Théorie des codes

Compression, Cryptage, Correction

2005

Yves Denneulin, Jean-Guillaume Dumas, Gregory Mounié,
Jean-Louis Roch, Karim Samaké, Eric Tannier, Sébastien Varrette

Table des matières

Table des figures

Liste des tableaux

Table des algorithmes

Introduction

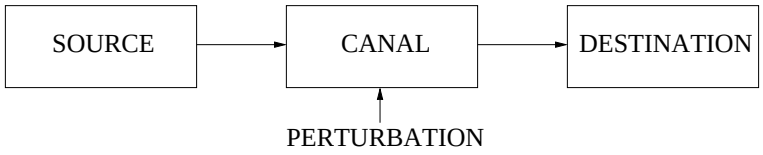

FIG. 1 – Schéma fondamental du codage

L'étude de la transmission de messages entre un émetteur et un destinataire via un canal de communication introduit les questions suivantes :

- Le canal permet de transmettre des signaux (souvent un signal électrique binaire). Il s'agit alors de transformer le message à émettre (par exemple un texte en français) en une séquence de signaux : on parle de *codage*. Le destinataire doit être capable de décoder la séquence de signaux reçus pour pouvoir lire le message émis.

- Dans un souci d'efficacité, la séquence de signaux doit être la plus courte possible. On s'intéresse donc à des codages qui minimisent la taille de la séquence émise : on parle de *compression*.

- Le medium peut introduire des erreurs : certains signaux émis peuvent être perdus ou altérés lors de la transmission. Dans ce cas, pour que le destinaitaire puisse décoder correctement le message reçu, on utilise des codages spécifiques, permettant de *détecter*, voire *corriger* les erreurs. On parle de *code détecteur/correcteur*.

- Le canal est généralement partagé par plusieurs émetteurs et destinataires. Pour qu'un émetteur puisse envoyer un message qui ne puisse être lu que par un destinataire spécifique (bien que la séquence de signaux associée soit visible par d'autres individus), le message doit être *crypté* grâce à un codage spécifique. La *cryptographie* étudie des codages et des protocoles permettant de communiquer des messages de manière secrète, de *signer* des documents ou d'*authentifier* un émetteur.

La théorie des codes et ses trois domaines d'application que sont la compression, la détection/correction d'erreurs et la cryptographie utilisent l'algèbre linéaire et les polynômes, l'arithmétique, des éléments de la théorie des

corps finis et quelques éléments de probabilités, de théorie des graphes et de manipulation des chaînes de symboles.

Le premier chapitre de ce cours présente les outils mathématiques essentiels utilisés. Dans le deuxième chapitre, nous présentons le concept de code et des propriétés fondamentales relatives à la déchiffrabilité puis, nous terminons ce chapitre par une esquisse du mécanisme utilisé pour passer du texte aux nombres. Les trois chapitres suivants, consacrés respectivement à la théorie de l'information et de la compression, à la cryptographie et aux détection et correction d'erreurs, présentent des résultats théoriques fondamentaux et les algorithmes génériques qui en découlent. Chacun de ces trois chapitres est illustré par une utilisation pratique dans le contexte des télécommunications.

Chapitre 1

Bases Mathématiques.

1.1 Ordres de grandeur

Dans ce livre, nous allons manipuler de très grands nombres. Les algorithmes permettant de les manipuler prennent un certain temps dépendant de la taille de ces nombres. Ainsi, une mesure pour essayer de prédire le temps de calcul est de compter le nombre d'opérations arithmétiques qu'ils nécessitent. Nous donnerons autant que possible cette mesure dans la suite du chapitre ainsi que le temps mis en pratique sur des exemples concrets.

Dans cette section, nous donnons quelques éléments astronomiques de comparaison, afin d'avoir une idée des ordres de grandeur en jeu :

Taille des nombres Nous considérons des nombres et leur taille, soit en chiffres, soit en bits. Ainsi, un nombre m possédera $\lceil \log[10](m) \rceil$ chiffres et $\lceil \log[2](m) \rceil$ bits. Par exemple, 128 bits font 39 chiffres ; 512 bits font 155 chiffres et 1024 bits sont représentables par 309 chiffres.

Quelle est la vitesse actuelle des ordinateurs ? Aujourd'hui n'importe quel PC est cadencé à au moins 1 GHz, c'est-à-dire qu'il effectue 1 milliard (10^9) d'opérations par seconde. À titre de comparaison, la vitesse la plus fantastique de l'univers est celle de la lumière : 300000 km/s $= 3.10^8$ m/s. Il ne faut que dix milliardièmes de seconde à la lumière pour traverser une pièce de 3 mètres ! Et bien, pendant ce temps là, votre ordinateur a donc effectué 10 opérations !!! On peut donc dire que *les ordinateurs actuels calculent à la vitesse de la lumière*. En outre si l'on suit la loi de Moore, cette vitesse tend à doubler tous les 18 mois.

