pmatrix} = \begin{pmatrix} \boldsymbol{fa} \\ 4\boldsymbol{e} \\ \boldsymbol{d}7 \\ 9\boldsymbol{b} \end{pmatrix}$$

On obtient en définitive l'état suivant :

(E$_{114}$):

0x89	*0xfd*	*0x29*	*0xfa*
0x46	*0xe1*	*0xc7*	*0x4e*
0xe3	*0xab*	*0x15*	*0xd7*
0x6c	*0xa5*	*0x00*	*0x9b*

La routine d'expansion de la clé :« Key Schedule »

***Clé* : Post-graduation** *(blanc)*

0x50	*0x5F*	*0x64*	*0x69*
0x6F	*0x67*	*0x75*	*0x6F*
0x73	*0x72*	*0x61*	*0x6E*
0x74	*0x61*	*0x74*	*0x20*

(K) :

On précise cette routine pour la ronde n° 1

1- Décalage de C1=1

0x69		*0x6F*			*0xa8*
0x6F		*0x6E*	→ ***Box→***	***S-(B) :***	*0x9f*
0x6E		*0x20*			*0xb7*
0x20		*0x69*			*0xf9*

2- On effectue la somme matricielle : 1ère colonne de ***(K)*** avec ***(B)*** et la matrice fixée par le système :

(AES) :

01
00
00
00

0x50		*0xa8*		*0x01*		*0xf9*
0x6 F		*0x9f*	+	*0x00*		*0xf0*
0x73		*0xb7*	+	*0x00*		*0xc4*
0x74		*0xf9*	=	*0x00*		*0x8d*

(F) *:*

On effectue la somme de la matrice 2ème colonne de ***(K)*** et de ***(F)*** , on obtient :

0x5 F	*0xf9*			*0xa6*
0x67	*0xf0*			*0x97*
0x72	*0xc4*	+		*0xb6*
0x61	*0x8d*	=		*0xec*

(G) *:*

On effectue la somme de la matrice 3ème colonne de ***(K)*** et de ***(G)*** , on obtient :

0x64	*0xa6*			*0xc2*
0x75	*0x97*			*0xe2*
0x61	*0xb6*	+		*0xd7*
0x74	*0xec*	=		*0x98*

(H) :

On effectue la somme de la matrice 4ème colonne de ***(K)*** et de ***(H),*** on obtient ***:***

0x69
0x6F
0x6E
0x20

0xc2
0xe2
0xd7
0x98

+ = **(I)** :

0xab
0x8d
0xb9
0xb8

On obtient alors l'état (clé étendue à l'issue du 1er tour) :

0xf9	0xa6	0xc2	0xab
0xf0	0x97	0xe2	0x8d
0xc4	0xb6	0xd7	0xb9
0x8d	0xec	0x98	0xb8

(K_1) :

4-1.2.5- Etape « Add Roundkey » : Maintenant on effectue la somme de la matrice de la matrice **(K_1)** avec la matrice**(E_{114}),** on obtient l'état donné par**(E_{115}):**

0xf9	0xa6	0xc2	0xab
0xf0	0x97	0xe2	0x8d
0xc4	0xb6	0xd7	0xb9
0x8d	0xec	0x98	0xb8

+

0x89	0xfd	0x29	0xfa
0x46	0xe1	0xc7	0x4e
0xe3	0xab	0x15	0xd7
0x6c	0xa5	0x00	0x9b

$= (E_{115})$:

$$\begin{pmatrix} 0x700x5b0xeb0x & \\ 0xb60x760x250x & \\ 0x270x1d0xc20x & \\ 0xe10x490x980x & \end{pmatrix}$$

La matrice (E_{115}) est le résultat obtenu à la fin de la ronde n° 1, et constitue le point de départ de la ronde n°2 qui se déroule selon les mêmes étapes que celles de la ronde n°1, et ainsi de suite, jusqu'à la dernière ronde n°10.

On obtient ainsi la suite des états 1 à 10 :

Début de la ronde	*Après SubBytes*	*Après ShiftRows*	*Après MixColumns*	*RoundKey*

