# L'ORGANISATION DE LA CRYPTOLOGIE MODERNE

*par*

Robert Rolland

***Résumé.*** — Nous donnons ici une vue d'ensemble des primitives cryptographiques utilisées dans la cryptographie actuelle pour construire divers protocoles standards classiques ou moins classiques. Nous essayons de mettre en perspective ces primitives avec le cadre complexe des applications la cryptologie moderne.

## Table des matières

**Partie I. Le cadre général** ..... 2
1. Introduction ..... 2
2. Fonctionnalités à assurer ..... 6

**Partie II. Techniques d'intérêt général** ..... 8
3. Introduction ..... 8
4. Fonction de hachage cryptographique ..... 9
5. Tirer au hasard un nombre occasionnel ..... 10
6. Générateur pseudo-aléatoire ..... 12
7. Générateur de masque ou de clé (KDF) ..... 13
8. Chiffrement ..... 15
9. Échange de clé ..... 21
10. Signature ..... 23
11. Code d'authentification de message ..... 24

**Partie III. Techniques plus spécialisées** ..... 24

***Mots clefs.*** — cryptologie, cryptographie,

12. Généralités ..... 24
13. La signature en aveugle ..... 25
14. Les preuves à divulgation nulle ..... 27
15. Engagement ..... 29
16. Le chiffrement homomorphique ..... 30
17. Le partage de secret ..... 31
18. Le mix net ..... 33

**Partie IV. Tendances actuelles** ..... 34
19. Les considérations de la NSA ..... 34
20. Le chiffrement à clé secrète ..... 35
21. La cryptographie elliptique ..... 39

**Partie V. Annexes** ..... 40
22. Techniques d'enrobage ..... 40
23. Résistance et taille des systèmes ..... 45
24. Les organismes de standardisation ..... 46
Références ..... 48

# PARTIE I
# LE CADRE GÉNÉRAL

## 1. Introduction

La *cryptographie* est passée en quelques années d'une technique confidentielle, exclusivement orientée vers des applications militaires, à une discipline relativement large, au croisement des mathématiques de l'informatique et de l'électronique, qui intervient dans tous les domaines où on doit échanger des données. Elle s'insère alors naturellement aux côtés d'autres techniques informatiques dans la *conception de réseaux sécurisés*. Poussée par l'essor technologique des réseaux informatiques, des supports de type cartes à puce, tokens, RFID, des moyens de télécommunications, la cryptographie a évolué vers des applications civiles industrielles et commerciales. De ce fait la notion de coût est devenue capitale. Inutile de dépenser de l'argent pour protéger une

information sans valeur. Donc il faut en permanence se poser la question du niveau de protection souhaité, compte tenu de la valeur de ce qu'on protège. En outre il faut prendre en compte le coût de la mise en service d'un système de protection (par exemple combien coûte le déploiement d'un système de cartes bancaires) la qualité espérée ainsi que les retombées de cette protection, que ce soit en terme de limitation de la fraude ou en terme d'image de marque. Aussi ne s'étonnera-t-on pas que les industriels demandent aux cryptographes des preuves de la sécurité de leurs systèmes et même de manière plus précise, une quantification de cette sécurité au regard des attaques connues.

L'évolution de la cryptographie est jalonnée par quelques textes fondateurs très importants. Je citerai en premier le texte d'Auguste Kerckhoffs « La cryptographie militaire, Journal des Sciences militaires, vol. IX, pp. 5-38, Janvier 1883 et pp. 161-191, Février 1883 » dont voici un extrait : *« Il faut bien distinguer entre un système d'écriture chiffrée, imaginé pour un échange momentané de lettres entre quelques personnes isolées, et une méthode de cryptographie destinée à régler pour un temps illimité la correspondance des différents chefs d'armée entre eux. Ceux-ci, en effet, ne peuvent à leur gré et à un moment donné, modifier leurs conventions ; de plus, ils ne doivent jamais garder sur eux aucun objet ou écrit qui soit de nature à éclairer l'ennemi sur le sens des dépêches secrètes qui pourraient tomber entre ses mains.*

*Un grand nombre de combinaisons ingénieuses peuvent répondre au but qu'on veut atteindre dans le premier cas ; dans le second, il faut un système remplissant certaines conditions exceptionnelles, conditions que je résumerai sous les six chefs suivants :*

1° *Le système doit être matériellement, sinon mathématiquement, indéchiffrable ;*

2° *Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvénient tomber entre les mains de l'ennemi ;*

3° *La clef doit pouvoir être communiquée et retenue sans le secours de notes écrites, et être changée ou modifiée au gré des correspondants ;*

4° *Il faut qu'il soit applicable à la correspondance télégraphique ;*

5° *Il faut qu'il soit portatif, et que son maniement ou son fonctionnement n'exige pas le concours de plusieurs personnes ;*

6° *Enfin il est nécessaire, vu les circonstances qui en commandent l'application, que le système soit d'un usage facile, ne demandant ni tension d'esprit, ni la connaissance d'une longue série de règles à observer. »*

Un peu plus loin dans le texte, il conclut : *« Eh bien! si l'Administration veut mettre à profit tous les services que peut rendre un système de correspondance cryptographique bien combiné, elle doit absolument renoncer aux méthodes secrètes, et établir en principe qu'elle n'acceptera qu'un procédé qui puisse être enseigné au grand jour dans nos écoles militaires, que nos élèves seront libres de communiquer à qui leur plaira, et que nos voisins pourront même copier et adopter si cela leur convient. »* Ce texte est intéressant à divers titres. Tout d'abord pour les idées qu'il développe sur les qualités qu'on devrait attendre d'un bon système cryptographique ainsi que le principe fondamental, appelé justement « Principe de Kerckhoffs », qui exprime que la sécurité d'un système cryptographique doit reposer sur la résistance de la clé et non pas sur l'hypothèse, presque toujours mise en défaut, que l'adversaire ne connaît pas la conception du système utilisé.

Le deuxième texte que je voudrais citer est celui de Claude Shannon publié en 1949 « Communication theory of secrecy systems, In : Bell System Technical Journal, vol. 328, n. 4, pp. 656-715, 1949 ». Claude Shannon, dans la ligne de ses travaux sur la théorie de l'information introduit la notion de système parfaitement sûr. Nous exposerons dans la suite les principaux résultats de ces travaux. L'idée principale étant qu'un système est parfaitement sûr si la donnée du texte chiffrée n'apporte aucune information sur le texte clair. Autrement dit, la probabilité d'un texte clair est la même que la probabilité de ce même texte clair conditionnée par la connaissance du texte chiffré correspondant. Shannon donne des conditions nécessaires et suffisantes pour qu'un système soit parfaitement sûr. Il apparaît que la construction d'un système parfaitement sûr est quasiment irréalisable pour des systèmes ouverts largement déployés couvrant des applications publiques.

C'est au cours de la deuxième moitié du vingtième siècle que la cryptographie, ainsi entrainée par les développements des divers moyens technologiques et le foisonnement des applications impliquant des échanges de données sécurisées en milieu ouvert, a largement étendu le champ

de ses activités. L'introduction de la cryptographie à clé publique avec d'une part l'article de Whitfield Diffie et Martin Hellman publié en 1976 « New Directions in Cryptography, In : IEEE Transactions on Information Theory, 22(6) pp. 644-654, November 1976 », d'autre part l'introduction du chiffrement RSA en 1977 par Ronald Rivest, Adi Shamir, Leonard Adleman : « A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM, Vol. 21 (2), pp.120–126. 1978. Previously released as an MIT Technical Memo in April 1977. Initial publication of the RSA scheme », ont profondément modifié l'abord de la cryptographie. L'article de Whitfield Diffie et Martin Hellman est une révolution dans la conception des systèmes cryptographiques.

Enfin, si on rajoute au tableau la publication du « Data Encryption Standard (DES) »par le National Institute of Standards and Technology (NIST) dans sa note FIPS PUB 46 (Federal Information Processing Standards Publications) du 15 Janvier 1977, comme standard de chiffrement à clé secrète pour les documents non classifiés, on voit qu'on se retrouve à cette date en possession d'un outillage de primitives cryptographiques de base sur lequel asseoir la cryptographie moderne. Celle-ci s'appuie désormais non pas sur la sécurité parfaite au sens de Shannon, mais sur la sécurité calculatoire, c'est-à-dire l'impossibilité actuelle de mener à terme dans un temps raisonnable les algorithmes qui permettraient de battre en brèche la sécurité du système cryptographique considéré.

Ainsi la cryptographie commerciale est basée sur la sécurité calculatoire. Ceci a des conséquences importantes en terme de sécurité, notamment sur la prévision de la durée de vie de la protection procurée par de tels systèmes. En effet, la sécurité repose sur l'impossibilité actuelle de résoudre un problème difficile pour des instances de taille au moins $n$ à l'instant $T$. Qu'en sera-t-il dans 5 ans, 10 ans, 50 ans ? Peut on affirmer qu'un document chiffré maintenant et dont on voudrait assurer la sécurité pendant une longue période, sera encore protégé quelques années plus tard ? À l'évidence non. Bien sûr, dans l'hypothèse où les algorithmes d'attaques ne changent pas vraiment de nature, seule l'amélioration des implémentations et l'accélération des moyens de calculs interviennent. On peut alors tenter une extrapolation et prévoir à peu près les tailles

des systèmes, des clés, à utiliser pour projeter cette protection dans le futur. Mais si entre temps on découvre des algorithmes dont la complexité est radicalement plus faible, tout s'écroule. La conclusion la plus raisonnable est qu'en matière de cryptographie, à usage commercial, largement déployée, il ne faut pas concevoir des systèmes qui ont besoin de secret ayant une longue durée de vie. Il me semblerait très risqué de protéger des documents devant rester secrets 50 ans par des systèmes de ce type.

Nous avons parlé jusqu'à présent de la cryptographie en mettant en avant le chiffrement des données, qui est son activité historique principale. Cependant, de nouvelles fonctionnalités sont apparues, qui ont enrichi le domaine. Ces extensions concernent l'authentification, la signature, l'intégrité des données, et bien d'autres choses développées pour les besoins de la sécurité informatique, des échanges commerciaux, du vote, de la monnaie électronique, et tout un foisonnement d'applications qui apparaisent journellement. Nous introduirons dans la suite quelques unes des principales fonctionalités requises par les applications et nous verrons comment construire effectivement des systèmes aptes à les assurer.

## 2. Fonctionnalités à assurer

Pour bien comprendre les fonctionnalités à assurer et les contextes qui requièrent ces fonctionnalités, nous partirons de quelques exemples. Commençons par l'exemple de SSH (Secure Shell) qui permet d'assurer un accès sécurisé sur un compte situé sur un hôte distant. L'utilisateur, depuis un terminal voulant se connecter sur son compte situé sur l'hôte distant sera amené à un moment ou un autre à prouver qu'il a les droits d'accès au compte, par exemple par système login mot de passe. Il n'est pas raisonnable de faire transiter en clair le mot de passe de l'utilisateur sur la ligne de transmission qui peut être écoutée. La première chose à faire est d'établir un canal sécurisé entre l'utilisateur et l'hôte distant. L'utilisateur qui dans un premier temps n'a pas à prouver son identité veut en revanche être sûr que l'hôte distant est bien le bon, et qu'il pourra lui transmettre en toute confiance les informations confidentielles du login. Ainsi on a déjà un certain nombre de fonctionnalités à assurer :

(1) authentification de l'hôte auprès de l'utilisateur qui doit être sûr qu'il établit le tunnel sécurisé avec le bon correspondant ;

(2) établissement d'un canal sécurisé, c'est-à-dire établissement d'un système de chiffrement efficace entre les deux parties communicantes ; on peut dire que ceci peut être réalisé par un échange de clé secrète pour un système de chiffrement à clé secrète ;

(3) le chiffrement à clé secrète lui-même qui correspond au fonctionnement du canal sécurisé ;

(4) enfin pour finir cet exemple la connexion sur le compte n'est complète qu'après que l'utilisateur se soit identifié, soit par un système de login password, soit par un système de signature.

Remarquons que dans cet exemple l'hôte distant et le terminal cherchant à se connecter sur un compte de cet hôte ne jouent pas un rôle symétrique.