Taille et âge de l'univers Malgré cette vitesse véritablement astronomique, la taille des nombres que l'on manipule reste énorme. En effet, rien que pour compter jusqu'à un nombre de 39 chiffres il faut énumérer 10^{39} nombres. Pour ce convaincre que c'est énorme, calculons l'âge de l'univers

en secondes :

$$Univers = 15\ milliards * 365.25 * 24 * 60 * 60 \approx 5.10^{17}\ secondes$$

Ainsi, un ordinateur cadencé à 1 GHz mettrait plus de deux millions de fois l'âge de l'univers pour simplement compter jusqu'à un nombre de "seulement" 39 chiffres ! Quant à un nombre de 155 chiffres (couramment utilisé en cryptographie), c'est tout simplement le carré du nombre d'électrons contenus dans l'univers. En effet, d'après S. Muller[1], notre univers contiendrait environ 3.10^{12} galaxies chacune renfermant grosso modo 200 milliards d'étoiles. Comme notre soleil pèse à la louche 2.10^{30} kg et qu'un électron ne pèse que $0,83.10^{-30}$ kg, on obtient :

$$Univers = (2.10^{30}/0,83.10^{-30}) * 200.10^{9} * 3.10^{12} \approx 10^{84}\ electrons$$

Nous allons voir dans la suite comment travailler avec de tels nombres entiers.

1.2 Eléments d'arithmétique dans $\mathbb{Z}$.

1.2.1 Algorithme d'Euclide

L'algorithme d'Euclide permet de calculer le pgcd de deux nombres entiers. Le principe est simple : si a et b sont deux entiers avec $b < a$ et ont un diviseur commun d alors $a - b$,$a - 2b$,... sont aussi divisibles par d.

Si $a - nb = 0$ cela signifie que a est divisible par b et le problème est donc résolu. Pour itérer l'algorithme on prend $n = a/b$ (i.e. le quotient entier de la division de a par b) ; on a alors $a - nb = a \bmod b$. On prend alors ce reste et on itère le procédé.

On itère donc la fonction G suivante :

$$G\ :\ \begin{bmatrix} a \\ b \end{bmatrix} \longmapsto \begin{bmatrix} 0 & 1 \\ 1 & -a \text{ div } b \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix}$$

Etant donnés a et b, l'algorithme d'Euclide permet de trouver $g = \text{pgcd}(a, b)$ et aussi x tels que : $x.b \equiv g \pmod{a}$.

Exemple : Déterminons le pgcd de 522 et 453. On calcule successivement :

(a)	522	
(b)	453	
(c)	$522 - 453 = 69$	$(a - b)$
(d)	$453 - 6 * 69 = 39$	$(b - 6c)$
(e)	$69 - 39 = 30$	$(c - d)$
(f)	$39 - 30 = 9$	$(d - e)$
(g)	$30 - 27 = 3$	$(e - 3f)$
(h)	$9 - 9 = 0$	$(f - 3g)$.

[1] http ://www.astrosurf.com/borealis/gogol.html

L'obtention de la valeur 0 arrête l'algorithme et 3 est donc le pgcd de 522 et 453. Pour trouver x et y tel que $x \times 522 + y \times 453 \equiv$ pgcd(522,453), on écrit les matrices correspondant à l'itération avec la fonction G, on a :

$$\begin{aligned}\begin{bmatrix} 3 \\ 0 \end{bmatrix} &= \begin{bmatrix} 0 & 1 \\ 1 & -3 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & -3 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 0 & 1 \\ 1 & -6 \end{bmatrix} \\ &\quad \begin{bmatrix} 0 & 1 \\ 1 & -1 \end{bmatrix}\begin{bmatrix} 522 \\ 453 \end{bmatrix} \\ &= \begin{bmatrix} 46 & -53 \\ -151 & 174 \end{bmatrix}\begin{bmatrix} 522 \\ 453 \end{bmatrix}\end{aligned}$$

On obtient ainsi $-151 \times 522 + 174 \times 453 = 3$. D'où $d = 3$, $x = -151$ et $y = 174$.

L'algorithme d'Euclide étendu EE ci-dessous réalise ce calcul en ne stockant que la première ligne de G. Il affecte les variables x, y et d de façon à vérifier en sortie : $d =$pgcd(a, b) et $ax + by = d$.

```
void EE(const Integer a, const Integer b, Integer& d, Integer& x, Integer& y) {
   if ( b==0 )  {
       d = a; x = 1; y = 1 ;
   }
   else {
       EE( b, a % b, d, x, y ) ;
       Int tmp = x ;
       x = y ;
       y = tmp - (a / b) * y ;
   }
}
```

A chaque étape, le diviseur étant supérieur à 2, le plus grand des nombres perd au moins 1 bit. Le nombre d'étapes est donc $O(\log_2(a+b))$. A chaque étape, on fait une multiplication et un calcul de quotient-reste de coût $O(\log_2^2 p)$. Le coût total est donc majoré[2] par $O(\log_2^3 p)$. La figure 1.1 de la page 18 donne une estimation du temps de calcul moyen pour calculer le pgcd de deux nombres choisis au hasard entre 1 et 10^n pour $n = 1, \cdots, 100000$ sur un Pentium III à 735MHz avec la bibliothèque GMP[3], une bibliothèque d'algorithmes écrits en C++ permettant de calculer avec des entiers de taille quelconque [17].