Entrée

42	*6f*	*65*	*ef*
65	*75*	*20*	*64*
6b	*63*	*53*	*61*
68	*68*	*61*	*20*

k			
50	*5f*	*64*	*69*
6f	*67*	*75*	*6f*
73	*72*	*61*	*6e*
74	*61*	*74*	*20*

Ronde 1

12	*30*	*01*	*86*
0a	*12*	*55*	*0b*
18	*11*	*32*	*0f*
1c	*09*	*15*	*00*

c9	*04*	*7c*	*44*
67	*c9*	*fc*	*2b*
ad	*82*	*23*	*76*
9c	*01*	*59*	*63*

c9	*04*	*7c*	*44*
c9	*fc*	*2b*	*67*
23	*76*	*ad*	*82*
63	*9c*	*01*	*59*

89	*fd*	*29*	*fa*
46	*e1*	*c7*	*4e*
e3	*ab*	*15*	*d7*
6c	*a5*	*00*	*9b*

K_1			
f9	*a6*	*c2*	*ab*
f0	*97*	*e2*	*8d*
c4	*b6*	*d7*	*b9*
8d	*ec*	*98*	*b8*

Ronde 2

70	*5b*	*eb*	*51*
b6	*76*	*25*	*c3*
27	*1d*	*c2*	*e6*
e1	*49*	*98*	*23*

51	*39*	*e9*	*d1*
e4	*38*	*3f*	*2e*
cc	*a4*	*25*	*9f*
f8	*3b*	*46*	*26*

51	*39*	*e9*	*d1*
38	*3f*	*2e*	*4e*
25	*9f*	*cc*	*a4*
26	*f8*	*3b*	*46*

e9	*54*	*4c*	*89*
68	*05*	*c1*	*fc*
49	*30*	*09*	*06*
a2	*00*	*b4*	*0e*

K_2			
a6	*00*	*c2*	*69*
a6	*31*	*d3*	*5e*
a8	*1e*	*c9*	*70*
ef	*03*	*9b*	*23*

Ronde 3

4f	*54*	*8e*	*e0*
ce	*34*	*12*	*a2*
e1	*2e*	*c0*	*76*
4d	*03*	*2f*	*2d*

84	*20*	*19*	*e1*
8b	*18*	*c9*	*3a*
f8	*31*	*ba*	*38*
e3	*7b*	*15*	*d8*

84	*20*	*19*	*e1*
18	*c9*	*3a*	*8b*
ba	*38*	*f8*	*31*
d8	*e3*	*7b*	*15*

59	*db*	*ff*	*7b*
b9	*02*	*05*	*aa*
80	*a7*	*45*	*37*
9e	*4c*	*1f*	*a8*

K_3			
fa	*fa*	*38*	*51*
f7	*c6*	*15*	*4b*
8e	*90*	*59*	*29*
16	*15*	*8e*	*ad*

Ronde 4

a3	*21*	*c7*	*2a*
4e	*c4*	*10*	*e1*
0e	*37*	*1c*	*1e*
88	*59*	*91*	*05*

0a	*fd*	*c6*	*e5*
2f	*1c*	*ca*	*f8*
ab	*9a*	*9c*	*72*
c4	*cb*	*81*	*6b*

0a	*fd*	*c6*	*e5*
1c	*ca*	*f8*	*2f*
9c	*72*	*ab*	*9a*
6b	*c4*	*cb*	*81*

c7	*12*	*e4*	*bb*
e6	*20*	*00*	*8f*
88	*84*	*35*	*7d*
48	*37*	*8f*	*98*

K_4			
41	*bb*	*83*	*d2*
52	*94*	*81*	*ca*
1b	*8b*	*d2*	*fb*
c7	*d2*	*5c*	*f1*

86	a9	67	69
b4	b4	81	45
93	0f	e7	86
8f	e5	d3	69

44	d3	85	f9
8d	8d	0c	6e
dc	76	94	44
73	d9	66	f9

44	d3	85	f9
8d	0c	6e	8d
94	44	dc	76
f9	73	d9	66

69	9e	a6	75
1b	74	ff	04
ea	c2	38	32
3c	c0	8f	27

K_5			
25	9e	1d	cf
5d	c9	48	82
ba	31	e3	18
72	a0	fc	0d

Ronde 5

Début de la ronde ***Après SubBytes*** ***Après ShiftRows*** ***Après MixColumns*** ***RoundKey***

↓ ↓ ↓ ↓ ↓

4c	00	bb	ba
46	bd	b7	86
50	f3	db	2a
4e	60	73	2a

29	63	ea	f4
5a	7a	a9	44
53	0d	b9	e5
2f	d0	8f	e5

29	63	ea	f4
7a	a9	44	5a
b9	e5	53	0d
e5	2f	d0	8f

80	ec	80	9f
e8	31	47	d8
0e	6a	63	3e
69	b7	89	55

K_6			
16	88	95	5a
f0	39	71	f3
6d	5c	bf	a7
f8	58	a4	a9

Ronde 6

96	64	15	c5
18	08	36	2b
63	36	dc	99
91	ef	2d	fc

90	43	59	a6
ad	30	05	f1
fb	05	86	Ee
81	df	d8	b0

90	43	59	a6
30	05	f1	ad
86	ee	fb	05
b0	81	df	d8

5d	e6	9e	66
d1	e1	69	30
7c	19	3f	72
66	37	44	f2

K_7			
5b	d3	46	1c
ac	95	e4	17
be	e2	5d	fa
46	1e	ba	13

Ronde 7

06	35	d8	7a
7d	74	8d	27
c2	fb	62	88
20	29	F e	e1

6f	96	61	Da
ff	92	5d	Cc
25	0f	aa	c4
b7	a5	bb	f8

6f	96	61	da
92	5d	cc	ff
aa	c4	25	0f
f8	b7	a5	bb

21	a3	0d	01
4d	cc	28	95
a1	9a	13	ed
62	4d	1b	e8

K_8			
2b	f8	be	a2
81	14	f0	e7
c3	21	7c	86
da	c4	7e	6d

Ronde 8

0a	5b	b3	a3
cc	d8	d8	72
62	bb	6f	6b
b8	89	65	85

67	39	6d	0a
4b	61	61	40
aa	ea	a8	7f
6c	a7	4d	97

67	39	6d	0a
61	61	40	4b
a8	7f	aa	ea
97	6c	a7	4d

52	c2	17	6e
d1	16	af	f4
ef	12	90	59
55	8d	08	25

K_9			
a4	5c	e2	40
c5	d1	21	c6
ff	de	a2	24
e0	24	5a	37

Ronde 9

f6	*9e*	*f5*	*e2*
14	*c7*	*8e*	*32*
10	*cc*	*32*	*7d*
b5	*a9*	*52*	*12*

42	*0b*	*e6*	*31*
fa	*c6*	*19*	*23*
ca	*4d*	*23*	*Ff*
d5	*d3*	*00*	*c9*

42	*0b*	*e6*	*31*
c6	*19*	*23*	*fa*
23	*ff*	*ca*	*4d*
c9	*d5*	*d3*	*00*

K_{10}			
26	*7a*	*98*	*d8*
f3	*22*	*03*	*c5*
65	*bb*	*19*	*3d*
e9	*cd*	*97*	*a0*

Ronde10

On obtient ***en output*** l'état final, c'est-à-dire le chiffré (***ciphertext***) :

0x64	*0x71*	*0x7e*	*0xe9*
0x35	*0x3b*	*0x20*	*0x3f*
0x46	*0x44*	*0xd3*	*0x76*
0x20	*0x18*	*0x44*	*0xa0*

⇔ *(après décodage ASCII)*

d	*q*	~	E
5	*;*		?
F	*D*	*Ó*	V
	CAN	*D*	

qui correspond au chiffré (*ciphertext*) :

d5F(*blanc*)**q ;DCAN** ~(*blanc*) **ÓDé ?**

v(*blanc*) alors que le message était : **Bekhouche** (*blanc*)**Saïda**(*blanc*)

4-2- Déchiffrement

Le déchiffrement est obtenu par inversion des processus de chiffrement, que l'on peut décrire de façon formelle selon les étapes suivantes (en « Pseudo *C* ») :

```
I_Round (State, I_RoundKey)
{
InvByteSub(State) ;
InvShiftRow (State) ;
InvMixColumn(State) ;
AddRoundKey(State, I_RoundKey) ;
}

I_FinalRound(State, I_RoundKey) ;
{
InvByteSub(State) ;
InvShiftRow(State) ;
AddRoundKey(State, RoundKey 0) ;
}
```

Ces notations sont précisées comme suit :

- InvByteSub est la substitution qui utilise la S-Box inverse
- InvShiftRow est la transformation inverse de la transformation **ShiftRow**
- InvMixColumn est similaire à la transformation MixColumn sauf qu'il faut utiliser le polynôme ***d(x)*** défini par : ***d(x)=0B x^3+ 0D x^2 + 09 x + 0E.***