Le deuxième exemple que nous prendrons est celui de la signature du courrier sur internet. Il s'agit de signer numériquement de manière incontestable (fonctionnalité de non-répudiation) un courrier, un bon de commande, une déclaration d'impôts etc. Plusieurs systèmes sont en concurrence pour réaliser de telles opérations, par exemple pgp, ssl/tls. Ces systèmes sont des systèmes complexes permettant de faire diverses opérations, mais la fonctionnalité sur laquelle nous insistons ici est la signature numérique. Celle-ci est indissociablement liée au fonctionnement des serveurs de clés, à la certification des clés, soit de manière hiérarchique dans les systèmes plutôt fermés, soit par diffusion dans les systèmes largement ouverts. Dans les systèmes hiérarchiques, les clés des utilisateurs sont signées par une autorité de certification qui atteste que la clé en question appartient à une personne identifiée, qui est bien celle qu'elle prétend être. Dans les systèmes largement ouvert, le lien entre la clé et une identité est plus ténu. Chacun fait signer sa clé par des proches qui le connaissent et la confiance que quelqu'un peut attribuer à une clé dépend de sa connaissance ou non des quelques signataires de la clé en question.

Ainsi à la lumière des exemples précédents, nous voyons se dégager un certain nombre de fonctionnalités principales indispensables dont nous faisons une synthèse maintenant.

– La confidentialité,

– l'intégrité des données,
– l'auhentification, l'identification
– la signature
– la gestion des clés

Ces fonctionnalités ne sont pas les seules, de nombreuses variantes apparaissent régulièrement à l'occasion de nouveaux problèmes de sécurité posés par des applications spécifiques.

# PARTIE II
# TECHNIQUES D'INTÉRÊT GÉNÉRAL

## 3. Introduction

Les *protocoles* sont les structures de la cryptographie qui permettent, en s'appuyant sur des techniques cryptographiques, d'élaborer des programmes complexes permettant, dans les applications concrètes, d'atteindre un ou plusieurs des buts suivants qui concernent la sécurité dans les échanges de données :

(1) Assurer la confidentialité des données.

(2) Assurer l'authentification des diverses composantes qui participent à un protocole. Ce peut être des individus, des machines, des serveurs, des parties d'un réseau etc.

(3) Assurer la signature de messages divers.

(4) Assurer l'intégrité des données ou tout au moins le contrôle de cette intégrité.

(5) Assurer la protection du matériel cryptographique lui-même, en particulier des clés, qui comme on s'en doute, sont les secrets qui sont au centre des techniques cryptographiques.

(6) Prouver qu'on connaît un secret sans rien dévoiler qui permette de le calculer en pratique.

Les *protocoles cryptographiques*, dont les fonctionnements exacts sont précisés dans des *standards* (RFC, PKCS, FIPS, IEEE, ISO, ITU (voir

l'annexe 24)) utilisent un certain nombre de *techniques cryptographiques* aptes à assurer les fonctions de base dont ils ont besoin.

Si on veut se faire une idée des techniques généralistes les plus utilisées, il suffit de regarder quelques standards connus, par exemple *openssh*, *gnupg*, *openssl*, *IPSec*, *ISO-18033-2*.

Précisons aussi que la plupart des objets cryptographiques dans les protocoles courants sont dans un format de type *suite d'octets*. C'est le cas des *textes clairs*, des *textes chiffrés*, des *clés publiques*, *clés privées*, *clés secrètes*, des *empreintes*, des *codes d'authentification de message*, des *appendices de signature*. Quand on a besoin de ces données sous une autre forme, par exemple sous forme d'entiers, pour mener à bien un certain nombre de calculs arithmétiques, ou pour avoir des sorties lisibles, on utilise alors des fonctions de traduction ainsi qu'il est expliqué dans [**1**].

Pour toutes ces raisons, introduisons $B = \{0,1\}^8$ l'ensemble des octets et $B^*$ l'ensemble des mots construits sur l'alphabet $B$, c'est-à-dire l'ensemble des suites finies d'octets.

Il faut noter aussi que certaines techniques qui apparaissent sous des noms différents dans la littérature sont parfois très semblables, sinon quasiment identiques, sur le plan formel. Il faut toutefois remarquer qu'en cryptographie l'aspect important de la *réalisation pratique* conduit à distinguer des objets de fonctionnalité équivalente, mais dont la réalisation procède de techniques tout à fait distinctes. L'exemple des générateurs pseudo-aléatoires est de ce point de vue très instructif.

Nous décrivons par la suite, un certain nombre de techniques cryptographiques indispensables à la cryptographie moderne, et nous citons à titre d'exemples, à étudier plus en détail par ailleurs, quelques primitives cryptographiques qui permettent la mise en œuvre de ces techniques.

## 4. Fonction de hachage cryptographique

Soit $k$ un nombre fixé qui représentera un nombre d'octets. Une fonction de hachage est une application :

$$h : B^* \longrightarrow B^k$$

qui vérifie un certain nombre de propriétés :

(1) résistance à la préimage : soit $y \in B^k$. Il est impossible en pratique de trouver un $x$ tel que $h(x) = y$.

(2) résistance aux collisions : il est impossible de calculer en pratique deux éléments distincts $u$ et $v$ tels que $h(u) = h(v)$.

En réalité, une fonction de hachage cryptographique ne peut pas hacher des textes aussi longs qu'on veut. Il y a une limite d'utilisation. Ainsi, on pourrait préciser qu'une fonction de hachage cryptographique est une fonction de $B^*_{l(k)}$ (ensemble des suites ayant au plus $l(k) - 1$ octets) dans $B^k$.

Prenons à titre d'exemple la primitive de hachage *SHA256* qui pour un message de longueur inférieure à $2^{64}$ bits fournit une empreinte de 256 bits ($k = 32$, $l(k) = 2^{61}$).

La résistance de cette primitive est pour le moment de l'ordre de 128 (cf. annexe 23). Remarquons aussi que *SHA256* permet de traiter des fichiers de taille inférieure à $2^{31}$ Go, ce qui permet tout de même des fichiers assez énormes.

Dans cette gamme de fonctions de hachage proposée par le *NIST* (*N*ational *I*nstitute of *S*tandards and *T*echnologie) on dispose de *SHA1* (déconseillé), *SHA224* (pour utiliser avec le *triple DES*), *SHA256* (pour utiliser avec par exemple *AES128*), *SHA384* (pour utiliser avec par exemple *AES192*), *SHA512* (pour utiliser avec par exemple *AES256*).

## 5. Tirer au hasard un nombre occasionnel

De nombreuses applications ont besoin d'un nombre aléatoire occasionnel, c'est-à-dire qui soit pris de manière isolée, hors d'un contexte de tirage systématique. C'est par exemple le cas quand on veut lors d'un protocole tirer une fois pour toutes *un* nombre qui va servir de valeur initiale : *germe* d'un générateur pseudo-aléatoire, *valeur initiale* d'un chiffrement à clé secrète, *clé secrète de session*, etc. C'est ce qu'on utilise, lorsqu'on fait un tirage de quelques centaines de bits, qui n'est pas relié systématiquement, dans le déroulement du protocole, à de nombreux tirages répétés.

Un tirage occasionnel est souvent obtenu par un processus physique, doublé d'une fonction de correction apte à répartir la « quantité d'aléa » fournie par le processus physique. La plupart des bons systèmes d'exploitation fournissent un « réservoir d'aléa » alimenté par exemple par le mixage de divers apports extérieurs plus ou moins imprévisibles que sont les déplacements, les clics de la souris, les frappes au clavier, les informations de l'horloge, le bruit des composants etc.

Sous le système *linux* par exemple on dispose du périphérique :

*/dev/random* (ou */dev/urandom*)

qui joue le rôle de ce réservoir.

À titre d'exemple, voici comment on peut procéder sous linux, avec OpenSSL installé :

```
$ head -c 128 /dev/random | openssl dgst -sha256
   -binary > seed.bin
```

On peut voir ce que ça a donné avec od :

```
$ od -t x seed.bin
0000000 6026882e a19a8e6b feaabdc3 8bc4eef9
0000020 c441e65d 213adeaf bb383900 f7eca25f
```

on voit les 32 octets (256 bits qui ont été renvoyés). Attention ici les données renvoyées sont des entiers sur 32 bits écrits en hexadécimal dans le mode *little endian*. Autrement dit si on écrit la suite d'octets dans l'ordre des adresses croissantes on obtient :

$$2e, 88, 26, 60, 6b, 8e, 9a, a1, \cdots, 5f, a2, ec, f7.$$

Si on veut que le fichier renvoyé soit en base64 (voir l'annexe (22.2) :

```
$ head -c 128 /dev/random | openssl dgst -sha256
  -binary | openssl enc -base64 > seed.b64
```

Dans ces commandes, on a commencé par extraire 128 octets (c'est-à-dire 1024 bits) du périphérique /dev/random par « head -c 128 /dev/random », puis la commande « openssl dgst » est utilisée avec les options « -sha256 -binary ». pour effectuer un hachage, ici avec *sha256*,

la sortie étant binaire (256 bits). La redirection > dirige la sortie dans le fichier binaire « seed.bin ».

Si on rajoute dans le *pipe* la commande « openssl enc » avec l'option « -base64 », la sortie binaire précédente est encodée en base64 pour fournir une sortie ASCII. Cette sortie est redirigée dans le fichier affichable « seed.b64 ».

## 6. Générateur pseudo-aléatoire

Un générateur pseudo-aléatoire est en général construit à partir d'une fonction mathématique déterministe, qui calcule par récurrence une suite d'états, dont est tirée la suite pseudo-aléatoire. La suite d'états est elle même une suite de valeurs internes cachées. La valeur initiale de l'état est appelée le germe. Si un ennemi ignore la valeur du germe, il doit lui être impossible en pratique, à partir des premiers termes d'une *suite pseudo-aléatoire cryptographique*, de prévoir le terme suivant. Cette propriété, grâce au théorème de Yao (voir un exposé dans [**6**]) est en un certain sens équivalente au fait qu'il doit être en pratique calculatoirement impossible de distinguer une répartition de bits construits avec le générateur pseudo-aléatoire, d'une répartition purement aléatoire.

Un cas typique est celui où on dispose d'une fonction $f$ et d'une fonction à sens unique $F$ qui produit 1 bit. À partir du germe $s_0$ on calcule une suite cachée d'états : $s_k = f(s_{k-1})$, et la suite pseudo-aléatoire $x_k = F(s_k)$.

Voici l'exemple du générateur de Blum-Blum-Shub :

On se donne un produit $n = pq$ de deux grands nombres premiers de Blum (de la forme $4k + 3$) où $p$ et $q$ sont gardés secrets. On calcule alors à partir d'un germe $1 \leq s_0 < n$, nombre au hasard occasionnel, gardé lui aussi secret les états successifs $s_k$ par la formule de récurrence :

$$s_{k+1} = s_k^2 \mod n.$$

De l'état $s_k$ (qui est une valeur interne secrète) on tire le $k^{ieme}$ bit $u_k$ de la suite pseudo-aléatoire en prenant le bit de poids faible de $s_k$ :

$$u_k = lsb(s_k) = s_k \mod 2.$$

Le germe lui-même est construit en utilisant un procédé décrit dans le paragraphe 5. Par exemple pour le générateur de Blum-Blum-Shub, comme on a intérêt pour diverses raisons à avoir dès le début un germe $s_0$ qui soit un résidu quadratique modulo $n$, on peut par exemple construire un nombre occasionnel $a$ de 256 bits, ainsi qu'on l'a fait au paragraphe 5, puis prendre $s_0 = a^4 \mod n$. La taille initiale de $a$ permet d'éviter toute attaque par force brute, et la construction de $s_0$ assure que c'est bien un résidu quadratique.

Ce générateur ne peut pas convenir, à cause de sa lenteur relative, pour l'utilisation principale d'un *véritable* générateur pseudo-aléatoire construit spécifiquement pour cette utilisation, c'est-à-dire pour la génération en temps réel d'un flot de bits pouvant servir par exemple à du chiffrement par flot et qui iraient plus vite que du chiffrement par bloc.

Pour des applications où le flot de bits créé n'a pas besoin d'être aussi volumineux ni aussi rapide, on peut réaliser des générateurs pseudo-aléatoires avec des circuits de chiffrement par bloc ou des fonctions de hachage (cf. par exemple la construction d'un KDF (key derivation function)).