1.2.2 Théorème de Fermat

Soit $n \geq 2$ un entier. On note $\mathbb{Z}_n^*$ l'ensemble des entiers positifs plus petits que n et premiers avec n :

$$\mathbb{Z}_n^* = \{x \in \mathbb{N} : 1 \leq x < n \text{ et } \text{pgcd}(x, n) = 1\}.$$

[2]Les meilleurs algorithmes pour le calcul de l'inverse modulo ont un coût $O(\log_2 p. \log_2^2 \log_2 p. \log_2 \log_2 \log_2 p)$.

[3]voir à l'url http ://www.gnu.org/gnulist/production/gnump.html

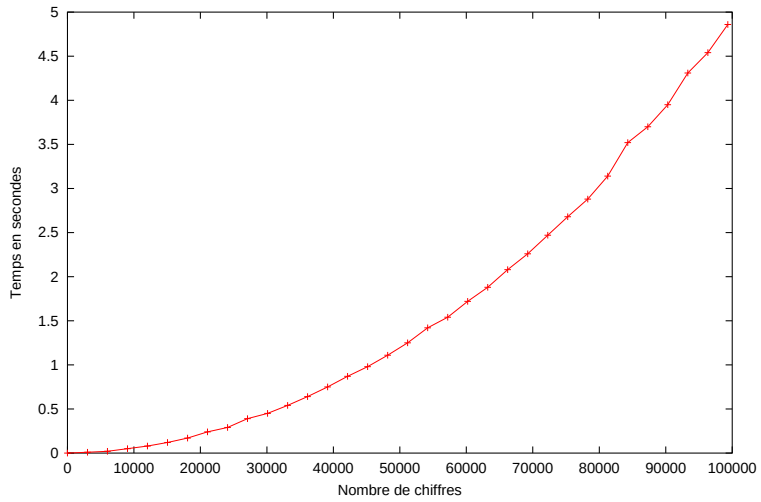

FIG. 1.1 – Algorithme d'Euclide sur un Pentium III à 735MHz avec GMP.

Le cardinal de $\mathbb{Z}_n^*$ est noté $\phi(n)$. La fonction ϕ est appelée fonction d'Euler. Par exemple, si p est premier, $\varphi(p) = p - 1$. De la même manière, dans $\mathbb{Z}/p^k\mathbb{Z}$, seuls les multiples de p ne sont pas premiers avec p. Ils sont $\{p, 2p, 3p, \ldots, p^{k-1}p\}$. Ainsi $\varphi(p^k) = p^k - p^{k-1} = (p-1)p^{k-1}$. Enfin, si m et n sont premiers entre eux, les entiers premiers avec mn sont chacun des entiers premiers avec m multipliés par chacun des entiers premiers avec n et seulement ceux là ; d'où l'on tire $\varphi(mn) = \varphi(m)\varphi(n)$. Au final, on obtient la proposition suivante :

Propriétés 1. ***[4, Théorèmes 7.1 & 7.2]*** *Soient $p_1, \ldots, p_n$ des entiers premiers alors,*

$$\forall (k_1, \ldots, k_n) \in \mathbb{N}, \varphi\left(\prod p_i^{k_i}\right) = \prod (p_i - 1) p_i^{k_i - 1}$$

Ainsi, pour calculer $\varphi(n)$, il faut savoir factoriser n.

Dans $\mathbb{Z}_n^*$, tout élément x a un inverse : en effet, comme x est premier avec n, l'identité de Bezout assure l'existence de deux entiers de signes opposés u et v ($1 \leq |u| < n$ et $1 \leq |v| < x$), tels que :

$$u.x + v.n = 1$$

On a alors $u.x \equiv 1 \bmod n$, i.e. $u = x^{-1} \bmod n$. On appelle u *inverse de x modulo n*. L'existence de u montre que $\mathbb{Z}_n^*$, muni de la loi de multiplication modulo n, est un groupe commutatif fini. Le calcul de u se fait grâce à l'algorithme d'Euclide étendu.

Théorème 1 (Théorème d'Euler). *Soit a un élément quelconque de $\mathbb{Z}_n^*$. On a : $a^{\phi(n)} \equiv 1$ modulo n.*

Preuve. Soit a un élément quelconque de $\mathbb{Z}_n^*$. Alors, l'ensemble des produits par a, $G_a = \{y = ax \bmod n \text{ pour } x \in \mathbb{Z}_n^*\}$ est en fait égal à $\mathbb{Z}_n^*$. En effet, quel que soit $y \in \mathbb{Z}_n^*$, on peut poser $x = a^{-1}y$ puisque a est inversible et reciproquement si a et x sont inversibles modulo n alors leur produit l'est aussi ($(ax)^{-1} = x^{-1}a^{-1} \bmod n$). Ainsi, comme ces deux ensembles sont égaux, les produits respectifs de tous leurs éléments sont aussi égaux modulo n :

$$\prod_{x \in \mathbb{Z}_n^*} x = \prod_{y \in G_a} y = \prod_{x \in \mathbb{Z}_n^*} ax$$

Or, comme la multiplication est commutative dans $\mathbb{Z}_n^*$, on peut alors sortir les a du produit, et comme il y a $\phi(n)$ éléments dans G_a on obtient la formule suivante modulo n :

$$\prod_{x \in \mathbb{Z}_n^*} x = a^{\phi(n)} \prod_{x \in \mathbb{Z}_n^*} x$$

La conclusion vient alors du fait que les éléments de $\mathbb{Z}_n^*$ étant inversibles, leur produit l'est aussi et donc, en simplifiant, on obtient $1 = a^{\phi(n)} \bmod n$. □

Le théorème de Fermat se déduit directement du théorème d'Euler dans le cas où n est un nombre premier.