Dans notre cas, la situation de départ se présente comme suit:

Chiffré (**Ciphertext**): **d5F**(*blanc*)**q ;DCAN ~**(*blanc*) **ÓDé ?v**(*blanc*)
Clé (**K**): **Post-graduation** *(blanc)*

4.2.1-Etape d'initialisation

Chiffré

d	Q	~	é
5	;		?
F	D	**Ó**	v
	CAN	D	

Clé (K)

P	-	d	i
o	g	u	o
s	r	a	n
t	a	t	

↓

4.2.1 .1-Codage ASCII 8 bits (hexadécimale)

Le codage du chiffré et de la clé en Hexadécimal, selon le code ASCII donne:

0x64	*0x71*	*0x7e*	*0xe9*
0x35	*0x3b*	*0x20*	*0x3f*
0x46	*0x44*	*0xd3*	*0x76*
0x20	*0x18*	*0x44*	*0xa0*

4.2.2.1 - *Ronde n°1*

0x50	0x5F	0x64	0x69
0x6F	0x67	0x75	0x6F
0x73	0x72	0x61	0x6E
0x74	0x61	0x74	0x20

4.2.2.1.1- Etape « AddRoundkey » :

On réalise l'opération **XOR,** bit à bit entre le chiffré et la clé de ronde n° 10 : ($\mathbf{K_{10}}$);

0x64	*0x71*	*0x7e*	*0xe9*
0x35	*0x3b*	*0x20*	*0x3f*
0x46	*0x44*	*0xd3*	*0x76*
0x20	*0x18*	*0x44*	*0xa0*

+

0x26	0x7a	0x98	0xd8
0xf3	0x22	0x03	0xc5
0x65	0xbb	0x19	0x3d
0xe9	0xcd	0x97	0xa0

On obtient l'état:

↓

(E_{211}) :

0x42	*0x0b*	*0xe6*	*0x31*
0xc6	*0x19*	*0x23*	*0xfa*
0x23	*0xff*	*0xca*	*0x4b*
0xc9	*0xd5*	*0xd3*	*0x00*

4.2.2.1.2- Etape « InvShiftRow »

La 1ère Ligne de l'état précédent est maintenue.
La 2ère Ligne de l'état est décalée vers la droite d'un cran
La 3ère Ligne de l'état est décalée vers la droite de deux crans.
La 4ère Ligne de l'état est décalée vers la droite de trois crans, on obtient l'état suivant:

(E_{212}) :

0x42	*0x0b*	*0xe6*	*0x31*
0xfa	*0xc6*	*0x19*	*0x23*
0xca	*0x4b*	*0x23*	*0xff*
0xd5	*0xd3*	*0x00*	*0xc9*

4.2.2.1.3. Etape « InvSubBytes » :

On substitue à chaque élément de l'état (state) précédent, l'élément correspondant de la ***S-Box inverse*** ; on obtient l'état suivant :

(E_{213}) :

0xf6	*0x9e*	*0xf5*	*0x2e*
0x14	*0xc7*	*0x8e*	*0x32*
0x10	*0xcc*	*0x32*	*0x7d*
0xb5	*0xa9*	*0x52*	*0x12*

4.2.2.1.4. Etape « AddRoundKey,I-ExpandedKey+Nb) »

On effectue un ***XOR*** entre l'état ***(E_{213})*** et la clé étendue du tour n° 9 qui est :

(K_9) :

0xa4	*0x5c*	*0xe2*	*0x40*
0xc5	*0xd1*	*0x21*	*0xc6*
0xff	*0xde*	*0xa2*	*0x24*
0x52	*0xc2*	*0x1*	*0x6e*

On obtient l'état suivant :

(E_{214}) :

0xe0	*0x24*	*0x57*	*0x97*
0xd1	*0x16*	*0xaf*	*0xf4*
0xef	*0x12*	*0x9c*	*0x59*
0x55	*0x8d*	*0x08*	*0x25*

On effectue maintenant l'opération **« InvMixColumn »** en multipliant chaque colonne de la matrice précédente par le polynôme : ***$d(x) = 0Bx^3 + 0Dx^2 + 09x + 0E$*** *(imposé par l'AES)*

On part de la matrice imposée par l'AES

(D) **:** $\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix}$ et on fait les opérations suivantes :

1er : On multiplie la matrice ***(D)*** par la 1ère colonne de la matrice ***(E_{214})*** ; on obtient :

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \text{x} \begin{pmatrix} 52 \\ \boldsymbol{d}1 \\ \boldsymbol{ef} \\ 55 \end{pmatrix} = \begin{pmatrix} 67 \\ 61 \\ \boldsymbol{a}8 \\ 97 \end{pmatrix};$$

2ème : On multiplie ***(D)*** par la 2ème colonne de ***(E_{214})*** ; on obtient :

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \text{x} \begin{pmatrix} \boldsymbol{c}2 \\ 16 \\ 12 \\ 8\boldsymbol{d} \end{pmatrix} = \begin{pmatrix} 39 \\ 61 \\ 7\boldsymbol{f} \\ 6\boldsymbol{c} \end{pmatrix}$$

3ème : On multiplie ***(D)*** par la 3ème colonne de ***(E_{214})*** ; on obtient :

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \text{x} \begin{pmatrix} 17 \\ \boldsymbol{af} \\ 90 \\ 08 \end{pmatrix} = \begin{pmatrix} 6\boldsymbol{d} \\ 40 \\ \boldsymbol{aa} \\ \boldsymbol{a}7 \end{pmatrix}$$

4ème : On multiplie ***(D)*** par la 4ème colonne de ***(E_{214})***; on obtient :

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \text{x} \begin{pmatrix} 6\boldsymbol{e} \\ \boldsymbol{f}4 \\ 59 \\ 25 \end{pmatrix} = \begin{pmatrix} 0\boldsymbol{a} \\ 4\boldsymbol{b} \\ \boldsymbol{ea} \\ 4\boldsymbol{d} \end{pmatrix}$$

On obtient en définitive l'état suivant :

(E_{215}):

0x67	*0x39*	*0x6d*	*0x0a*
0x61	*0x61*	*0x40*	*0x4b*
0xa8	*0x7f*	*0xaa*	*0xea*
0x97	*0x6c*	*0xa7*	*0x4d*

Ce qui clôture la ronde n° 1

4.2.2.2-*Ronde n°2*

On part de l'état (E_{215}) :