Cependant, on préfèrerait avoir des procédés spécifiques qui peuvent travailler réellement par flot et non par bloc. Hélas, en l'état actuel, ce type de circuit n'est pas (encore) très conseillé (voir par exemple les failles du protocole WEP, utilisé au début du Wi-Fi ([**2**, pp. 197-205], ainsi que l'analyse faite par la DCSSI [**3**, page 18]). Il existe cependant un travail très sérieux qui a été fait, par le projet européen eStream. Ce projet a rendu son rapport fixant un portofoglio de quatre systèmes en software et quatre systèmes en hardware (dont un a été cassé depuis).

http ://www.ecrypt.eu.org/stream/

## 7. Générateur de masque ou de clé (KDF)

C'est en quelque sorte un générateur pseudo-aléatoire, en ce sens qu'à partir d'une suite d'octets (qu'on peut considérer comme un germe) on

récupère une suite d'octets pseudo-aléatoire d'une longueur donnée. Cependant en général, compte tenu de la taille de la sortie, il n'est pas indispensable d'avoir une fonction de générateur de flot aléatoire spécifique. Une construction à l'aide d'une fonction de hachage convient parfaitement.

Un générateur de masque est aussi proche d'une fonction de hachage dans la mesure où à partir d'une suite d'octets en entrée, on récupère une suite d'octets de taille donnée, à part ici que la taille peut être variable. De manière plus formelle un générateur de masque est une application $H$ de $B^* \times \mathbb{N}$ dans $B^*$ qui à une suite finie d'octets $x$ et à un entier $n$, fait correspondre une suite d'octets $H(x, n)$ de longueur $n$.

Un générateur de masque est aussi appelé un *KDF* (Key Derivation Function) car il permet de construire à partir d'une suite d'octets, une ou plusieurs clés de taille donnée.

Voici une façon classique de construire un générateur de masque $H$ à partir d'une fonction de hachage $h$. Notons $t$ la taille en octets de l'empreinte calculée par cette fonction de hachage :

$$t := \text{TailleOct}(h(u)).$$

La valeur de $H(x, n)$ se calcule de la façon suivante :

(1) on pose :

$$k = \left\lceil \frac{n}{t} \right\rceil$$

(2) pour $0 \leq j \leq k - 1$ on note $y_j$ la suite de 4 octets dont la valeur correspondante en nombre entier est le nombre $j$. c'est-à-dire par exemple :

$$y_0 = 00000000000000000000000000000000,$$

$$y_1 = 00000000000000000000000000000001$$

(3) puis on calcule $z_j = h(x||y_j)$,
(4) et enfin $H(x, n)$ est obtenu en prenant les $n$ premiers octets de :

$$z_0||z_1||\cdots||z_{k-1}.$$

## 8. Chiffrement

On rappelle que le chiffrement est divisé en deux grandes catégories toutes deux indispensables : le chiffrement à clé secrète et le chiffrement à clé publique.

**8.1. Clé secrète.** — Le chiffrement à clé secrète est lui-même divisé en deux types : le chiffrement à flot (au fil de l'eau) et le chiffrement par blocs. Dans les deux cas, on dispose de deux fonctions publiques : la fonction de chiffrement $\mathcal{E}$ qui à un texte clair et une clé secrète $K$ fait correspondre un texte chiffré $y$, et la fonction de déchiffrement $\mathcal{D}$ qui à un texte $y$ et une clé $K$ fait correspondre le texte clair $x$ si $y$ est le chiffré de $x$ avec la clé $K$, et part en erreur si $y$ n'est pas un chiffré valide avec la clé $K$. Ainsi l'expéditeur et le destinataire doivent posséder la même clé secrète $K$. Celle-ci est dans la plupart des protocoles construite aléatoirement à la volée (clé de session) et échangée avec une méthode d'échange de clé basée sur de la cryptographie à clé publique.

*8.1.1. Chiffrement à flot.* — L'obtention d'un système de chiffrement à flot spécifique, c'est-à-dire qui ne soit pas simulé par du chiffrement par bloc, est tributaire de la construction d'un générateur pseudo-aléatoire performant et suffisamment sûr. On ne peut que rappeler ici à ce propos la situation décrite au paragraphe 6.

Si on dispose d'un générateur pseudo-aléatoire alors le principe du chiffrement est celui du masque jetable. Appelons $x_1, x_2, \cdots, x_n, \cdots$ le flot de bits construit par le générateur à partir du germe $s_0$. Le germe $s_0$ constitue la clé secrète. Ce flot est aussi connu du *destinataire* grâce au partage de la clé secrète $s_0$. Si l'exéditeur veut envoyer le texte clair $b_1, b_2, \cdots, b_n, \cdots$, il masque ce flot avec le flot $x_1, x_2, \cdots, x_n, \cdots$, pour obtenir le texte chiffré :

$$c_1, c_2, \cdots, x_n, \cdots, \qquad \text{où } c_i = b_i \oplus x_i.$$

*8.1.2. Chiffrement par bloc.* — Pour décrire le chiffrement par bloc, il faut distinguer deux parties :

(1) Le circuit proprement dit qui lorsqu'on lui fournit une clé secrète $k$ et un bloc de texte clair $b$ (bloc de la taille exacte de l'entrée du circuit)

renvoie un bloc chiffré $c = \mathcal{F}(b, k)$ et qui réciproquement, si on lui fournit un bloc chiffré $c$ et la clé secrète $k$ renvoie le bloc de texte clair $b$. Bien entendu, tout est public sauf la clé secrète (et le texte clair ! !).

(2) Le mode d'utilisation du circuit qui explique comment on chiffre un texte clair en le découpant en blocs et en effectuant un certain nombre d'autres opérations.

La taille des clés recommandée actuellement est 128 bits ou même 256 bits pour une sécurité de très haut niveau. En l'état actuel, une clé de 80 bits résiste encore, mais il faut prendre des précautions si on veut chiffrer des documents ayant une longue durée de vie. Si dans un futur plus ou moins lointain, un circuit est cassé, tous les chiffrés que « l'ennemi » aura conservés seront accessibles. Un autre paramètre important est la taille du bloc de données traitée par le circuit. Là aussi une taille de 128 bits est conseillée. Nous pouvons citer par exemple les circuits *AES128* (128 bits de clé, 128 bits de données), *AES256* (256 bits de clé, 128 bits de données).

Le *NIST* décrit dans son standard de chiffrement par bloc plusieurs modes d'utilisation. Nous nous contenterons d'indiquer ici le plus utilisé : *CBC* (cipher block chaining) ainsi qu'un autre mode très simple, très efficace et très sûr, bien adapté au nouveau standard *AES*, le mode *CTR* (counter). Si le texte clair n'a pas exactement la taille qu'il faut afin que le découpage en blocs tombe juste, on procèdera au *padding* de manière à avoir un nombre exact de blocs.

On remarque que le mode *CTR* simule à peu près un chiffrement à flot, le masque étant construit par chiffrement du compteur.

(1) **Cipher Block Chaining Mode : CBC**

**Le chiffrement**. Le texte clair est découpé en blocs $P_i$ de taille $t_d$. En plus du texte clair $P = P_1 P_2 \cdots P_k$, on a besoin là encore d'un bloc initial tiré aléatoirement $C_0$. On calcule alors successivement

$$\begin{aligned} C_1 &= \mathcal{E}_K(C_0 \oplus P_1), \\ C_2 &= \mathcal{E}_K(C_1 \oplus P_2), \\ \cdots &= \cdots\cdots, \\ C_k &= \mathcal{E}_K(C_{k-1} \oplus P_k). \end{aligned}$$

Le message chiffré est alors $C = C_0C_1C_2 \cdots C_k$.

Le **déchiffrement**. Il s'effectue en calculant successivement

$$P_1 = \mathcal{D}_K(C_1) \oplus C_0,$$
$$P_2 = \mathcal{D}_K(C_2) \oplus C_1,$$
$$\cdots = \cdots\cdots,$$
$$P_k = \mathcal{D}_K(C_k) \oplus C_{k-1}.$$

(2) **Counter Mode : CTR**

**Le chiffrement**. Le texte clair est découpé en blocs de taille $t_d$. Dans ce mode on dispose d'un compteur de taille $t_d$. On tire au sort une valeur initiale $CTR_1$ pour ce compteur. On calcule alors

$$Z_1 = \mathcal{E}_K(CTR_1) \qquad C_1 = P_1 \oplus Z_1.$$

Puis on incrémente le compteur, c'est-à-dire qu'on calcule

$$CTR_2 = CTR_1 + 1 \mod 2^{t_d}.$$

Plus généralement, on calcule successivement

$$CTR_i = CTR_{i-1} + 1 \mod 2^{t_d},$$
$$Z_i = \mathcal{E}_K(CTR_i),$$
$$C_i = P_i \oplus Z_i.$$

Le texte chiffré transmis est constitué de la valeur $CTR_1$ du compteur suivi des blocs $C_1C_2 \cdots C_k$.

**Le déchiffrement**. Il se fait en calculant succéssivement à partir de la valeur initiale du compteur

$$Z_i = \mathcal{E}_K(CTR_i),$$
$$P_i = C_i \oplus Z_i,$$
$$CTR_{i+1} = CTR_i + 1 \mod 2^{t_d}.$$

On trouvera ainsi dans des applications telles que *openssl*, des dénominations de chiffrement, telles que *AES128-CBC*.

Actuellement un mode qui combine à la fois le mode counter avec un code MAC a vu le jour, il s'agit du Galois Counter Mode. Ce mode d'utilisation est très performant et très sûr. Nous en donnons une description dans un paragraphe ultérieur (cf. paragraphe 20).

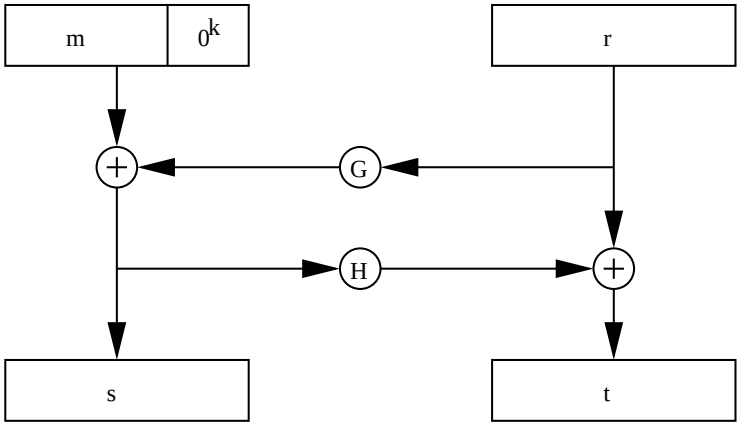
**8.2. Clé publique.** — Pour le chiffrement à clé publique, chaque utilisateur $X$ posséde une paire de clés : la clé publique $e_X$ mise à la disposition de tous (sur un serveur par exemple) et la clé privée $d_X$ connue seulement de son propriétaire $X$. On dispose alors de deux fonctions publiques, une fonction de chiffrement $\mathcal{E}$ qui permet à tout le monde de chiffrer un message $x$ à destination de l'utilisateur $X$ en calculant $y = \mathcal{E}(x, e_X)$ et une fonction de déchiffrement $\mathcal{D}$ qui permet à $X$ de déchiffrer grâce à sa clé privée en calculant $x = \mathcal{D}(y, d_X)$. Cette technique, qui s'applique à des messages courts, à cause de la relative lenteur des primitives cryptographiques correspondantes, sert essentiellement à échanger des clés.

*8.2.1. RSA.* — La primitive de chiffrement à clé publique le plus connu est *RSA*. Il est basé sur la difficulté de la factorisation. On n'insistera pas sur l'aspect mathématique de *RSA*, qui doit être étudié en profondeur dans un autre cadre.

Là encore la primitive de base ne suffit pas. Pour chiffrer proprement avec *RSA*, il ne suffit pas de prendre le message et de lui appliquer la fonction de chiffrement *RSA*. D'une part, ceci conduirait à un chiffrement déterministe, c'est-à-dire que si on chiffre deux fois le même message on obtient deux fois le même chiffré, ce qui peut avoir des inconvénients, d'autre part, dans le cas de messages très courts on risque des attaques spécifiques, puisqu'on ne bénéficie plus de la totale diversité fournie par la taille de l'espace des messages proposée en théorie par *RSA*. Il convient donc de préparer le message à chiffrer par une phase dite « phase de padding (bourrage en français) ».

En 1994, M. Bellare et P. Rogaway ont introduit le système *OAEP* (Optimal Asymetric Encryption Padding) qui permet cette phase de padding.

*OAEP* est un moyen de préparer le message à chiffrer ("padding" du message) qui n'est pas spécifique à l'encapsulation d'une clé, mais qui peut être employé pour un chiffrement asymétrique général.