Théorème 2 (Théorème de Fermat). *Si p est premier, alors pour tout $a \in \mathbb{Z}_p$: $a^p \equiv a$ modulo p.*

Preuve. Si a est inversible, alors le théorème d'Euler nous donne $a^{p-1} \equiv 1$ modulo p. En remultipliant par a on obtient la relation désirée. Le seul non-inversible de $\mathbb{Z}_p$ si p est premier est 0. Dans ce cas, on a immédiatement $0^p \equiv 0 \mod p$. □

1.2.3 Théorème chinois

Le théorème chinois, permet de combiner plusieurs congruences modulo des nombres premiers entre eux, pour obtenir une congruence modulo le produit de ces nombres.

Théorème 3. *Soient $m_1, \ldots, m_n$ des entiers positifs premiers entre eux. Alors, pour tous résidus $a_1, \ldots, a_n$ il existe un unique $x \leq \prod m_i$ tel que les restes de x modulo les m_i coïncident avec les a_i : $x \equiv a_i \mod m_i$.*

Preuve. Nous procédons en deux temps : nous montrons d'abord l'existence de x puis l'unicité. On pose $M = \prod m_i$ et $M_i = \frac{M}{m_i}$. Alors M_i et m_i sont premiers entre eux. Le théorème d'Euclide nous assure alors l'existence de y_i tel que $y_i M_i \equiv 1 \mod m_i$. Nous posons ensuite $x = \sum_{i=1}^{n} a_i y_i M_i \mod M$

et il est facile de vérifier que x ainsi défini convient ! En effet, pout tout i, $x \equiv a_i y_i M_i \mod m_i$ puisque m_i divise tous les M_j avec $j \neq i$. Mais alors la définition de y_i donne $x \equiv a_i.1 \equiv a_i \mod m_i$. Il nous reste à prouver l'unicité de x. Supposons qu'il en existe deux, x_1 et x_2. Alors $x_2 - x_1 \equiv 0 \mod m_1$ et $x_2 - x_1 \equiv 0 \mod m_2$. Donc $x_2 - x_1 = k_1 m_1 = k_2 m_2$ pour certains k_1 et k_2. Or, m_1 et m_2 sont premiers entre eux, donc, le théorème d'Euclide nous indique l'existence de u et v tels que $um_1 + vm_2 = 1$. Donc $u(x_2 - x_1) = k_1 u m_1 = k_1(1 - vm_2) = uk_2 m_2$, ce qui prouve que $k_1 = m_2(uk_2 + vk_1) = k_1' m_2$. Ainsi $x_2 - x_1 = k_1' m_2 m_1$ et donc $x_2 - x_1 \equiv 0 \mod m_1 m_2$. En procédant par récurrence, comme m_{i+1} est premier avec le produit $m_1 m_2 \ldots m_i$, on en déduit que $x_2 - x_1 \equiv 0 \mod M$, ou encore que $x_2 \equiv x_1 \mod M$, ce qui montre bien l'unicité. □

Ce théorème nous permet de calculer la fonction d'Euler de la manière suivante :

$$\varphi(p_1^{e_1} \ldots p_k^{e_k}) = \prod_{i=1}^{k} p_i^{e_i}(p_i - 1)$$

En effet, nous avons vu que $\varphi(p) = p - 1$ si p est premier ; il est facile de voir que $\varphi(p^e) = p^e - p^{e-1}$ puisque tous les nombres non premiers avec p^k sont les multiples de p plus petits que p^k ; et le théorème chinois nous permet de conclure que $\varphi(n_1 n_2) = \varphi(n_1)\varphi(n_2)$ puisque m premier avec $n_1 n_2$ est équivalent à $m \mod n_1$ est premier avec n_1 et $m \mod n_2$ est premier avec n_2, si n_1 et n_2 sont premiers entre eux.

Enfin, en combinant le théorème d'Euler et le théorème chinois, on peut obtenir une relation de congruence commune à tous les éléments de $\mathbb{Z}_n$ lorsque n est le produit de deux nombres premiers. C'est un résultat de Rivest, Shamir et Adleman, à l'origine de la cryptographie à clef publique RSA, que nous verrons au chapitre 4.4.