0x67	0x39	0x6d	0x0a
0x61	0x61	0x40	0x4b
0xa8	0x7f	0xaa	0xea
0x97	0x6c	0xa7	0x4d

4.2.2.2.1 - Etape « InvShiftRow »

La 1ère Ligne de l'état précédent est maintenue.
La 2ère Ligne de l'état est décalée vers la droite d'un cran
La 3ère Ligne de l'état est décalée vers la droite de deux crans.
La 4ère Ligne de l'état est décalée vers la droite de trois crans, on obtient l'état suivant:

(E_{221}) :

0x67	0x39	0x6d	0x0a
0x4b	0x61	0x161	0x40
0xaa	0xea	0xa8	0x7f
0x6c	0xa7	0x4d	0x97

4. 2.2.2.2. Etape « InvSubBytes » :

On substitue à chaque élément de l'état (state) précédent (E_{221}) l'élément correspondant de la ***S-Box inverse*** ; on obtient l'état suivant :

(E_{222}) :

0x0a	0x5b	0xb3	0xa3
0xcc	0xd8	0xd8	0x72
0x62	0xbb	0x6f	0x6b
0xb8	0x89	0x65	0x85

4.2.2.2.3. Etape « AddRoundKey,I-ExpandedKey+Nb) »

On effectue un ***XOR*** entre l'état (E_{222}) et la clé étendue du tour n° 8 qui est :

(K_8) :

0x2b	*0xf8*	*0xbe*	*0xa2*
0x81	*0x14*	*0xf0*	*0xe7*
0xc3	*0x21*	*0x7c*	*0x86*
0xda	*0xc4*	*0x7e*	*0x6d*

On obtient l'état suivant :

(E_{223}) :

0x21	*0xa3*	*0x0d*	*0x01*
0x4d	*0xcc*	*0x28*	*0x95*
0xa1	*0x9a*	*0x13*	*0xed*
0x62	*0x4d*	*0x1b*	*0xe8*

On effectue maintenant l'opération « **InvMixColumn** » en multipliant chaque colonne de la matrice précédente par le polynôme : $d(x) = 0Bx^3 + 0Dx^2 + 09x + 0E$ *(imposé par l'AES)*

On part de la matrice imposée par l'*AES*

(D) : $\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix}$ et on fait les opérations suivantes :

1er : On multiplie la matrice (D) par la 1ère colonne de la matrice (E_{223}) ; on obtient :

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \times \begin{pmatrix} 21 \\ 4d \\ a1 \\ 62 \end{pmatrix} = \begin{pmatrix} 6f \\ 92 \\ aa \\ f8 \end{pmatrix}$$

2ème : On multiplie (D) par la 2ème colonne de (E_{223}) ; on obtient :

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \times \begin{pmatrix} a3 \\ cc \\ 9a \\ 4d \end{pmatrix} = \begin{pmatrix} 96 \\ 5d \\ c4 \\ b7 \end{pmatrix}$$

3ème : On multiplie ***(D)*** par la 3ème colonne de ***(E_{223})*** ; on obtient :

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \mathbf{x} \begin{pmatrix} 0d \\ 28 \\ 13 \\ 1b \end{pmatrix} = \begin{pmatrix} 61 \\ \boldsymbol{cc} \\ 25 \\ \boldsymbol{a}5 \end{pmatrix}$$

4ème : On multiplie ***(D)*** par la 4ème colonne de ***(E_{223})*** ; on obtient :

$$\begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \mathbf{x} \begin{pmatrix} 01 \\ 95 \\ ed \\ e8 \end{pmatrix} = \begin{pmatrix} \boldsymbol{da} \\ \boldsymbol{ff} \\ 0\boldsymbol{f} \\ \boldsymbol{bb} \end{pmatrix}$$

On obtient en définitive l'état suivant :

$$\boldsymbol{(E_{224}):} \begin{pmatrix} 0 & x & 60 & fx & 90 & 6x & 60 & 1x \\ 0 & x & 90 & 2x & 50 & dx & c0 & cx \\ 0 & x & a0 & ax & c0 & 4x & 20 & 5x \\ 0 & x & f08 & x & b0 & 7x & a0 & 5x \end{pmatrix}$$

Ce qui clôture la ronde n° 2

La matrice ***(E_{224})*** est le résultat obtenu à la fin de la ronde n° 2, et constitue le point de départ de la ronde n°3 qui se déroule selon les mêmes étapes que celles de la ronde n°2, et ainsi de suite, jusqu'à la dernière ronde n°10.

On obtient la suite des états 1 à 10 :

Début de la ronde	*Après InvShiftRows*	*Après InvSubBytes*	*Après RoundKey*	*Après AddRoundKey*
↓	↓	↓	↓	↓

64	*71*	*7e*	*e9*
35	*3b*	*20*	*3f*
46	*44*	*d3*	*76*
20	*18*	*44*	*a0*

K_{10}			
26	*7a*	*98*	*d8*
f3	*22*	*03*	*c5*
65	*bb*	*19*	*3d*
e9	*cd*	*97*	*a0*

Entrée

42	*0b*	*e6*	*31*	*42*	*0b*	*e6*	*31*	*f6*	*9e*	*f5*	*2e*		K_9			*52*	*c2*	*17*	*6e*
c6	*19*	*23*	*fa*	*fa*	*c6*	*19*	*23*	*14*	*c7*	*8e*	*32*	*a4*	*5c*	*e2*	*40*	*d1*	*16*	*af*	*f4*
23	*ff*	*ca*	*4b*	*ca*	*4b*	*23*	*ff*	*10*	*cc*	*32*	*7d*	*c5*	*d1*	*21*	*c6*	*ef*	*12*	*90*	*59*
c9	*d5*	*d3*	*00*	*d5*	*d3*	*00*	*c9*	*b5*	*a9*	*52*	*12*	*ff*	*de*	*a2*	*24*	*55*	*8d*	*08*	*25*
												e0	*24*	*5a*	*37*				

invRonde1

67	*39*	*6d*	*0a*	*67*	*39*	*6d*	*0a*	*0a*	*5b*	*b3*	*a3*		K_8			*21*	*a3*	*0d*	*01*
61	*61*	*40*	*4b*	*4b*	*61*	*61*	*40*	*cc*	*d8*	*d8*	*72*	*2b*	*f8*	*be*	*a2*	*4d*	*cc*	*28*	*95*
a8	*7f*	*aa*	*ea*	*aa*	*ea*	*a8*	*7f*	*62*	*bb*	*f6*	*b6*	*81*	*14*	*f0*	*e7*	*a1*	*9a*	*13*	*ed*
97	*6c*	*a7*	*4d*	*6c*	*a7*	*4d*	*97*	*b8*	*89*	*65*	*85*	*c3*	*21*	*7c*	*86*	*62*	*4d*	*1b*	*e8*
												da	*c4*	*7e*	*6d*				