Le message $m$ est rallongé en un message $M$ (sur le dessin on a rajouté des 0, mais ça peut être autre chose). Un nombre aléatoire $r$ est tiré au sort. On dispose par ailleurs d'une fonction publique de hachage $H$ et d'un *générateur de masque* (ou *KDF*) public $G$. La fonction $H$ peut bien entendu être remplacée sans inconvénient par un générateur de masque. On calcule alors :

$$s = G(r) \oplus M,$$
$$t = H(s) \oplus r.$$

Le message encodé est alors la concaténation $x = s||t$. Le chiffré est

$$y = RSA_e(x).$$

Le déchiffrement s'effectue à partir de $y$ en calculant d'abord $x$ par déchiffrement *RSA* :

$$x = RSA_d(y).$$

Ensuite les valeurs $s$ et $t$ dont on connait les longueurs respectives sont extraites de $x$. On calcule alors :

$$r = H(s) \oplus t,$$
$$M = G(r) \oplus s.$$

Le message $m$ est extrait de $M$.

Pour terminer sur *RSA*, disons que la primitive elle même peut craindre quelques attaques dans des cas qu'on peut controler, par exemple :

(1) Les facteurs $p$ et $q$ du module $n$ sont trop proches.

(2) $p-1$ (ou $q-1$) est friable, c'est-à-dire n'a que des petits facteurs premiers.

(3) L'exposant de chiffrement $d$ est petit (par exemple sa taille en nombre de bits a moins du quart de la taille du module $n$).

*8.2.2. Elgamal.* — Le chiffrement d'*Elgamal* est lui, basé sur la difficulté du problème du logarithme discret. Il est utilisé dans certains contextes, notamment dans des protocoles de vote électronique par exemple à cause de la possibilié de modifier « en vol » un chiffré sans compromettre son déchiffrement par le destinataire.

Soit $q$ un nombre premier de Sophie Germain c'est-à-dire tel que $p = 2q+1$ soit aussi un nombre premier. On supposera que $p$ soit assez grand (par exemple 2048 bits) pour que le problème du logarithme discret soit difficile dans le groupe multiplicatif $G = \mathbb{Z}/p\mathbb{Z}^*$. On ne sait pas prouver grand chose sur les nombres de Sophie Germain. En particulier on ne sait même pas s'il y en a une infinité. Mais on arrive toujours en temps raisonnable, à tirer un tel nombre au hasard. Comme $p-1 = 2q$ les seuls ordres possibles pour les éléments de $\mathbb{Z}/p\mathbb{Z}^*$ sont 1, 2, $q$ et $2q$. Il y a un seul élément d'ordre 1 (c'est 1) et 1 élément d'ordre 2 (c'est $-1$, c'est-à-dire $p-1$). Ces éléments là sont reconnaissables. Les autres sont d'ordre $q$ ou d'ordre $2q$. Si on tire au sort un élément de $1 < \alpha < p-1$ il y a une chance sur deux qu'il soit d'ordre $q$ (pour tester, on élève à la puissance $q$ par l'algorithme square and multiply). On va donc rapidement arriver à construire un élément $\alpha$ d'ordre $q$. Notons $H$ le sous-groupe de $G$ engendré par $\alpha$. Remarquons que $H$ est le sous-groupe des résidus quadratiques. On tire au sort une clé privée $1 \leq a < q$. La clé publique est $\beta = \alpha^a$. Si on a un texte clair $0 \leq m < q$ à transmettre, alors ou bien $m \in H$, et on le laisse tel quel, et on pose $x = m$, ou bien $m$ n'est pas un résidu quadratique, mais alors $p-m$ en est un et on pose $x = p-m$. C'est $x$ (qui lui est dans $H$) qu'on va chiffrer en calculant :

$$y_1 = \alpha^k \mod p,$$
$$y_2 = x\beta^k \mod p,$$

où $k$ est un nombre tiré au sort qu'on ne doit pas réutiliser. Le destinataire, qui bien entendu est le propriétaire de la clé secrète $a$, peut donc calculer :

$$z = y_1^a \mod p = \beta^k \mod p.$$

Une fois qu'il a calculé $z$, il obtient :

$$x = y_2 z^{-1} \mod p.$$

Il lui suffit maintenant de tester si $0 \leq x < q$ ou si $q \leq x < p$ pour savoir si $m = x$ ou si $m = p - x$.

## 9. Échange de clé

Une primitive de chiffrement à clé publique peut servir à l'échange d'une clé secrète. Cependant plusieurs protocoles préfèrent utiliser des méthodes spécifiques (échange de clé de Diffie-Hellman par exemple) qui sont souvent mieux à même d'assurer plus de sécurité rétroactive. Nous parlerons donc de deux techniques d'échange de clé :

(1) Diffie-Hellman.
(2) *KEM* : key encapsulation mechanism.

**9.1. Échange de clé de Diffie-Hellman.** — Il s'agit donc, comme il est exigé par de nombreux protocoles, d'échanger entre deux interlocuteurs $A$ et $B$ une clé secrète $K$ de taille $t$ octets. Pour cela $A$ et $B$ disposent d'un groupe cyclique fini $G$ et d'un générateur $a$ de ce groupe (les éléments de $G$ sont donc, si on note multiplicativement l'opération du groupe, $1, a, a^2, \cdots, a^{s-1}$ où $s$ est l'ordre de $G$). Prenons par exemple pour $G$ le groupe multiplicatif $(\mathbb{Z}/p\mathbb{Z})^*$ où $p$ est un grand nombre premier et $a$ un élément générateur de ce groupe (mais ça pourrait être aussi un générateur d'un grand sous-groupe de $(\mathbb{Z}/p\mathbb{Z})^*$).

Voici comment se passe (de manière schématique) l'échange. Les calculs indiqués sont faits dans le groupe $G$, donc dans notre exemple modulo $p$.

- Données publiques : le groupe $G = (\mathbb{Z}/p\mathbb{Z})^*$, un générateur $a$ de ce groupe, un générateur de masque $h$.
- $A$ tire au sort un entier $n$ tel que $1 < n < p - 1$ et le garde secret.
- $A$ envoie $a^n$ à $B$ (calcul fait dans le groupe, donc ici modulo $p$).

- $B$ tire au sort un entier $m$ tel que $1 < m < p-1$ et le garde secret.
- $B$ envoie $a^m$ à $A$.
- $A$ calcule $s = (a^m)^n$.
- $B$ calcule $s = (a^n)^m$.
- $A$ et $B$ disposent maintenant même $s$

Rappelons que les calculs sont faits modulo $p$ bien sûr. Le nombre $s$ est à peu près de la taille de $p$ et doit certainement être adapté à la taille de la clé commune convoitée. Ceci est fait grâce au générateur de masque :

$$K = h(s, t)$$

où $t$ est la taille en octets de la clé secrète cherchée $K$.

**9.2. Le système *RSA-KEM*.** — Le système *RSA-KEM* utilise un entier positif $KeyLen$ qui représente la longueur en octets de la clé secrète à encapsuler ainsi qu'un algorithme $RSAKeyGen$ de génération de couple de clés *RSA* et un générateur de masque (ou Key Derivation Function) *KDF*.

On doit donc avant toute chose avoir établi un système *RSA* de paramètres $(n, e, d)$ (module $n$, exposant public $e$, exposant privé $d$). On note $\mathcal{L}(n)$ la taille de $n$ en octets.

*RSA-KEM* définit alors deux algorithmes : $RSA-KEM.Encrypt(n, e)$ et $RSA-KEM.Decrypt(n, d, C)$. Le premier encapsule une clé secrète en utilisant le chiffrement *RSA* de clé publique $(n, e)$. Le deuxième retrouve la clé secrète encapsulée dans $C$ en utilisant le déchiffrement RSA de clé privée $(n, d)$.

*9.2.1. Encapsuler la clé secrète.* —
L'algorithme $RSA - KEM.Encrypt(n, e)$ fonctionne de la manière suivante :

1) On génère au hasard un nombre $r \in [0..n[$.
2) On chiffre ce nombre avec *RSA* : $v = RSA_{(n,e)}(r)$.
3) On transforme ce nombre en suite d'octets de taille $\mathcal{L}(n)$ octets :

$$C = I2OSP(v, \mathcal{L}(n))$$

4) On calcule la clé secrète $K = KDF(I2OSP(r, \mathcal{L}(n)), KeyLen)$.
5) On dispose donc de $C$ et $K$.

**Remarque :** bien entendu, seul $C$ est destiné à être transmis au destinataire.

*9.2.2. Décapsuler la clé secrète. —*

1) À partir de $C$ on retrouve $r$ puis $I2OSP(r, \mathcal{L}(n))$ par déchiffrement *RSA* utilisant la clé privée $(n, d)$.

2) On calcule $K = KDF(I2OSP(r, \mathcal{L}(n)), KeyLen)$.

3) On dispose de $K$.

## 10. Signature

La signature est une primitive cryptographique qui garantit l'intégrité d'un message, fournit l'authentification du signataire et assure la non-répudiation. Chaque utilisateur $X$ dispose d'une clé publique $e_X$ et d'une clé privée $d_X$. On dispose aussi d'une fonction de hachage publique $h$. Lorsque $X$ veut signer un message $x$, il utilise sa clé privée et la fonction publique de signature $\mathcal{S}$ pour calculer l'appendice $s = \mathcal{S}(h(x), d_X)$. Il transmet alors le couple $(x, s)$ constitué du message $x$ et de l'appendice $s$. Tout le monde peut vérifier que la signature est valide en utilisant la fonction publique de vérification $\mathcal{V}$ et la clé publique de $X$, par le calcul de $\mathcal{V}((x, s), e_X)$. La signature peut être utilisée pour s'authentifier par signature de messages tests envoyés par le correspondant. Le standard *DSS* (Digital Signature Standard) du *NIST* décrit un standard de signature qui prévoit l'emploi au choix d'une des primitives de base suivantes :

(1) Signature *RSA* basée sur le système RSA qui utilise la difficulté du problème de la factorisation.

(2) Signature *DSA* (Digital Signature Algorithme) qui utilise la difficulté du problème du logarithme discret dans un groupe multiplicatif $\mathbb{Z}/p\mathbb{Z}^*$, et qui est une variante de la signature d'Elgamal.

(3) Signature *ECDSA* (Elliptic Curve Digital Signature Algorithm) qui utilise la difficulté du problème du logarithme discret sur le groupe des points d'une courbe elliptique.

### 11. Code d'authentification de message

Les MAC (ou encore fonctions de hachage à clé secrète) permettent à un utilisateur de s'authentifier et en même temps d'assurer l'intégrité d'un message auprès d'un correspondant qui partage la même clé secrète que lui. On peut construire un MAC avec un circuit de chiffrement à clé secrète. Mais on préfère souvent utiliser un circuit spécifique. Les primitives MAC sont aussi appelées des fonctions de hachage à clé secrète. Cela décrit en effet très bien leur fonctionnement. Un mac peut être considéré comme une fonction de $B^* \times B^k$ dans $B^n$ qui à une suite finie d'octets et une clé secrète fait correspondre un bloc de taille fixée $n$ octets. Bien entendu, en pratique la taille de l'entrée, comme pour une fonction de hachage, est limitée. Le circuit $AES$ utilisé en mode $CBC$, pour lequel on ne garde que la valeur initiale et le dernier bloc, fournit un très bon MAC. Un très bon MAC est aussi fourni dans le mode de chiffrement Galois Counter Mode dont nous avons parlé et auquel nous consacrons dans un chapitre ultérieur un paragraphe (cf. paragraphe 20).

## PARTIE III
## TECHNIQUES PLUS SPÉCIALISÉES

### 12. Généralités

Il s'agit ici de décrire quelques autres techniques cryptographiques qui, en collaboration avec les techniques générales décrites dans la partie II, sont employées dans des protocoles plus spécialisés, comme par exemple le vote électronique ou encore la monnaie numérique. Nous ne décrirons pas ici ces protocoles, qui pour certains ne sont pas encore à ce jour satisfaisants, et qui sont encore actuellement sujets de recherche. Les techniques que nous décrivons ici peuvent aussi, quoiqu'à la base de protocoles plus complexes, être considérées comme des protocoles rudimentaires. Nous sommes ici à la frontière entre *primitive cryptrographique* et *protocole*. Cette partie doit beaucoup à [**4**] et à [**7**].