Théorème 4 (Théorème RSA). *Soit $n = pq$ un produit de deux nombres premiers. Soit a un élément quelconque de $\mathbb{Z}_n$. On a : $\forall k, a^{k\phi(n)+1} \equiv a$ modulo n.*

Preuve. Il y a deux cas : si a est inversible dans $\mathbb{Z}_n$, alors on applique directement le théorème d'Euler et donc $a^{\phi(n)} \equiv 1 \mod n$. En élevant à la puissance k et en remultipliant par a, on obtient la relation désirée. Dans le deuxième cas, on utilise deux fois le théorème de Fermat. En effet, p est premier donc $a^{p-1} \equiv 1 \mod p$ si $a \neq 0 \mod p$. Donc, en élevant à la puissance $k(q-1)$ et en remultipliant par a, on obtient $a^{k\phi(n)+1} \equiv a \mod p$. Si a est nul la relation modulo p est immédiate. De la même manière on obtient une relation similaire modulo q : $a^{k\phi(n)+1} \equiv a \mod q$. Comme p et q sont premiers entre eux, il suffit maintenant d'appliquer l'unicité du théorème chinois, pour avoir $a^{k\phi(n)+1} \equiv a \mod pq$. □

1.2.4 Elévation à la puissance modulo

Il s'agit du calcul de $a^b \bmod n$ où a, b et n sont des entiers, avec $a, b < n$. Ce calcul ne nécessite que $\log_2 n$ multiplications modulo n.

Par exemple :

$$a^{11} = a \times a^{10} = a \times (a^5)^2 = a \times (a \times a^4)^2 = a \times (a \times ((a)^2)^2)^2$$

autrement dit 5 multiplications suffisent : 3 élévations au carré et 2 multiplications.

La méthode générale est appelée *élévation récursive au carré*; l'algorithme suivant, en C++, l'implémente pour tout n. Le type `Integer` implémente des entiers en précision infinie.

```
Integer PuissanceModulo (const Integer a, const Integer k, const Integer n) {
   if (k==0)  { return Integer(1) ; }
   else {
        Integer d = PuissanceModulo( a , k / 2, n ) ;
        d = (d*d) % n ;
        if (k % 2 == 1) d = (d*a)  % n
        return d ;
   }
}
```

À chaque appel, l'exposant k est divisé par 2. Il y a donc $\log_2 k$ appels. Lors de chaque appel, on effectue au plus 2 multiplications : élévation au carré de d et éventuellement multiplication par a. Ces opérations sont faites modulo n, donc sur des nombres de $\log_2 n$ chiffres. Même en utilisant les algorithmes de mutiplication naïfs (ceux vus à l'école primaire), le coût d'une telle multiplication est $O(\log_2^2 n)$.

Le coût final de l'algorithme est donc $O(\log_2 k \log_2^2 n)$. La figure 1.3 à la page 22) donne une estimation du temps de calcul moyen pour calculer les puissances d'un nombre a modulo un nombre b choisis au hasard entre 1 et 10^n pour $n = 1, \cdots, 100000$ (pour en savoir plus, voir [19], section 4.6.3). Ce coût est raisonnable par rapport à $O(\log_2 n)$ qui est le temps nécessaire pour la lecture de a ou l'écriture du résultat.

1.2.5 Génération de nombres premiers

Si l'on ne connait pas d'algorithme pour factoriser un entier n en temps polynomial $\log_2^{O(1)} n$, il est revanche relativement facile de générer un nombre premier p.

La génération d'un nombre premier peut être faite grâce au test suivant proposé par Miller-Rabin.

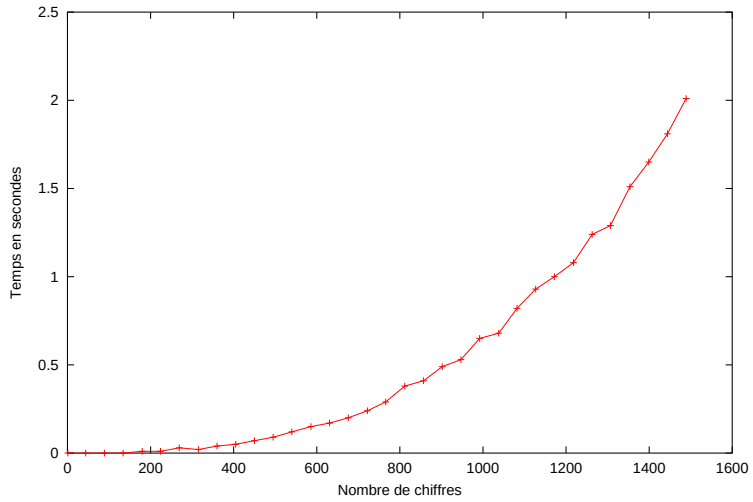

FIG. 1.2 – Calcul de puissances sur un Pentium III à 735 MHz avec GMP.

Test de Miller-Rabin Soit n un nombre impair et soient s et t tels que $n-1 = t2^s$ avec t impair. Pour un entier $a < n$ quelconque, on a alors

$$a^{(n-1)} - 1 = a^{t2^s} - 1 = (a^t - 1)(a^t + 1)(a^{2t} + 1) \dots (a^{(2^{s-1})t} + 1).$$

Si n est premier, d'après le théorème de Fermat, $a^{(n-1)} - 1 \equiv 0 \mod n$; donc
- soit $a^t - 1 \equiv 0 \mod n$;
- soit $a^{t2^i} + 1 \equiv 0 \mod n$ avec $0 \leq i < s$.