invRonde2

6f	*96*	*61*	*da*	*6f*	*96*	*61*	*Da*	*06*	*35*	*d8*	*7a*		K_7			*5d*	*e6*	*9e*	*66*
92	*5d*	*cc*	*ff*	*ff*	*92*	*5d*	*Cc*	*7d*	*74*	*8d*	*27*	*5b*	*d3*	*46*	*1c*	*d1*	*e1*	*69*	*30*
aa	*c4*	*25*	*0f*	*25*	*0f*	*aa*	*c4*	*c2*	*fb*	*62*	*88*	*ac*	*95*	*e4*	*17*	*7c*	*19*	*3f*	*72*
f8	*b7*	*a5*	*bb*	*b7*	*a5*	*bb*	*f8*	*20*	*29*	*fe*	*e1*	*be*	*e2*	*5d*	*fa*	*66*	*37*	*44*	*f2*
												46	*1e*	*ba*	*13*				

invRonde3

90	*43*	*59*	*a6*	*90*	*43*	*59*	*a6*	*96*	*64*	*15*	*c5*		K_6			*80*	*ec*	*80*	*9f*
30	*05*	*f1*	*ad*	*ad*	*30*	*05*	*f1*	*18*	*08*	*36*	*2b*	*16*	*88*	*95*	*5a*	*e8*	*31*	*47*	*d8*
86	*ee*	*fb*	*05*	*fb*	*05*	*86*	*ee*	*63*	*36*	*dc*	*99*	*f0*	*39*	*71*	*f3*	*0e*	*6a*	*63*	*3e*
b0	*81*	*df*	*d8*	*81*	*df*	*d8*	*b0*	*91*	*ef*	*2d*	*fc*	*6d*	*5c*	*bf*	*a7*	*69*	*b7*	*89*	*55*
												f8	*58*	*a4*	*a9*				

invRonde4

29	*63*	*ea*	*f4*	*29*	*63*	*ea*	*f4*	*4c*	*00*	*bb*	*ba*		K_5			*69*	*9e*	*a6*	*75*
7a	*a9*	*44*	*5a*	*5a*	*7a*	*a9*	*44*	*46*	*bd*	*b7*	*86*	*25*	*e9*	*1d*	*cf*	*1b*	*74*	*ff*	*04*
b9	*e5*	*53*	*0d*	*53*	*0d*	*b9*	*e5*	*50*	*f3*	*db*	*2a*	*5d*	*c9*	*48*	*82*	*ea*	*c2*	*38*	*32*
e5	*2f*	*d0*	*f8*	*2f*	*d0*	*f8*	*e5*	*4e*	*60*	*73*	*2a*	*ba*	*31*	*e3*	*18*	*3c*	*c0*	*8f*	*27*
												72	*a0*	*fc*	*0d*				

invRonde5

Début de la ronde	***Après InvShiftRows***	***Après InvSubBytes***	***Après RoundKey***	***Après AddRoundKey***
↓	↓	↓	↓	↓

invRonde6

44	*d3*	*85*	*f9*
8d	*0c*	*6e*	*8d*
94	*44*	*dc*	*76*
f9	*73*	*d9*	*66*

44	*d3*	*85*	*f9*
8d	*8d*	*0c*	*6e*
dc	*76*	*94*	*44*
73	*d9*	*66*	*f9*

86	*a9*	*67*	*69*
b4	*b4*	*81*	*45*
93	*0f*	*e7*	*86*
8f	*e5*	*d3*	*69*

K_4			
41	*bb*	*83*	*d2*
52	*94*	*81*	*Ca*
1b	*8b*	*d2*	*Fb*
c7	*d2*	*5c*	*f1*

c7	*12*	*e4*	*bb*
e6	*20*	*00*	*8f*
88	*84*	*35*	*7d*
48	*37*	*8f*	*98*

invRonde7

0a	*fd*	*c6*	*e5*
1c	*ca*	*f8*	*2f*
9c	*72*	*ab*	*9a*
6b	*c4*	*cb*	*81*

0a	*fd*	*c6*	*e5*
2f	*1c*	*ca*	*f8*
ab	*9a*	*9c*	*72*
c4	*cb*	*81*	*6b*

a3	*21*	*c7*	*2a*
4e	*c4*	*10*	*e1*
0e	*37*	*1c*	*1e*
88	*59*	*91*	*05*

K_3			
fa	*fa*	*38*	*51*
f7	*c6*	*15*	*4b*
8e	*90*	*59*	*29*
16	*15*	*8e*	*Ad*

59	*db*	*ff*	*7b*
b9	*02*	*05*	*aa*
80	*a7*	*45*	*37*
9e	*4c*	*1f*	*a8*

invRonde8

84	*20*	*19*	*e1*
18	*c9*	*3a*	*8b*
ba	*38*	*f8*	*31*
d8	*e3*	*7b*	*15*

84	*20*	*19*	*e1*
8b	*18*	*c9*	*3a*
f8	*31*	*ba*	*38*
e3	*7b*	*15*	*d8*

4f	*54*	*8e*	*e0*
ce	*34*	*12*	*a2*
e1	*2e*	*c0*	*76*
4d	*03*	*2f*	*2d*

K_2			
a6	*00*	*c2*	*69*
a6	*31*	*d3*	*5e*
a8	*1e*	*c9*	*70*
ef	*03*	*9b*	*23*

e9	*54*	*4c*	*89*
68	*05*	*c1*	*fc*
49	*30*	*09*	*06*
a2	*00*	*b4*	*0e*

invRonde9

51	*39*	*e9*	*d1*
38	*3f*	*2e*	*4e*
25	*9f*	*cc*	*a4*
26	*f8*	*3b*	*46*

51	*39*	*e9*	*d1*
4e	*38*	*3f*	*2e*
cc	*a4*	*25*	*9f*
f8	*3b*	*46*	*26*

70	*5b*	*eb*	*51*
b6	*76*	*25*	*c3*
27	*1d*	*c2*	*6e*
e1	*49*	*98*	*23*

K_1			
f9	*a6*	*c2*	*Ab*
f0	*97*	*e2*	*8d*
c4	*b6*	*d7*	*b9*
8d	*ec*	*98*	*b8*

89	*fd*	*29*	*fa*
46	*e1*	*c7*	*4e*
e3	*ab*	*15*	*d7*
6c	*a5*	*00*	*9b*

invRonde10

c9	*04*	*7c*	*44*
c9	*fc*	*2b*	*67*
23	*76*	*ad*	*82*
63	*9c*	*01*	*59*

c9	*04*	*7c*	*44*
67	*c9*	*fc*	*2b*
ad	*82*	*23*	*76*
9c	*01*	*59*	*63*

12	*30*	*01*	*86*
0a	*12*	*55*	*0b*
18	*11*	*32*	*0f*
1c	*09*	*15*	*00*

K			
50	*5f*	*64*	*69*
6f	*67*	*75*	*6f*
73	*72*	*61*	*6e*
74	*61*	*74*	*20*

Qui est l'état final (Plaintext) :