Les techniques suivantes nous semblent importantes, notamment pour les applications au vote électronique ou à la monnaie numérique :

(1) La signature en aveugle.
(2) Les preuves à divulgation nulle.
(3) Les mises en gage.
(4) Le chiffrement homomorphique
(5) Le partage de secret
(6) Le mix net

## 13. La signature en aveugle

La signature en aveugle consiste à faire signer un document à une autorité, sans que l'utilisateur qui obtient cette signature puisse être tracé. En particulier :

(1) L'autorité ne peut pas prendre connaissance du document au moment de la signature.
(2) Si le document signé revient à la vue de l'autorité, celle-ci ne pourra pas déterminer dans quelles circonstances elle a signé ce document. En particulier, elle ne pourra pas dire qui lui a demandé de signer ce document.

Parfois on demandera aussi une fonction qui empêche le *rejeu* d'un document signé.

**13.1. Signature en aveugle simple.** — Voici une description fonctionnelle : $x \in D$ est un document, $h$ est une fonction de hachage,

$$h : D \to \{0,1\}^k.$$

On dispose aussi d'une fonction de *camouflage* $g$ de deux variables

$$g : R \times \{0,1\}^k \to D'$$

qui à tout alea $r \in R$ et tout haché $y \in \{0,1\}^k$ fait correspondre un texte à signer $g(r,y)$. Soit par ailleurs $S$ la fonction de signature de l'autorité. On suppose que :

(1) La fonction $g$ est facilement calculable.

(2) Connaissant $g(r, y)$ il est quasi-impossible de déterminer les valeurs de $r$ et de $y$.

(3) Connaissant $r$ et $S(g(r, y))$ il est facile de trouver $S(y)$.

Dans ces conditions, voici comment peut se présenter une signature en aveugle :

(1) L'utilisateur $U$ veut faire signer le document $x$. Il calcule donc $y = h(x)$.

(2) L'utilisateur $U$ tire au sort un alea $r$ puis calcule :

$$z = g(r, y) = g(r, h(x)).$$

(3) L'utilisateur $U$ fait signer $z$ par l'autorité $A$ :

$$s' = S(z) = S\left(g\left(r, h\left(x\right)\right)\right).$$

(4) L'utilisateur $U$ obtient ainsi :

$$s' = S\left(g\left(r, h\left(x\right)\right)\right) = g\left(r, S\left(h\left(x\right)\right)\right).$$

Puisqu'il connait aussi $r$ il peut calculer

$$s = S\left(h\left(x\right)\right).$$

Le couple $(x, s)$ constitue un document muni d'une signature valide de l'autorité.

Même si l'autorité maintient une base de tous les documents qui lui ont été présentés, elle ne pourra jamais retrouver trace du couple $(x, s)$ dans sa base (sauf évidemment si la base ne contient qu'une signature !).

Si on choisit comme fonction de signature, une signature *RSA* de module $n$, d'exposant public $e$ et d'exposant privé $d$. Alors on peut choisir comme fonction $g$ :

$$g(r, y) = r^e\, y.$$

Dans ce cas l'utilisateur tire au sort un entier $r$ premier avec $n$ puis calcule $r^e\, h(x)$. Il obtient de l'autorité :

$$S(g(r, y)) = (r^e\, y)^d \mod n = r\, S(y) \mod n.$$

Connaissant $r$ et $S(g(r, y))$ il est donc facile de calculer $S(y)$.

**13.2. Éviter le rejeu.** — Il existe un amélioration du protocole qui évite le rejeu du document signé. Si ce document signé représente une monnaie numérique par exemple il est très important qu'il ne puisse pas être représenté plusieurs fois. Nous détaillerons pas cette amélioration qu'on peut trouver par exemple dans [**7**, p. 168-170]

## 14. Les preuves à divulgation nulle

Dans les preuves à divulgation nulle, un *prouveur* tente de convaincre un *vérificateur* qu'il connait un certain secret. Mais il doit le faire sans fournir d'information qui pourrait permettre au vérificateur de calculer en pratique (par un algorithme non déterministe polynomial par exemple) des informations qu'il n'aurait pas pu calculer sans les communications du prouveur.

Du point de vue de la terminologie, on parlera de prouveur (resp. de vérificateur) honnête lorsque celui-ci suivra à la lettre le protocole défini. Un *tricheur* est un prouveur ou un vérificateur qui ne suivent pas le protocole imposé. En particulier un prouveur tricheur cherchera à faire croire au vérificateur qu'il connait un secret dont il ne dispose pas.

Voici par exemple l'identification de Fiat-Shamir et l'identification de Schnorr.

**14.1. Identification de Fiat-Shamir.** — Cette primitive est basée sur le problème de la résiduosité quadratique. Soit $n = pq$ un produit de deux nombres premiers distincts grands. Notons $Q_n$ l'ensemble des résidus quadratiques modulo $n$, qui sont premiers avec $n$. Le prouveur $P$ dispose d'un secret $s = (s_1, \cdots, s_k)$ constitués de $k$ éléments $s_i \in \mathbb{Z}/n\mathbb{Z}$ premiers avec $n$. La clé publique est $n$ et $u = (s_1^2, \cdots, s_k^2) = (u_1, \cdots, u_k)$. L'identification du prouveur $P$ auprès du vérificateur $V$ a lieu de la façon suivante :

(1) Le prouveur $P$ tire au hasard $r \in (\mathbb{Z}/n\mathbb{Z})^*$ premier avec $n$. Il calcule $v = r^2 \mod n$ et l'envoie au vérificateur $V$.

(2) Le vérificateur $V$ tire au hasard $e = (e_1, \cdots, e_k) \in \{0,1\}^k$ et l'envoie à $P$.

(3) $P$ calcule $w = r \prod_{i=1}^{k} s_i^{e_i} \mod n$ et l'envoie à $V$.

(4) $V$ calcule $w^2 \mod n$ et accepte la preuve s'il trouve $v \prod_{i=1}^{k} u_i^{e_i}$.

**14.2. Identification de Schnorr.** — On met en place un groupe multiplicatif $\mathbb{Z}/p\mathbb{Z}^*$ et un sous groupe $H$ d'ordre $q$ où $q$ est un grand nombre premier qui divise $p-1$. Pour des détails sur la construction du couple $(q,p)$ on peut voir le site *http ://www.acrypta.fr* ou le livre [**2**]. Soit $\alpha$ un générateur de $H$ (on tire au sort $\beta \in \mathbb{Z}/p\mathbb{Z}^*$, si $\beta^{(p-1)/q} \neq 1$ alors on pose $\alpha = \beta^{(p-1)/q}$, sinon on retire un $\beta$). La clé secrète $a$ du prouveur est tirée au sort dans $\mathbb{Z}/q\mathbb{Z}$. La clé publique est $(p,q,\alpha,u)$ où $u = \alpha^a \mod p$. Le prouveur $P$ va prouver au vérifieur $V$ qu'il connait le logarithme discret de $u$.

(1) $P$ tire au sort $r \in \mathbb{Z}/q\mathbb{Z}$, calcule $x = \alpha^r \mod p$ et envoie le résultat à $V$.
(2) $V$ tire au sort $z \in \mathbb{Z}/q\mathbb{Z}$ et l'envoie à $P$.
(3) $P$ calcule $v = r - az \mod q$ et l'envoie à $V$.
(4) $V$ vérifie que $x = \alpha^v u^z \mod p$.

**14.3. Preuve sans divulgation et signature.** — Une preuve interactive, sans divulgation, comme les deux exemples que nous venons de voir peut se modifier en une signature. Les bits qui lors du protocole de preuve étaient envoyés par le vérificateur, sont maintenant générés par le message à signer.

Reprenons les notations du paragraphe 14.2. On fixe donc $h : \{0,1\}^* \to \mathbb{Z}/q\mathbb{Z}$ une fonction de hachage cryptographique. Si $P$ veut signer un message $m$, il hache la concaténation $m||x$, et cette empreinte va jouer le rôle du $z$ du protocole du paragraphe 14.2 :

$$z = h(m||\alpha^r \mod p).$$

La signature est alors : $(z,v) = (z, r - az \mod q)$.

La vérification de la signature consiste à tester si la condition suivante est satisfaite :

$$z = h(m||\alpha^v u^z \mod p).$$

## 15. Engagement

On est souvent confronté au problème suivant : on veut s'engager sur une valeur (oui ou non par exemple pour un vote), et cacher cette valeur jusqu'à ce que le protocole demande de la démasquer. Là on impose bien sûr que cette valeur n'ait pas pu être modifiée après l'engagement. Nous allons donner deux exemples d'engagement.

**15.1. Engagement basé sur la résiduosité quadratique.** — Ici nous allons engager un bit $b = 0$ ou $b = 1$. Soit $n = pq$ le produit de deux nombres premiers distincts grands. Soit $Q$ le sous-groupe des éléments carrés de $\mathbb{Z}/n\mathbb{Z}$ qui sont premiers avec $n$. Soit $J^+$ le sous-groupe des éléments de symbole de Jacobi 1. Soit enfin $N = J^+ \setminus Q$ les éléments non-carrés de $J^+$. Voici le protocole entre Alice qui veut engager un bit et le receveur Bob.

(1) Alice construit $p, q, n$, tire au sort $v \in N$.

(2) Alice engage le bit $b$. Pour cela elle tire au hasard $r \in \mathbb{Z}/n\mathbb{Z}^*$ et pose

$$c = r^2\, v^b \mod n,$$

et envoie $n, c$ et $v$ à Bob.

(3) Pour révéler l'engagement, Alice envoie $p, q, r, b$ à Bob.

(4) Bob peut vérifier que : $p$ et $q$ sont premiers, $n = pq$, $r \in \mathbb{Z}/n\mathbb{Z}^*$, $v \notin Q$, $c = r^2 v^b \mod n$.

On peut remarquer que les contraintes qu'on voudrait sur le système sont bien respectées :

(1) Le destinataire Bob, tant qu'Alice ne révèle rien, ne peut pas déterminer si $b = 0$ ou si $b = 1$. En effet, sinon il pourrait déterminer si la valeur aléatoire $c$ est un carré ou non contredisant ainsi la difficulté du problème de la résiduosité quadratique.

(2) L'expéditrice Alice, ne peut pas modifier la valeur engagée. En effet elle a déjà fourni $n$, $c$, donc ne peut plus tricher sur la parité de $b$ qui correspond au fait que $b$ soit un carré ou pas modulo $n$.

**15.2. Engagement basé sur le logarithme discret.** — Ici, on va engager un message $x \in \{0, \cdots, q-1\}$. Pour cela on utilise un environnement lié au logarithme discret, c'est-à-dire un groupe $\mathbb{Z}/p\mathbb{Z}^*$ où $p$ est un grand nombre premier, muni d'un sous-groupe $H$ d'ordre $q$ où $q$ est un nombre premier qui divise $p-1$. Le protocole se déroule de la manière suivante :

(1) Bob construit $p, q, \alpha, v$ où $\alpha$ et $v$ sont des générateurs du sous-groupe $H$ de $\mathbb{Z}/p\mathbb{Z}^*$ (dans ce cas particulier, des éléments de $H$ distincts de 1). Il envoie $p, q, \alpha, v$ à Alice.

(2) Alice vérifie que $p$ et $q$ sont premiers, que $q$ divise $p-1$, que $\alpha$ et $v$ sont d'ordre $q$. Pour engager le message $x \in \{0, \cdots, q-1\}$, elle tire $r \in \{0, \cdots, q-1\}$ au sort et calcule :

$$c = \alpha^r \, v^m \mod p.$$

Elle envoie $c$ à Bob.

(3) Pour révéler $m$, Alice envoie $r$ et $m$ à Bob qui vérifie que $c = \alpha^r \, v^m \mod p$.

Là encore le lecteur pourra s'assurer que les contrainte voulues sont bien satisfaites.

## 16. Le chiffrement homomorphique

Soit $\mathcal{E}$ un chiffrement probabiliste. Soit $\mathcal{M}$ l'espace des textes clairs et $\mathcal{C}$ l'espace des chiffrés. On suppose que $\mathcal{M}$ et $\mathcal{C}$ sont des groupes pour des opérations respectives $\oplus$ et $\otimes$. Nous dirons que le chiffrement $\mathcal{E}$ est homomorphique si étant donnés deux chiffrés $c_1 = \mathcal{E}(m_1, r_1)$ et $c_2 = \mathcal{E}(m_2, r_2)$ obtenus avec des aléas $r_1$ et $r_2$, il existe un aléa $r$ tel que :

$$c_1 \otimes c_2 = \mathcal{E}(m_1 \oplus m_2, r).$$

Dans un tel schéma on peut déchiffrer un $\otimes$ de plusieurs chiffrés, sans déchiffrer chaque chiffré individuellement. Ainsi on évite de tracer qui a chiffré quoi.