Le test de composition de Miller-Rabin est basé sur cette propriété. On dit que a réussit le test de composition de Miller-Rabin pour n si $a^t - 1 \not\equiv 0 \mod n$ et si $a^{t2^i} + 1 \not\equiv 0 \mod n$ pour tout $i = 0, \dots, s-1$. En effet, dans ce cas, $a^{(n-1)} - 1 \not\equiv 0 \mod n$; en utilisant le théorème de Fermat, on en déduit que n n'est pas premier (on dit aussi composé).

Si n est impair et non premier, moins de $(n-1)/4$ nombres a échouent au test de composition de Miller-Rabin. En choisissant a au hasard dans $\{1, \dots, n-1\}$, la probabilité d'échouer est donc inférieure à $\frac{1}{4}$.

Ce test peut être utilisé efficacement pour construire un nombre premier avec une probabilité inférieure à 4^{-k} de se tromper. On procède comme suit :
- On tire au hasard un nombre impair n ;
- On tire au hasard k nombres a_i distincts, $1 < a_i < n$. On applique le test de composition Miller-Rabin pour chaque entier a_i.
- Si aucun a_i ne réussit le test de composition, on en déduit que n est premier ; la probabilité de se tromper est inférieure à 4^{-k} ;
- sinon, on recommence en remplaçant n par $n+2$.

```
bool MillerRabin(const Integer n) {
    if (n < 2) return false;
    if (n <= 3) return true;
    Integer t=n-1,a,q;
    a = random() % n;
    long s=0;
    for( ; !(t & 0x1) ; t <<= 1)
        ++s
    q = PuissanceModulo(a,t,n);
    if ( (q==1) || (q == (n-1))) return true;
    for(;s>1;--s) {
        q = (q*q) % n;
        if (q == (n-1)) return true;
    }
    return false;
}
```

Le calcul du test de Miller-Rabin est similaire à celui de l'élévation à la puissance modulo ; il est donc en $O(\log_2^3 n)$ (voir la figure 1.2 de la page 23). De plus, le nombre de nombres premiers inférieurs à n est asymptotiquement $n/\ln(n)$. On en déduit que partant de n impair, en moyenne $\ln(n)$ itérations suffisent pour trouver un nombre premier en ajoutant 2 à n à chaque itération. Le coût moyen de l'algorithme est donc majoré par $O(k \log_2^4 n)$.

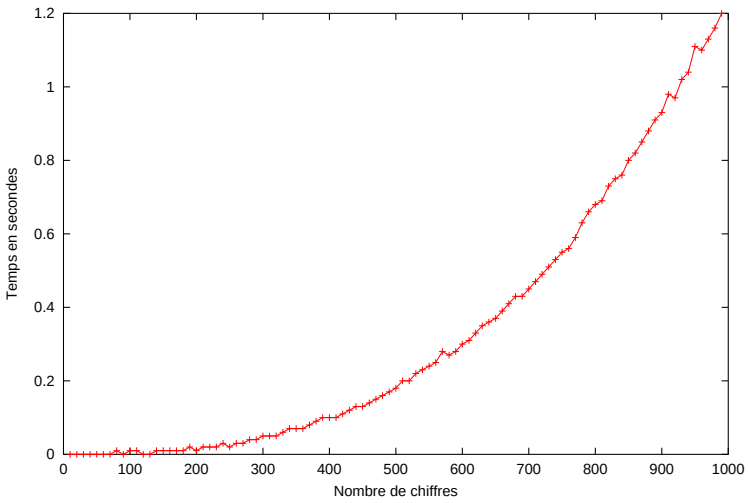

FIG. 1.3 – Test de Miller-Rabin sur un Pentium III à 735 MHz avec GMP.

En pratique, avec ce test il est facile de générer un nombre premier de 200 chiffres décimaux avec une probabilité d'erreur arbitrairement faible.

En outre, il est possible de rendre déterministe l'algorithme de Miller-Rabin en testant suffisamment de nombres a. Par exemple, Burgess a prouvé que tester tous les nombres a jusqu'à seulement $n^{0.134}$ est suffisant Toutefois, le test devient alors exponentiel en la taille de n. Finalement, Eric Bach a montré en 1990 qu'il suffit de tester les $2\log_2 n$ premiers entiers, mais son théorème n'est valide que si l'Hypothèse de Riemann Généralisée est vraie [1].

Test AKS Enfin, en août 2002, Manindra Agrawal, Neeraj Kayal et Nitin Saxena ont montré, sans l'hypothèse de Riemann généralisée, qu'il existe un algorithme déterministe pour tester si un nombre est premier ou non. L'idée est en fait voisine de celle de Miller-Rabin : si n est premier, l'automorphisme de Frobenius donne pour tout a

$$(X-a)^n \equiv (X^n - a)[n].$$

L'algorithme AKS vérifie cette égalité pour un certain nombre de témoins a, en développant explicitement $(X-a)^n$. Pour rendre le test polynomial, il faut réduire la taille des polynômes (ce qui est fait en effectuant le test modulo (X^r-1) pour r vérifiant certaines propriétés[4]) et il faut suffisament de témoins a, mais seulement de l'ordre de $log^{O(1)}(n)$.