42	6f	65	ef
65	75	20	64
6b	63	53	61
68	68	61	20

Après décodage ASCII 8 bits, on obtient :

B	o	e	ï
e	u		d
k	c	S	a
h	h	a	

On obtient en définitive notre message reconstitué :

Bekhouche (*blanc***) Saïda (***blanc***)**

Conclusion

Nous avons explicité les mathématiques afférentes à l'AES, et nous avons également présenté une description complète de ce cryptosystème.

Nous avons détaillé toutes les étapes de fonctionnement de l'AES, aussi bien en chiffrement qu'en déchiffrement, à travers une application.

Nous pouvons dire que nous maîtrisons parfaitement ce standard de chiffrement avancé, utilisé par les plus grandes nations : il reste seulement à écrire le logiciel correspondand, ce qui est une simple affaire d'Informatique.

La sécurité du cryptosystème peut être amélioré en changeant la S-Box et en calculant son inverse par un procédé mathématique d'inversion de matrice, et ce en procédant au réarrangement de la table *ASCCI 8 bits* ;

ANNEXES

Détail de la S-Box et son inverse

```
/* La S-Box */
const F256 SBox[256] = {
    0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE,
    0xD7, 0xAB, 0x76, 0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4,
    0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0, 0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7,
    0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15, 0x04, 0xC7, 0x23, 0xC3,
    0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75, 0x09,
    0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3,
    0x2F, 0x84, 0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE,
    0x39, 0x4A, 0x4C, 0x58, 0xCF, 0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85,
    0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8, 0x51, 0xA3, 0x40, 0x8F, 0x92,
    0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2, 0xCD, 0x0C,
    0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19,
    0x73, 0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14,
    0xDE, 0x5E, 0x0B, 0xDB, 0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2,
    0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79, 0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5,
    0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08, 0xBA, 0x78, 0x25,
    0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A,
    0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86,
    0xC1, 0x1D, 0x9E, 0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E,
    0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF, 0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42,
    0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16
};
```

```
/* la S-Box inverse */
const F256 InvSBox[256] = {
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA3, 0x9E, 0x81,
    0xF3, 0xD7, 0xFB, 0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E,
    0x43, 0x44, 0xC4, 0xDE, 0xE9, 0xCB, 0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23,
    0x3D, 0xEE, 0x4C, 0x95, 0x0B, 0x42, 0xFA, 0xC3, 0x4E, 0x08, 0x2E, 0xA1, 0x66,
    0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA2, 0x49, 0x6D, 0x8B, 0xD1, 0x25, 0x72,
    0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5C, 0xCC, 0x5D, 0x65,
    0xB6, 0x92, 0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x46,
    0x57, 0xA7, 0x8D, 0x9D, 0x84, 0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A,
    0xF7, 0xE4, 0x58, 0x05, 0xB8, 0xB3, 0x45, 0x06, 0xD0, 0x2C, 0x1E, 0x8F, 0xCA,
    0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xBD, 0x03, 0x01, 0x13, 0x8A, 0x6B, 0x3A, 0x91,
    0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xCF, 0xCE, 0xF0, 0xB4, 0xE6,
    0x73, 0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x37, 0xE8,
    0x1C, 0x75, 0xDF, 0x6E, 0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F,
    0xB7, 0x62, 0x0E, 0xAA, 0x18, 0xBE, 0x1B, 0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2,
    0x79, 0x20, 0x9A, 0xDB, 0xC0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4, 0x1F, 0xDD, 0xA8,
    0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x10, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7A, 0x9F, 0x93,
    0xC9, 0x9C, 0xEF, 0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB,
    0xBB, 0x3C, 0x83, 0x53, 0x99, 0x61, 0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6,
    0x26, 0xE1, 0x69, 0x14, 0x63, 0x55, 0x21, 0x0C, 0x7D
};
```