Partons du chiffrement d'Elgamal. On dispose donc de $p, q, \alpha$, avec comme toujours $p$ et $q$ premiers, $q$ divise $p-1$ et $\alpha$ un générateur du sous-groupe $G_q$ d'ordre $q$. Soit $a$ la clé privée et $\beta = \alpha^a$ la clé publique.

On définira le chiffrement de $m$ comme étant le chiffrement d'Elgamal de $\alpha^m$.

Un chiffrement de $m_1$ est un couple $(\alpha^{r_1}, \alpha^{m_1}\beta^{r_1})$. Un chiffrement de $m_2$ est un couple $(\alpha^{r_2}, \alpha^{m_2}\beta^{r_2})$.

Le couple $(\alpha^{r_1+r_2}, \alpha^{m_1+m_2}\beta^{r_1+r_2}$ est un chiffrement de $m_1+m_2 \mod q$.

En prenant donc pour loi $\oplus$ la loi $+$ sur les messages et pour loi $\otimes$ le produit composante par composante des chiffrés, on a un chiffrement homomorphique.

Le problème est q'une fois le message $\alpha^m$ (qui joue le rôle du texte clair pour le chiffrement d'Elgamal), il faudrait savoir récupérer le vrai message clair $m$. C'est un problème de log discret, mais c'est faisable si $m$ est petit, par recherche exhaustive. En particulier si $m = 0$ ou 1, c'est très simple.

## 17. Le partage de secret

Le partage de secret, est une technique importante en cryptographie, qui permet de partager secret entre $n$ participants et de ne permettre la reconstitution de ce secret que si $t$ participants au moins se mettent d'accord pour reconstituer le secret. C'est le cas par exemple d'une clé privée partagée entre $n$ autorités, et où déchiffrer un message chiffrée avec la clé publique correspondante demande la participation d'au moins $t$ des autorités.

**17.1. Partage à seuil de Shamir.** — Ce système de partage est basé sur l'interpolation de Lagrange. On va supposer l'existence d'un centre de confiance $C$ qui met en place le système et distribue les $n$ morceaux du partage aux $n$ autorités. Voici comment se passe le partage du secret $s$.

On suppose donc qu'on a $n$ autorités $A_i$ à qui il faut partager un secret $s$, et que la coopération d'au moin$t$ d'entre elles permettent de reconstituer ce secret. Par ailleurs on dispose du centre de confiance $C$ qui va faire le partage.

(1) Le centre $C$ choisit un nombre premier $p > \max(n, s)$ et pose $a_0 = s$.

(2) $C$ tire au hasard et indépendamment des nombres $a_1, \cdots, a_{t-1}$ appartenant à $\{0, 1 \cdots, p-1\}$, obtenant ainsi un polynôme

$$P(X) = a_0 + a_1 X + \cdots + a_{t-1} X^{t-1}.$$

(3) $C$ tire au sort $n$ points distincts $x_i \in \{1, \cdots, p-1\}$ et calcule le secret partiel $s_i = P(x_i)$. Il communique secrètement $(x_i, s_i)$ à l'autorité $A_i$.

Toute collaboration d'au moins $t$ autorités permet de reconstituer par interpolation de Lagrange le polynôme $P(X)$, puis de calculer $s = P(0)$. Si au plus $t-1$ secrets partiels sont connus, alors tout élément de $\{0, 1, \cdots, p-1\}$ a une égale probabilité d'être le secret $s$.

**17.2. Variante éliminant le centre de confiance - Protocole de Pedersen.** — Le protocole précédent a besoin d'un centre de confiance, on peut s'en passer de la manière suivante. Tout d'abord on se place dans un environnement classique pour le logarithme discret, c'est-à-dire qu'on construit $(p, q, \alpha)$ où $p$ et $q$ sont des nombres premiers, $q$ divise $p-1$ et $\alpha$ est un générateur du sous-groupe $G_q$ d'ordre $q$ de $\mathbb{Z}/p\mathbb{Z}^*$. L'établissement de ces données est fait une fois pour toute et les autorités se mettent d'accord sur ces valeurs. Nous aurons aussi besoin d'un engagement, nous prendrons l'engagement décrit au 15.2, en utilisant les données sur lesquelles toutes les autorités se sont accordées complétées par un autre générateur $v$ de $G_q$. Nous noterons $C(x, r)$ l'engagement du message $x$ utilisant le tirage au sort $r$.

(1) Chaque autorité $A_i$ tire au sort $x_i \in G_q$ et calcule $h_i = \alpha^{x_i}$. $A_i$ tire aussi au sort un $r_i$ et calcule :

$$c_i = C(h_i, r_i) = \alpha^{r_i} v^{h_i}.$$

Cette valeur est transmise à toutes les autorités.

(2) Chaque $P_i$ ouvre $c_i$.

(3) La clé publique est :

$$h = \prod_{i=1}^{n} h_i.$$

La clé secrète est $x = \sum_{i=1}^{n} x_i$ et n'est connue d'aucun participant à moins de se mettre tous ensemble.

Le tout est maintenant de savoir comment faire en sorte de pouvoir récupérer tous les $x_i$ dont on a besoin pour déterminer le secret $x$, alors que seulement $t$ autorités sont disponibles pour reconstituer le secret.

L'idée du protocole qu'on peut trouver en détail dans [**5**] est que chacun des $x_i$ peut être partagée par l'authorité $A_i$, qui joue alors pour le secret $x_i$ le rôle de centre de confiance, entre toutes les autres autorités, grâce au protocole de base de Shamir décrit dans le paragraphe 17.1. Dans ce cas, si $t$ autorités collaborent, elles peuvent récupérer tous les secrets $x_i$ et reconstituer $x$.

## 18. Le mix net

Le mix net, est une méthode concurrente du chiffrement homomorphique, et qui permet de ne pas pouvoir tracer qui a chiffré quoi.

Cette technique repose sur un chiffrement aléatoire qui permette de calculer sans repasser par le texte clair, une nouvelle forme aléatoire du chiffré du même texte clair (*re-randomization*). Le chiffrement d'Elgamal permet cette opération.

Plaçons nous dans un environnement de chiffrement d'Elgamal $(p, q, \alpha)$ avec une clé privée $a$ et une clé publique $h = \alpha^a$. Rappelons que $p$ et $q$ sont deux nombres premiers tels que $q$ divise $p-1$ et $\alpha$ un générateur du groupe $G_q$ d'ordre $q$.

Le chiffré d'un message $m$ est un couple $(u, v)$ où, $r$ étant un élément tiré au sort,

$$u = \alpha^r,$$

$$v = m\,\beta^r.$$

On constate dans ces conditions que $(u', v')$ où :

$$u' = u\,\alpha^t,$$

$$v' = v\,\beta^t,$$

est un autre chiffré aléatoire du même texte clair, avec la même clé publique. De ce fait, on ne reconnait plus le chiffré initial, et l'intervenant

intermédiaire a pu réaliser ce nouvelle forme aléatoire sans connaitre le texte clair.

# PARTIE IV
# TENDANCES ACTUELLES

## 19. Les considérations de la NSA

La NSA dans un article intitulé "NSA Suite B cryptography " (Fact Sheet Suite B Cryptography, NSA, 07/2008) définit ses recommandations concernant les primitives cryptographiques de base. Ceci inclut le chiffrement (qui rappelons le, pour un flux de données ne peut se faire qu'avec du chiffrement à clé secrète), la signature numérique, l'échange de clés, les fonctions de hachage. La caractéristique la plus visible est l'abandon de la cryptographie RSA au profit de la cryptographie elliptique sur un corps fini premier. Ainsi seules les courbes elliptiques sur $\mathbb{Z}/p\mathbb{Z}$ sont conseillées, celles sur les corps finis ayant $2^n$ éléments ne sont pas citées (bien entendu le cas $q = p^n$ avec $p$ premier quelconque et $n > 1$ n'était déjà plus envisagé depuis un certain temps).

Si on se reporte au niveau de sécurité conseillé pour le chiffrement à clé secrète on voit que :

1) AES avec 128 bits de clé est conseillé pour des applications de niveau de securité élevé ;

2) AES avec 192 bits de clé est conseillé pour une securité "top level secret". Mais s'il est bien spécifié que 192 bits sont suffisant pour cette sécurité très élevée, pour des raisons de compatibilité c'est en définitive AES 256 bits de clés qui sera utilisé.

De ce fait les autres circuits (courbes elliptiques, hachage) doivent suivre cette sécurité, c'est-à dire une courbe elliptique de 256 bits et SHA256 pour aller avec AES128, une courbe de 384 bits et SHA384 pour aller avec AES192 (remplacé en fait par AES256). A vrai dire, si on voulait bénéficier de la sécurité complète de AES256 il faudrait prendre

une courbe elliptique de 512 bits et SHA512, mais ceci n'est pas le conseil de la NSA.

L'article "Les standards en cryptographie sont ils souhatables " de Arjen Lenstra à ce sujet est très intéressant et relativise un peu le rôle de la cryptographie dans le processus global de sécurité des systèmes.

## 20. Le chiffrement à clé secrète

Le chiffrement à clé secrète est indispensable pour des raisons de rapidité. Les chiffrements à flot ne sont pas conseillés pour des raisons de sécurité. Dans l'état actuel de la situation, AES semble un choix incontournable. C'est un système très rapide et qui assure une très bonne sécurité. Ceci étant dit, le choix du mode de fonctionnement se pose. C'est-à-dire, que faire lorsqu'on a un long message à chiffrer, un gros fichier, un gros flux de données ? Plusieurs choix plus ou moins judicieux sont possibles. Nous allons développer ici le Galois Counter Mode de D. McGrew et J. Viega, qui combine un mode très rapide et très sûr de chiffrement (le mode Counter) avec un contrôle d'intégrité également très rapide et très sûr.

**20.1. Idée générale.** — Bien que le mode GMC (Galois Counter Mode) accepte plusieurs tailles pour certains des paramètre qui interviennent, nous donnons ici les paramètres les plus conseillés (quand un choix est possible nous mettrons entre parenthèse qu'il s'agit d'une valeur conseillée).

(1) **Ce qu'on doit traiter.** On suppose que les données comportent

(a) un texte clair $P$ à chiffrer et dont on doit assurer l'intégrité dont le nombre $t$ de bits est tel que $0 \leq t \leq 2^{39} - 2^8$.

(b) des données additionnelles $A$ à transmettre sans chiffrement, mais dont on doit assurer l'intégrité, dont le nombre $a$ de bits vérifie $0 \leq a \leq 2^64$.

(2) **Ce dont on dispose.** On dispose d'un circuit de chiffrement par blocs (par exemple AES 128 bits) dont une clé de session $K$ a été choisie. La taille des blocs chiffrés est 128 bits

(3) Ce qu'on veut récupérer. On veut récupérer en sortie

(a) le chiffré $C$ de $P$ ayant la même taille que $P$
(b) les données additionnelles $A$
(c) un tag $T$ qui assure l'intégrité de chaine $C||A$.

(4) **Comment faire.**

(a) **Le chiffrement** On dispose d'une zone mémoire $Y$ de 128 bits (taille du block à chiffrer) qu'on va faire évoluer à partir d'une valeur $Y_0$ en $Y_1, Y_2, \ldots$. Pour chaque nouveau message à chiffrer avec une même clé $K$ on choisit une nouvelle valeur initiale (non encore utilisée avec cette clé) $IV$ de 96 bits (taille conseillée). Par ailleurs on a une valeur initiale $Z_0 =$ de 32 bits

$$Y_0 = 00 \ldots 01.$$

On construit $Y_0$ par

$$Y_0 = IV||Z_0.$$

Par récurrence on construit $Y_i$ en prenant

$$Y_i = IV||Z_i,$$

où

$$Z_i = Z_{i-1} + 1 \mod 2^{32}.$$

On notera $\mathrm{inc}(Y_{i-1})$ l'application qui fait passer de $Y_{i-1}$ à $Y_i$. Le texte clair $P$ étant découpé en $n-1$ blocs $P_i$ de taille 128 bits et en un dernier bloc $P_n^*$ de taille $0 < b \leq 128$ bits, on calcule les blocks de chiffré par :

$$C_i = P_i \oplus E(K, Y_i) \quad i = 1, \ldots, n-1,$$
$$C_n^* = P_n^* \oplus MSB_b\left(\mathcal{E}(K, Y_n)\right),$$

où $\mathcal{E}(K, *)$ est la fonction de chiffrement d'un bloc avec la clé secrète $K$.