1.2.6 Factorisation

La factorisation d'entiers est un problème qui peut s'exprimer de manière relativement simple mais qui n'a pas, jusqu'à présent, de solution vraiment efficace (le crible d'Eratosthène, par exemple, est inutilisable pour des nombres de plus de *10* chiffres).

De nombreux algorithmes très différents existent, le but de cette section n'est pas de les énumérer tous mais plutôt de donner une idée de ceux qui sont les plus efficaces pour différentes tailles de nombres à factoriser.

Rho de Pollard (Nombres de quelques chiffres) La première catégorie de nombres à factoriser est celle des "nombres composés de tous les jours" c'est-à-dire les nombres qui ont moins de *20* chiffres. Un algorithme très efficace est celui de Pollard [20]. L'algorithme ne nécessite que quelques lignes de code (une quarantaine seulement au total) et est très simple à programmer. Il faut tout d'abord calculer une suite du type $u_{k+1} = f(\ u_k\ )\ [m]$ de grande période, de sorte que les u_k soient distincts le plus longtemps possible.

Ensuite, l'idée est que si p est un facteur de n les u_k distincts modulo m le seront moins souvent modulo p, voir tableau 1.1. Dans ce cas, si $u_i \equiv u_j[p]$

[4] r doit être premier avec n, le plus grand facteur premier q de r doit vérifier $q \geq 4\sqrt{r}\log n$, et enfin il faut que $n^{\frac{r-1}{q}} \neq 1 \mod r$.

0	1	2	...	p	p+1	p+2	...	kp	kp+1	kp+2	...	m-1
		u_i	u_l		u_k			u_h		u_j		

TAB. 1.1 – Distribution des multiples de p modulo m

alors le pgcd de m et de $u_i - u_j$ vaut kp, et est donc un facteur non trivial de m ! Si les u_i sont bien tous distincts, le calcul se termine alors en au plus p étapes. Une première version de l'algorithme consiste donc à produire des u_i et à chaque nouvel élément de calculer les pgcd avec tous les précédents u_k. Cette version présente deux défauts : tout d'abord il faut stocker de l'ordre de p éléments, et en outre, il faut faire j^2 calculs de pgcd si i et $j > i$ sont les plus petits indices tels que $u_i \equiv u_j[p]$. La deuxième astuce de Pollard est donc d'utiliser la détection de cycle de Floyd. Il s'agit de stocker uniquement les u_k tels que k soit une puissance de 2. En effet, puisque les u_k sont générés par une fonction, si $u_i \equiv u_j$, alors, $\forall h \geq 0, u_{i+h} \equiv u_{j+h}$ et un cycle se crée modulo p, même si il n'est pas visible directement.

En ne stockant que des puissances de 2, le cycle sera donc détecté seulement pour $u_{2^a} \equiv u_{2^a+j-i}$ avec $2^{a-1} < i \leq 2^a$, voir figure 1.4. Or, $2^a + j - i <$

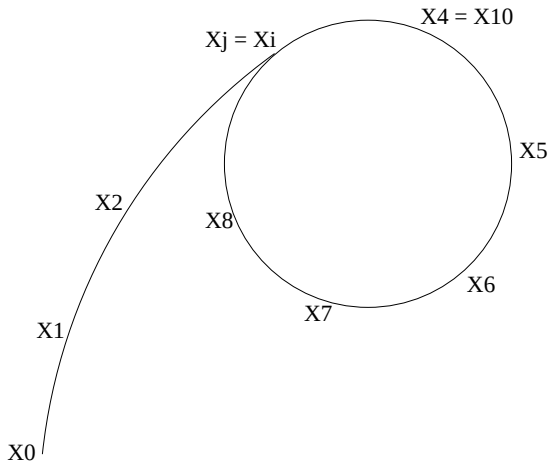

FIG. 1.4 – Détection de cycle de Floyd, sous forme de rho

$2i + j - i = i + j < 2j$. Donc on effectue au plus le double d'étapes nécessaires. Seulement, un seul pgcd est calculé à chaque étape et un seul élément supplémentaire est stocké au long de l'algorithme. Cela donne l'algorithme extrêmement simple suivant :

Si l'on prend $f(u) = u^2 + 1$, de sorte que les u_i soient bien tous distincts modulo m par exemple, il faudra au pire $2p$ itérations si p est le plus petit facteur de m, mais beaucoup moins en général :

Théorème 5. *L'algorithme Rho de Pollard a plus d'une chance sur deux de se terminer en $O(\sqrt{p})$ étapes.*

Algorithme 1 Factorisation de Pollard

Entrées Un entier m, composé.
Sorties p, un facteur non trivial de m.

```
p = 1 ;
Générer aléatoirement y ;
k = 0 ;
Tant que (p == 1) Faire
  Si k est une puissance de 2 Alors
    x=y ;
  Fin Si
  y = f(y) [m] ;
  p = pgcd(y-x,m) ;
  Incrémenter k ;
Fin Tant que
Renvoyer p.
```