Tables de correspondances entre écritures polynomiales et exponentielles

```
// ExpoToPoly[k] donne la representation polynomiale de w(x)^k
// 0 est representer par w(x)^255 bien que mathematiquement, w^255=1
const int ExpoToPoly[256] = {
    0x01, 0x03, 0x05, 0x0f, 0x11, 0x33, 0x55, 0xff, 0x1a, 0x2e, 0x72, 0x96, 0xa1,
    0xf8, 0x13, 0x35, 0x5f, 0xe1, 0x38, 0x48, 0xd8, 0x73, 0x95, 0xa4, 0xf7, 0x02,
    0x06, 0x0a, 0x1e, 0x22, 0x66, 0xaa, 0xe5, 0x34, 0x5c, 0xe4, 0x37, 0x59, 0xeb,
    0x26, 0x6a, 0xbe, 0xd9, 0x70, 0x90, 0xab, 0xe6, 0x31, 0x53, 0xf5, 0x04, 0x0c,
    0x14, 0x3c, 0x44, 0xcc, 0x4f, 0xd1, 0x68, 0xb8, 0xd3, 0x6e, 0xb2, 0xcd, 0x4c,
    0xd4, 0x67, 0xa9, 0xe0, 0x3b, 0x4d, 0xd7, 0x62, 0xa6, 0xf1, 0x08, 0x18, 0x28,
    0x78, 0x88, 0x83, 0x9e, 0xb9, 0xd0, 0x6b, 0xbd, 0xdc, 0x7f, 0x81, 0x98, 0xb3,
    0xce, 0x49, 0xdb, 0x76, 0x9a, 0xb5, 0xc4, 0x57, 0xf9, 0x10, 0x30, 0x50, 0xf0,
    0x0b, 0x1d, 0x27, 0x69, 0xbb, 0xd6, 0x61, 0xa3, 0xfe, 0x19, 0x2b, 0x7d, 0x87,
    0x92, 0xad, 0xec, 0x2f, 0x71, 0x93, 0xae, 0xe9, 0x20, 0x60, 0xa0, 0xfb, 0x16,
    0x3a, 0x4e, 0xd2, 0x6d, 0xb7, 0xc2, 0x5d, 0xe7, 0x32, 0x56, 0xfa, 0x15, 0x3f,
    0x41, 0xc3, 0x5e, 0xe2, 0x3d, 0x47, 0xc9, 0x40, 0xc0, 0x5b, 0xed, 0x2c, 0x74,
    0x9c, 0xbf, 0xda, 0x75, 0x9f, 0xba, 0xd5, 0x64, 0xac, 0xef, 0x2a, 0x7e, 0x82,
    0x9d, 0xbc, 0xdf, 0x7a, 0x8e, 0x89, 0x80, 0x9b, 0xb6, 0xc1, 0x58, 0xe8, 0x23,
    0x65, 0xaf, 0xea, 0x25, 0x6f, 0xb1, 0xc8, 0x43, 0xc5, 0x54, 0xfc, 0x1f, 0x21,
    0x63, 0xa5, 0xf4, 0x07, 0x09, 0x1b, 0x2d, 0x77, 0x99, 0xb0, 0xcb, 0x46, 0xca,
    0x45, 0xcf, 0x4a, 0xde, 0x79, 0x8b, 0x86, 0x91, 0xa8, 0xe3, 0x3e, 0x42, 0xc6,
    0x51, 0xf3, 0x0e, 0x12, 0x36, 0x5a, 0xee, 0x29, 0x7b, 0x8d, 0x8c, 0x8f, 0x8a,
    0x85, 0x94, 0xa7, 0xf2, 0x0d, 0x17, 0x39, 0x4b, 0xdd, 0x7c, 0x84, 0x97, 0xa2,
    0xfd, 0x1c, 0x24, 0x6c, 0xb4, 0xc7, 0x52, 0xf6, 0x01
```

```
};



// PolyToExpo[x] donne la puissance k de w telle que x=w^k,
// ou x est donne sous forme polynomiale
const int PolyToExpo[256] = {
    0xff, 0x00, 0x19, 0x01, 0x32, 0x02, 0x1a, 0xc6, 0x4b, 0xc7, 0x1b, 0x68, 0x33,
    0xee, 0xdf, 0x03, 0x64, 0x04, 0xe0, 0x0e, 0x34, 0x8d, 0x81, 0xef, 0x4c, 0x71,
    0x08, 0xc8, 0xf8, 0x69, 0x1c, 0xc1, 0x7d, 0xc2, 0x1d, 0xb5, 0xf9, 0xb9, 0x27,
    0x6a, 0x4d, 0xe4, 0xa6, 0x72, 0x9a, 0xc9, 0x09, 0x78, 0x65, 0x2f, 0x8a, 0x05,
    0x21, 0x0f, 0xe1, 0x24, 0x12, 0xf0, 0x82, 0x45, 0x35, 0x93, 0xda, 0x8e, 0x96,
    0x8f, 0xdb, 0xbd, 0x36, 0xd0, 0xce, 0x94, 0x13, 0x5c, 0xd2, 0xf1, 0x40, 0x46,
    0x83, 0x38, 0x66, 0xdd, 0xfd, 0x30, 0xbf, 0x06, 0x8b, 0x62, 0xb3, 0x25, 0xe2,
    0x98, 0x22, 0x88, 0x91, 0x10, 0x7e, 0x6e, 0x48, 0xc3, 0xa3, 0xb6, 0x1e, 0x42,
    0x3a, 0x6b, 0x28, 0x54, 0xfa, 0x85, 0x3d, 0xba, 0x2b, 0x79, 0x0a, 0x15, 0x9b,
    0x9f, 0x5e, 0xca, 0x4e, 0xd4, 0xac, 0xe5, 0xf3, 0x73, 0xa7, 0x57, 0xaf, 0x58,
    0xa8, 0x50, 0xf4, 0xea, 0xd6, 0x74, 0x4f, 0xae, 0xe9, 0xd5, 0xe7, 0xe6, 0xad,
    0xe8, 0x2c, 0xd7, 0x75, 0x7a, 0xeb, 0x16, 0x0b, 0xf5, 0x59, 0xcb, 0x5f, 0xb0,
    0x9c, 0xa9, 0x51, 0xa0, 0x7f, 0x0c, 0xf6, 0x6f, 0x17, 0xc4, 0x49, 0xec, 0xd8,
    0x43, 0x1f, 0x2d, 0xa4, 0x76, 0x7b, 0xb7, 0xcc, 0xbb, 0x3e, 0x5a, 0xfb, 0x60,
    0xb1, 0x86, 0x3b, 0x52, 0xa1, 0x6c, 0xaa, 0x55, 0x29, 0x9d, 0x97, 0xb2, 0x87,
    0x90, 0x61, 0xbe, 0xdc, 0xfc, 0xbc, 0x95, 0xcf, 0xcd, 0x37, 0x3f, 0x5b, 0xd1,
    0x53, 0x39, 0x84, 0x3c, 0x41, 0xa2, 0x6d, 0x47, 0x14, 0x2a, 0x9e, 0x5d, 0x56,
    0xf2, 0xd3, 0xab, 0x44, 0x11, 0x92, 0xd9, 0x23, 0x20, 0x2e, 0x89, 0xb4, 0x7c,
    0xb8, 0x26, 0x77, 0x99, 0xe3, 0xa5, 0x67, 0x4a, 0xed, 0xde, 0xc5, 0x31, 0xfe,
    0x18, 0x0d, 0x63, 0x8c, 0x80, 0xc0, 0xf7, 0x70, 0x07
};
```

ASCII control characters (character code 0-31)

The first 32 characters in the ASCII-table are unprintable control codes and are used to control peripherals such as printers.