(b) **l'intégrité** On calcule tout d'abord le chiffré du bloc null avec la clé $K$

$$H = \mathcal{E}(K, 0^{128}).$$

Le tag $T$ qui permet la vérification de l'intégrité (que nous prendrons de taille 128 bits, valeur conseillée) est calculé par

$$T = \mathrm{GHASH}(H, A, C) \oplus \mathcal{E}(K, Y_0),$$

où GHASH est une fonction que nous allons définir par la suite.

Remarquons que ce système avec une entrée $P$ de taille 0 permet de ne faire que de l'intégrité sur $A$.

**20.2. Le corps fini à $2^{128}$ éléments.** — On prend comme polynôme minimal, le polynôme

$$P(X) = X^{128} + X^7 + X^2 + X + 1.$$

Ce polynôme est irreductible sur $\mathbb{F}_2$. Le corps à $2^{128}$ éléments peut être représenté par

$$\mathbb{F}_{2^{128}} = \mathbb{F}_2[X]/\left(P(X)\right).$$

Autrement dit les éléments sont les polynômes à coefficients dans $\mathbb{F}_2$ de degré $\leq 127$, l'addition est l'addition des polynômes et la multiplication est la multiplication des polynômes modulo $P(X)$. On notera $\alpha$ la classe du polynôme $X$ dans ce passage au quotient. Tout élément $a \in \mathbb{F}_{2^{128}}$ s'écrit donc de manière unique sous la forme

$$a = \sum_{i=0}^{127} a_i \alpha^i.$$

En particulier l'élément $\alpha^{128}$ s'écrit

$$\alpha^{128} = 1 + \alpha + \alpha^2 + \alpha^7.$$

On notera $r = 1 + \alpha + \alpha^2 + \alpha^7$. Pour construire un algorithme efficace de multiplication nous allons commencer par la multilication d'un élément $a$ par $\alpha$ qu'on notera $m(a)$.

**procédure m(a)** {
**si** $a_{127} = 0$
**renvoyer** shr(a, 1) ;
**sinon**
$b$ = shr(a, 1) ;
**renvoyer** $b \oplus r$ ;
**finsi**
}

Maintenant nous allons effectuer la multiplication $c = b.a$ où

$$a = \sum_{i=0}^{127} a_i \alpha^i, \quad b = \sum_{i=0}^{127} b_i \alpha^i.$$

Pour cela nous allons reconstituer $b$ par l'algorithme de Hörner

**procédure mult**$(\mathbf{b}, \mathbf{a})$ {
$c = 0$ ;
**faire depuis** $i = 0$ **jusqu'à** 127
$c = m(c)$ ;
**si** $b_{127-i} = 1$
$c = c \oplus a$ ;
**finsi**
**finfaire**
}

**20.3. La fonction GHASH.** — Pour terminer la description complète du système Galois Counter Mode, il faut maintenant décrire la façon de calculer le tag $T$

$$T = \text{GHASH}(H, A, C) \oplus \mathcal{E}(K, Y_0,$$

c'est-à-dire décrire la fonction GHASH$(H, A, C)$. Pour cela on rappelle que

(1) $n$ est le nombre de blocs de texte chiffré

$$C = C_1 C_2 \dots C_{n-1} C_n^*,$$

où les blocs $P_i$ sont de taille 128 bits, le dernier bloc $P_n^*$ étant de taille $0 < t \leq 128$.

(2) $m$ est le nombre de blocs de données additionnelles

$$A = A_1 A_2 \dots A_{m-1} A_m^*,$$

où les blocs $A_i$ sont de taille 128 bits, le dernier bloc $A_m^*$ étant de taille $0 < t \leq 128$.

On définit alors par récurrence la suite finie $(X_i)_{i=0,\ldots,m+n+1}$ par

$$X_i = \begin{cases} 0 & \text{si } i = 0 \\ (X_{i-1} \oplus A_i) \,.\, H & \text{si } i = 1, \ldots, m-1 \\ (X_{m-1} \oplus (A_m^* || 0 \ldots 0)) \,.\, H & \text{si } i = m \\ (X_{i-1} \oplus C_i) \,.\, H & \text{si } i = m+1, \ldots, m+n-1 \\ \left(X_{m+n-1} \oplus (C_m^* || 0 \ldots 0)\right) \,.\, H & \text{si } i = m+n \\ \left(X_{m+n} \oplus (\text{taille\_bit}(A) || \text{taille\_bit}(C))\right) \,.\, H & \text{si } i = m+n+1 \end{cases}$$

où la multiplication par $H$ se fait dans le corps fini à 128 éléments. On pose alors

$$\text{GHASH}(H, A, C) = X_{m+n+1}.$$

On peut obtenir une implémentation efficace de ce calcul qui aille plus vite qu'AES lui-même, on pourra se référer pour cela à l'article original de D. McGrew et J. Viega intitulé *The Galois/Counter Mode of Operation (GCM).*

## 21. La cryptographie elliptique

En ce qui concerne la cryptographie à clé publique, la tendance actuelle est de remplacer les primitives RSA ou le calcul des logarithmes discrets dans $\mathbb{Z}/p\mathbb{Z}$ par des primitives basées sur le logarithme discret (ou problèmes proches) sur le groupe des points d'une courbe elliptique sur un corps premier $\mathbb{Z}/p\mathbb{Z}$. En effet RSA, le logarithme discret sur $\mathbb{Z}/p\mathbb{Z}$, ne peuvent plus suivre la sécurité des circuits à clé secrète en raison de la trop grande taille des clés publiques et privées qu'il faudrait alors déployer. Ainsi la signature sera ECDSA, l'échange de clé pourra se faire avec ECDH. De nombreux problèmes se posent alors. Nous n'en parlerons pas plus dans ce premier cours.

# PARTIE V
# ANNEXES

## 22. Techniques d'enrobage

**22.1. Présentation.** — Les diverses données qu'on est amené à manipuler sont en général des suites d'octets : clés publiques, clés privées clés secrètes etc. Ces suites d'octets, afin de pouvoir être communiquées, sont parfois transformées en caractères imprimables et entourées d'un en-tête et d'une fin.

**22.2. L'encodage Base64.** — Afin de transformer une suite d'octets en une suite de caractères imprimables et représentables en ASCII non étendu, on transforme 3 octets (24 bits) en 4 tranches de 6 bits. Une tranche de 6 bits représente un nombre $Val$ vérifiant :

$$0 \leq Val < 64.$$

À chaque valeur $Val$ on fait correspondre un symbole $Symb$ suivant la table 1.

Comme le texte d'entrée n'a pas forcément un multiple de 3 octets, la fin du texte de sortie doit être précisée.

(1) Si le texte d'entrée a $3k$ octets, tout se passe bien on les regroupe en exactement $4k$ blocs de 6 bits qui sont traduits en $4k$ symboles suivant la table 1.

(2) Si le texte d'entrée a $3k + 1$ octets, alors on rajoute 4 bits nuls à la suite des 2 bits restés seuls après regroupement par paquets de 6 bits. Les 6 bits ainsi obtenus sont transformés en 1 symbole suivant la table 1, symbole qui bien sûr est mis dans le texte de sortie, et de plus, on rajoute à la fin du texte de sortie 2 signes =.

(3) Si le texte d'entrée a $3k + 2$ octets, alors on rajoute 2 bits nuls à la suite des 4 bits restés seuls après regroupement par paquets de 6 bits. Les 6 bits ainsi obtenus sont transformés en 1 symbole suivant la table 1, symbole qui bien sûr est mis dans le texte de sortie, et de plus, on rajoute à la fin du texte de sortie 1 signe =.

| Val | Symb | val | Symb | Val | Symb | Val | Symb |
|---|---|---|---|---|---|---|---|
| 0 | A | 16 | Q | 32 | g | 48 | w |
| 1 | B | 17 | R | 33 | h | 49 | x |
| 2 | C | 18 | S | 34 | i | 50 | y |
| 3 | D | 19 | T | 35 | j | 51 | z |
| 4 | E | 20 | U | 36 | k | 52 | 0 |
| 5 | F | 21 | V | 37 | l | 53 | 1 |
| 6 | G | 22 | W | 38 | m | 54 | 2 |
| 7 | H | 23 | X | 39 | n | 55 | 3 |
| 8 | I | 24 | Y | 40 | o | 56 | 4 |
| 9 | J | 25 | Z | 41 | p | 57 | 5 |
| 10 | K | 26 | a | 42 | q | 58 | 6 |
| 11 | L | 27 | b | 43 | r | 59 | 7 |
| 12 | M | 28 | c | 44 | s | 60 | 8 |
| 13 | N | 29 | d | 45 | t | 61 | 9 |
| 14 | O | 30 | e | 46 | u | 62 | + |
| 15 | P | 31 | f | 47 | v | 63 | / |

TABLE 1. Table de conversion

**Remarque :** le nombre de symboles du texte de sortie, y compris les éventuels signes =, est un multiple de 4.

**22.3. Le contrôle CRC-24.** — Nous expliquons ici comment on calcule à partir de données binaires (par exemple le fichier binaire d'une clé publique) un code de contrôle (ici le CRC-24). Les données pour lesquelles on veut calculer un code de contrôle CRC (Cyclic Redondancy Code) sont considérées comme une suite d'octets :

$$V_0, V_1, \cdots, V_{s-1}.$$

L'octet $V_i$ est lui même constitué de 8 bits suivant le format :

$$V_i = b_7^{(i)} b_6^{(i)} b_5^{(i)} b_4^{(i)} b_3^{(i)} b_2^{(i)} b_1^{(i)} b_0^{(i)}.$$

La concaténation :

$$M = V_0 || V_1 || \cdots || V_{s-1}$$

des octets de cette suite nous fournit une suite de $n = 8s$ bits :

$$\begin{aligned} M &= b_7^{(0)} b_6^{(0)} b_5^{(0)} b_4^{(0)} b_3^{(0)} b_2^{(0)} b_1^{(0)} b_0^{(0)} \cdots \\ &\quad \cdots b_7^{(i)} b_6^{(i)} b_5^{(i)} b_4^{(i)} b_3^{(i)} b_2^{(i)} b_1^{(i)} b_0^{(i)} \cdots \\ &\quad \cdots b_7^{(s-1)} b_6^{(s-1)} b_5^{(s-1)} b_4^{(s-1)} b_3^{(s-1)} b_2^{(s-1)} b_1^{(s-1)} b_0^{(s-1)}. \end{aligned}$$

Afin de simplifier un peu les notations, notons :

$$M = m_{n-1} m_{n-2} \cdots m_0$$

cette suite de bits. Ainsi :

$$m_{n-1} = b_7^{(0)},$$

$$m_{n-2} = b_6^{(0)},$$

et plus généralement :

$$m_i = b_{i \mod 8}^{\left(s - \lceil \frac{i}{8} \rceil\right)},$$

où $\lceil \frac{i}{8} \rceil$ désigne le plus petit entier $> i/8$ et $i \mod 8$ le reste de la division de $i$ par 8.

On considère alors que $M$ (qui est une suite de $n$ bits) peut en fait représenter le polynôme $M(x)$ de degré $n-1$ à coefficient dans le corps à deux éléments $\{0, 1\}$, dont les coefficients sont les bits $m_i$ :

$$M(x) = m_{n-1} x^{n-1} + m_{n-2} x^{n-2} + \cdots + m_i x^i + \cdots + m_1 x + m_0.$$

On fixe deux polynômes, le polynôme générateur :

$$G(x) = x^{24} + x^{23} + x^{18} + x^{17} + x^{14} + x^{11} + x^{10} + x^7 + x^6 + x^5 + x^4 + x^3 + x + 1,$$

et le polynôme initial : :

$$I(X) = x^{23} + x^{21} + x^{20} + x^{18} + x^{17} + x^{16} + x^{10} + x^7 + x^6 + x^3 + x^2 + x.$$

Si on écrit les coefficients sous forme de suites de bits on obtient :

$$G = 1100001100100110011111011,$$

$$I = 10110111000001001100111 0,$$

ou encore en hexadécimal :

$$G = 0x1864cfb,$$

$$I = 0xb704ce.$$

Le calcul du crc de la donnée $M$ se fait alors de la façon suivant :

(1) on calcule :

$$A(x) = x^{24}\, M(x) + x^n\, I(x),$$

(2) on fait la division de $A(x)$ par $G(x)$, le reste est un polynôme $R(x)$ de degré au plus 23. Les 24 bits correspondants aux coefficients de ce polynôme constituent le *crc*. On a donc 3 octets de crc.