DEC	OCT	HEX	BIN	Symbol	HTML Number	HTML Name	Description
0	000	00	00000000	NUL	&#000;		Null char
1	001	01	00000001	SOH	&#001;		Start of Heading
2	002	02	00000010	STX	&#002;		Start of Text
3	003	03	00000011	ETX	&#003;		End of Text
4	004	04	00000100	EOT	&#004;		End of Transmission
5	005	05	00000101	ENQ	&#005;		Enquiry
6	006	06	00000110	ACK	&#006;		Acknowledgment
7	007	07	00000111	BEL	&#007;		Bell
8	010	08	00001000	BS	&#008;		Back Space
9	011	09	00001001	HT			Horizontal Tab
10	012	0A	00001010	LF
	Line Feed
11	013	0B	00001011	VT	&#011;		Vertical Tab
12	014	0C	00001100	FF			Form Feed
13	015	0D	00001101	CR
	Carriage Return
14	016	0^{E}	00001110	SO	&#014;		Shift Out / X-On
15	017	0F	00001111	SI	&#015;		Shift In / X-Off
16	020	10	00010000	DLE	&#016;		Data Line Escape
17	021	11	00010001	DC1	&#017;		Device Control 1 (oft. XON)
18	022	12	00010010	DC2	&#018;		Device Control 2
19	023	13	00010011	DC3	&#019;		Device Control 3 (oft. XOFF)
20	024	14	00010100	DC4	&#020;		Device Control 4
21	025	15	00010101	NAK	&#021;		Negative Acknowledgement
22	026	16	00010110	SYN	&#022;		Synchronous Idle
23	027	17	00010111	ETB	&#023;		End of Transmit Block
24	030	18	00011000	CAN	&#024;		Cancel
25	031	19	00011001	EM	&#025;		End of Medium
26	032	1A	00011010	SUB	&#026;		Substitute
27	033	1B	00011011	ESC	&#027;		Escape
28	034	1C	00011100	FS	&#028;		File Separator
29	035	1D	00011101	GS	&#029;		Group Separator
30	036	1E	00011110	RS	&#030;		Record Separator
31	037	1F	00011111	US	&#031;		Unit Separator

ASCII printable characters (character code 32-127)

odes 32-127 are common for all the different variations of the ASCII table, they are called printable characters, represent letters, digits, punctuation marks, and a few miscellaneous symbols. You will find almost every character on your keyboard.

BIBLIOGRAPHIE

[Bih] **Biham E. & Shamir A.** Differential cryptanalysis of the full16-round DES *Advances of Cryptology, proceedings of CRYPTO 91*. [4] Preprint, Dec.1991

[Cho] **Chor B. & Rivest R.L.** A knapsack-type public key cryptosystem based on arithmetic in finite fields. *IEEE Trans Inform Theory* 34 (1998): 901-909

[Cou] **Courtois N.** & **Pieprzyk J.** (2002). "Cryptanalysis of Block Ciphers with Overdefined Systems of Equations". *LNCS* **2501**: 267–287.

[Dae] **Daemen J. & V. Rijmen V.**. *The Design of Rijndael, AES- The Advanced Encryption Standard;* Springer-Verlag, 2002.

[Dav] **David Madore** . Code géometriques algébriques et arithmetique sur le corps finis .

[Den] **Denning D**. & **Sacco G**. Timestampes in key distributed protocols. *Communication of the ACM,* 24(8): 533-535, 1981

[Dif] **Diffie W. & Hellman M.E.**, New Directions in Cryptography, *IEEE Transactions on Information Theory*, Vol. IT-22, Nov. 1976, pp: 644–654.

[Fei] **Feige U., Fiat A. & Shamir A.** Zero-knowledge proofs of identity. *Journal of Cryptology,* 1:77-94, 1988.

[Fer] **Fergusson N., Kelsey J., Lucks S., Schneier B., Stay M., Wagner D. &Whiting D.** The best known attacks against AES/Rijndael. In *Fast Software Encryption*, *Proceedings FSE 2000,* pp. 213–230, Springer Verlag, 2000.

[Kar] **Karn P**. & **Simpson W**., Photuris: Session-Key, Management Protocol, March 1999.

[Ker] **Kerckhoffs A.** La cryptographie militaire. *Journal des sciences militaires.* Vol. IX, pp. 5-83 (Janvier 1883) et pp.161-191 (Février 1883)

[Knu] **Knudsen Lars R.** &, **Mathiassen John E**.: On the Role of Key Schedules in Attacks on Iterated Ciphers. *ESORICS 2004:* 322-334

[Lan] **Langford S.K.** & **Hellman M.E.** , "Linear-Differential Cryptanalysis," in *Advances in Cryptology: Proceedings of Crypto 94*, Yvo G. Desmedt, Editor, Springer-Verlag, Berlin, Lecture Notes in Computer Science #839, pp. 17-25, 1994.

[**Mat]** **Matsui, M.** & **A. Yamagishi**. 1994. A New Cryptanalytic Method for FEAL Cipher. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Science.* E77-A(1): 2-7.

[Mer] **Merkle R.** & **Hellman M**., Hiding Information and Signatures in Trapdoor Knapsacks, *IEEE Trans. Information Theory*, 24(5), September 1978, pp.525–530.

[Mur] **Murphy S.** The cryptanalysis of FEAL 4 with twenty chosen plaintexts. EUROCRYPT 1992: 81–91; J. Cryptology 2(3): 145–154 (1990).

[Nec] **Nechvatal *et al.*** *Report on the development of the Advanced Encryption Standard* (*AES*). Computer Security Division / Information Technology Laboratory National Institute of Standards and Technology (NIST); U.S. Department of Commerce October 2, 2000

[Oor] **Diffie W., van Oorschot P.C., Wiener M.J**. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, vol.2 (1992), pp.107-125.

[RSA Lab] RSA Laboratories, What is cipher block chaining mode? *Home: Standards Initiatives: Public-Key Cryptography Standards (PKCS): PKCS #6: Extended-Certificate Syntax Standard: 2.1 Cryptographic Tools: 2.1.4 What is a block cipher?*

[Sha] **Shannon C.E.,** "Communication Theory of Secrecy Systems," *Bell System Technical Journal,* Vol. 28, No. 4 (October 1949), pp. 656-715.

[Vau] **Vaudenay S.,** Cryptanalysis of of the Chor-Rivest cryptosystem. *Advances in Cryptology - CRYPTO '98 / Lecture Notes in Computer Science*. Volume 1462/1998, pp. 243-256

[Ver] **Vernam Gilbert Sandford**, "Cipher Printing Telegraph Systems For Secret Wire and Radio Telegraphic Communications", Journal of the IEEE, Vol 55, pp109-115 (1926)

[Xue] **Xuejia L.** & **Massey J.L.** A proposal for a new block encryption standard. *Proceedings of the workshop on the theory and application of cryptographic techniques on Advances in cryptology.* Aarhus-Denmark, 1991: 389-404, Springer-Verlag New York, Inc., New York, NY, USA