**Remarque :** Les deux polynômes $x^{24}\, M(x)$ et $x^n\, I(x)$ ont le même degré. La somme des deux polynômes correspond à un ou exclusif bit à bit sur les deux suites binaires représentant leurs coefficients.

Voici un programme qui calcule le CRC-24. Il utilise la procédure *crc_octets* qui est exactement celle donnée dans la note *RFC 2440*.

```
#include <stdlib.h>
#include <math.h>


/*************************************************/

#define CRC24_INIT 0xb704ceL
#define CRC24_POLY 0x1864cfbL
typedef long crc24;

crc24 crc_octets(unsigned char *octets, size_t len)
{
  crc24 crc=CRC24_INIT;
  int i;
  while (len--)
    {
      crc ^= (*octets++) <<16;
      for (i=0;i<8;i++)
        {
          crc <<= 1;
          if (crc & 0x1000000)
            crc^= CRC24_POLY;
        }
    }
```

```
  return crc & 0xffffffL;
}

void main()
{
  char *suite;
  size_t len=2;
  suite=(char *) malloc(len*sizeof(char));
  suite[0]=(char) 0xA5;
  suite[1]=(char) 0xca;
  crc24 crc=crc_octets((char *) suite,len);
  printf("%lx\n",crc);
}
```

**22.4. Synthèse : la transformation Radix-64.** — La transformation Radix-64 consiste en deux opérations. D'une part le calcul du crc. D'autre part l'utilisation de base64. Voici un exemple de clé publique (révoquée) sous forme d'armure ascii :

```
 -----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1.4.5 (GNU/Linux)

mIsEQ6TTOAEEANPVWUZar2lwmus7ziNqpX4CKcXFDqM2v+34O6RlIdgeiJdp8jgU
yTXssymZnaCidgWVw7OZ+G7b+/EBhSD7J1wFXaYdbY9M44PQr5S+45oCMiylanVA
1VH4KPy4LKzaZfx4HCBHS2YuRxXwkEmQ7jmiYu6NbOZ9OGGHwm9IlfdbAAYpiLIE
IAECABwFAkOk068VHQBjb3BpZSBkZSBzYXV2ZWdhcmRlAAoJEAHVEZOOCNHdgwME
AMCblyI4g1ogsHGrm3xQtflqDOUqSQtVmy1auWUl2uWdE7KNMOiofTCK+CjpcJPn
k3meQWSbpX6LCqFaviPV9dw9umjqIOqyLZ1YPuufGGdFDjtVSaq4k66VqZEdfc6B
xRUj7ya7xaO96Txs+lsbleRiGadCDepOxrL2Q+mNZ6C+tD9Sb2JlcnQgUm9sbGFu
ZCAoYWRyZXNzZSBwZXJzb25uZWxsZSkgPFJvYmVydC5Sb2xsYW5kMTNAZnJlZS5m
cj6IswQTAQIAHQUCQ6TTOAYLCQgHAwIEFQIIAwQWAgMBAh4BAheAAAoJEAHVEZOO
CNHdqe4D/RMx6QLQ9oL3+eO+efPNpUR2OuZKVUMBO7uvHRtLhUzvHR3OPIRwkMz1
dx1hZx2SZVk7ZzIEjR+ujOzyt5hheEzv5J5ShY/hZA+xJkNB8rIzqw6XctAff6RF
A5AomlNodAd5KTfUUBuQd8Guq8y1cKqTZiT7z1rreO4BszvIdMcp
=TJ6m
-----END PGP PUBLIC KEY BLOCK-----
```

On voit donc l'en-tête :

```
 -----BEGIN PGP PUBLIC KEY BLOCK-----
```

```
Version: GnuPG v1.4.5 (GNU/Linux)
```

Puis la clé sous forme base64 (c'est ce qu'on obtient en transformant le fichier binaire de la clé publique par base64) :

```
mIsEQ6TTOAEEANPVWUZar2lwmus7ziNqpX4CKcXFDqM2v+34O6RlIdgeiJdp8jgU
yTXssymZnaCidgWVw7OZ+G7b+/EBhSD7J1wFXaYdbY9M44PQr5S+45oCMiylanVA
1VH4KPy4LKzaZfx4HCBHS2YuRxXwkEmQ7jmiYu6NbOZ9OGGHwm9IlfdbAAYpiLIE
IAECABwFAkOk068VHQBjb3BpZSBkZSBzYXV2ZWdhcmRlAAoJEAHVEZOOCNHdgwME
AMCblyI4g1ogsHGrm3xQtflqDOUqSQtVmy1auWUl2uWdE7KNMOiofTCK+CjpcJPn
k3meQWSbpX6LCqFaviPV9dw9umjqIOqyLZ1YPuufGGdFDjtVSaq4k66VqZEdfc6B
xRUj7ya7xaO96Txs+lsbleRiGadCDepOxrL2Q+mNZ6C+tD9Sb2JlcnQgUm9sbGFu
ZCAoYWRyZXNzZSBwZXJzb25uZWxsZSkgPFJvYmVydC5Sb2xsYW5kMTNAZnJlZS5m
cj6IswQTAQIAHQUCQ6TTOAYLCQgHAwIEFQIIAwQWAgMBAh4BAheAAAoJEAHVEZOO
CNHdqe4D/RMx6QLQ9oL3+e0+efPNpUR2OuZKVUMBO7uvHRtLhUzvHR3OPIRwkMz1
dx1hZx2SZVk7ZzIEjR+ujOzyt5hheEzv5J5ShY/hZA+xJkNB8rIzqw6XctAff6RF
A5AomlNodAd5KTfUUBuQd8Guq8y1cKqTZiT7z1rre04BszvIdMcp
```

On rajoute un nouvelle ligne commençant par le signe ′ =′ suivi des 4 caractères formant le base64 du CRC-24 du fichier binaire de la clé publique :

```
=TJ6m
```

Et enfin la fin :

```
-----END PGP PUBLIC KEY BLOCK-----
```

## 23. Résistance et taille des systèmes

Nous avons vu que dans un protocole, diverses techniques cryptographiques, et donc diverses primitives cryptographiques sont employées simultanément. Comme dans tout assemblage il ne doit pas y avoir de maillon faible. Il faut donc employer des primitives dont les *résistances* sont sensiblement les mêmes. Nous sommes donc amenés à comparer, du point de vue de la résistance aux attaques, des circuits aux fonctionnalités diverses, et dont les technologies sont dissemblables. Pour cela on considère pour chaque type de primitive cryptographique les temps d'exécution des meilleures attaques connues. Ainsi pour des circuits de chiffrement par bloc bien construits, les meilleures attaques connus ne font guère mieux que l'attaque brutale, c'est-à-dire l'essai de toutes les

clés possibles. Par exemple si un circuit de chiffrement par bloc a une clé de 128 bits, l'attaque brutale coûte $2^{128}$ essais, et on dira que sa résistance est de 128. L'attaque sur les fonctions de hachage la plus puissante est l'attaque dite des anniversaires. Si la taille du haché a 256 bits l'attaque des anniversaires coûte en moyenne $2^{128}$ essais. On dira qu'un tel circuit a aussi pour résistance 128. En ce qui concerne les primitives de cryptographie à clé publique, basés sur des problèmes arithmétiques, on se réfère aussi aux meilleures attaques connues pour jauger leur degré de resistance. Bien entendu, tout progrès notable sur la complexité des algorithmes de factorisation d'entiers, de recherche du logarithme discret sur tel ou tel groupe, aurait des répercussions sur les tables de correspondance des résistances des diverses primitives.

Dans la table suivante, on suppose que le composant particulier qu'on étudie n'est pas « cassé », c'est-à-dire qu'on ne peut pas porter une attaque particulière sur lui, notablement plus efficace que la meilleure attaque générale connue. En ce sens cassé ne veut donc pas dire qu'on peut en pratique surmonter sa protection, mais qu'il n'a pas (ou plus) en terme de résistance, la globalité de la résistance qu'on pourrait attendre d'un circuit de ce type. Par exemple *SHA1* dont les hachés sont des blocs de 160 bits, devrait avoir une résistance de l'ordre de 80. Or il existe une attaque particulière qui réussit en $2^{69}$ essais. Ce n'est plus la résistance attendu, même si $2^{69}$ essais reste à peu près hors de portée pour des applications commerciales.

| Système | Meilleure attaque | taille 1 | taille 2 | taille 3 | taille 4 |
|---|---|---|---|---|---|
| block | brutale | 80 bits | 112 bits | 128 bits | 256 bits |
| RSA | factorisation | 1024 bits | 2048 bits | 4096 bits | trop (15000 ?) |
| hachage | anniversaire | 160 bits | 224 bits | 256 bits | 512 bits |
| EC | EC log discret | 160 bits | 200 bits | 256 bits | 400 bits |

## 24. Les organismes de standardisation

Les questions de normalisation et de standardisation sont centrales dans tout problème d'informatique, d'informatique communicante et donc de cryptographie. Rappelons que les standards se définissent *de facto* et que les normes sont définies *de jure* par un organisme normalisateur comme :

– l' ISO ( International Standards Organization) ;
– l' ITU ( International Telecommunication Union) ;
– l' ANSI ( American National Standards Institute) ;
– l' ECMA ( European Computer Manufacturers Association) ;
– l' IEEE ( Institute of Electrical and Electronics Ingineers) ;
– le NIST ( National Institute of Standards and Technology) ;
– ou l' AFNOR ( Association Française pour la NORmalisation).

Dans le cas de l'internet, les standards sont appelés des RFC ( Request For Comments). Il peut s'agir de définitions de protocoles, de projets, de comptes-rendus de réunions, de spécifications de standards, etc. Les RFC sont des documents publics, accessibles par exemple sur `http://www.rfc-editor.org`. Il en existe deux sous-catégories notables, les STD et les FYI : les STD sont les standards officiels et les FYI sont les documents d'apprentissage (For Your Information). Les « drafts »sont les documents des groupes de travail, généralement mis à la disposition de tous et sur lesquels chacun peut émettre des remarques et suggestions.

Dans le cas du web, un organisme normalisateur *ad hoc* a été créé. C'est le W3C (Word Wide web Consortium). Il spécifie notamment le standard HyperText Markup Language (`http://www.w3.org`).

Un certain nombre de protocoles cryptographiques font l'objet d'une ou plusieurs RFC. Dans le cas de la cryptographie les principaux organismes normalisateurs sont le NIST, qui publie des FIPS (Federal Information Processing Standard), l'ISO et l'ANSI. Certaines entreprises, comme par exemple RSA, diffusent aussi des standards de fait.

En matière de sécurité des systèmes d'information, il faut citer :

– la norme ISO 17799 ;
– la norme ISO 15408, dite « Critères Communs »(CC), qui permet d'attribuer les niveaux de certification EAL (Evaluation Assurance Level) et ITSEC (Information Technology Security Evaluation Criteria) ;
– le « guide d'élaboration de politiques de sécurité des systèmes d'information » publié en France par la DCSSI (Direction Centrale de la Sécurité des Systèmes d'Information) ;
– les critères ITSEC.

Les équipements intégrant des fonctions cryptographiques (modules logiciels, boîtiers chiffrants, matériels réseau, etc.) peuvent faire l'objet d'un agrément par des organismes tels que la DCSSI ou le NIST (normes FIPS 140-1 et 140-2).

## Références

[1] Ainigmatias Cruptos, *Les conversions classiques entre types de données*, fichecrypto_109, ACrypTA, http ://www.acrypta.fr

[2] P. Barthélemy, R. Rolland, P. Véron, *Cryptographie, principes et mises en œuvre*, Hermes-Lavoisier, 2005

[3] DCSSI, *Mécanismes cryptographiques - Règles et recommandations concernant le choix et le dimensionnement des mécanismes cryptographiques de niveau de robustesse standard, version 1.10*, sur le site de la DCSSI, 19 Décembre 2006

[4] H. Delfs, H. Knebl, *Introduction to cryptography*, Springer-Verlag, 1998

[5] T. Pedersen, *A threshold cryptosystem without a trusted party*, Eurocrypt'91, LNCS vol. 547, p. 522-526, Springer-Verlag, 1991

[6] R. Rolland, *Sécurité des générateurs pseudo-aléatoires*, sur le site http ://www.acrypta.fr, 2008

[7] G. Zémor, *Cours de cryptographie*, Cassini, 2000

---

*27 février 2012*

R. Rolland, Institut de Mathématiques de Luminy, Case 907, 13288 Marseille cedex 9 et Association ACrypTA • *E-mail :* robert.rolland@acrypta.fr
*Url :* http://robert.rolland.acrypta.com