Preuve. La preuve est celle du paradoxe des anniversaires de la section 4.7.2. Si k valeurs distinctes u_i sont tirées au hasard, il y a A_p^k combinaisons qui ne donnent pas de collision entre les u_i sur un total de p^k. La probabilité de trouver un facteur est donc de $P = 1 - \frac{A_p^k}{p^k} = 1 - (1 - \frac{1}{p})(1 - \frac{2}{p}) \dots (1 - \frac{k-1}{p})$. Or, pour $0 < x \leq 1$, $e^{-x} > 1 - x$, donc $P > 1 - \prod_{i=0}^{k-1} e^{-\frac{i}{p}} = 1 - e^{-\frac{k(k-1)}{2p}}$. Ainsi, pour avoir $P > \alpha$, il suffit que $k(k-1) > 2p\ ln(1-\alpha)$. Pour $\alpha = 0.5$, cela donne $k > 0.5 + \sqrt{2ln(2)}\sqrt{p} \approx 0.5 + 1.17741002\sqrt{p}$. □

En pratique, cet algorithme factorise en quelques secondes les nombres de *1* à environ *25* chiffres (avec des facteurs de *12* ou *13* chiffres, cela donne environ 10 millions d'itérations !) et devient très rapidement inutilisable pour des facteurs au-delà de *15* chiffres.

Courbes elliptiques (Nombres de quelques dizaines de chiffres) Pour aller plus loin, la méthode des courbes elliptiques de Pollard et Lenstra [20] est une solution. Celle-ci utilise des courbes elliptiques (courbes du type $y^2 = x^3 + a\ x + b$) dont l'étude dépasse le cadre de ce livre. Néanmoins, sa programmation reste simple (environ 150 lignes de code) et nous pouvons donner quelques idées des propriétés de cet algorithme : il est conjecturé que pour factoriser un entier m, de plus petit facteur p, cet algorithme nécessite un nombre moyen d'opérations de l'ordre de

$$O\left(\ln(m)^2 e^{\sqrt{2\ln(p)\ln(\ln(p))}}\right)$$

En pratique, cet algorithme factorise en quelques secondes les nombres de *25* à *40* chiffres (avec deux facteurs de tailles semblables). En outre, si l'on a de la chance et que l'on choisit une courbe elliptique particulière, celle-ci

peut permettre de factoriser très rapidement : le projet ECMNET (Elliptic Curve Method on the Net) consiste à fournir une l'implémentation de cet algorithme, disponible sur internet[5]. Cet projet a permis de factoriser des nombres avec des facteurs jusqu'à *58* chiffres (record en décembre 2004 : un facteur de 58 chiffres a été trouvé par GMP-ECM le 31 octobre 2003 en seulement 113 secondes de calcul : $\frac{8.10^{141}-17}{9} = 55514337029074222988418639646932672446139101589665 69 \times 32131622766403394135660479154180649695503836925499813337 01$). Le problème est que les bonnes courbes elliptiques varient pour chaque nombre à factoriser et qu'il n'existe pas encore de moyen de trouver la bonne courbe pour un nombre donné. Toutefois, cette rapidité de calcul lorsque l'on a une "bonne" courbe elliptique est à l'origine du programme de factorisation sur internet[6] : en effet, de nombreux internautes peuvent récupérer le programme ECM et le lancer pour qu'il essaye différentes courbes elliptiques. Ainsi, un nombre très important de courbes elliptiques peut être exploré dans le même temps et accélérer potentiellement la recherche de facteurs premiers.

Cribles de corps de nombres (Le champion du monde) Enfin, le champion actuel pour la factorisation des codes RSA est l'algorithme "Number Field Sieve" [20] ou crible de corps de nombre, qui, pour factoriser un nombre m composé de deux facteurs de tailles respectives similaires, semble nécessiter un nombre moyen d'opérations de l'ordre de

$$O\left(e^{\sqrt[3]{(7.11112)\ln(m)\ln(\ln(m))^2}}\right)$$

Un exemple d'idée de crible est celle du crible quadratique dont le crible de corps de nombre est une généralisation. Il s'agit de trouver des couples de nombres tels que leurs carrés soient congrus modulo m : $x^2 \equiv y^2$ [m]. Alors, $x^2-y^2=(x-y)(x+y)$ est un multiple de m et, avec de la chance, l'un de ces deux nombres permet donc de décomposer m. Toute la difficulté de l'algorithme consiste à trouver de tels entiers x et y !
Si l'idée de base est relativement simple, la programmation est un peu plus délicate que pour les précédents algorithmes mais cet algorithme détient les records actuels avec, en particulier, la factorisation, le 22 août 1999, d'une clé RSA de *155* chiffres (*512* bits). Le temps de calcul nécessaire pour cette dernière factorisation a été gigantesque : *7* mois de calcul sur *300* PC et stations de travail repartis dans *12* sites et *6* pays, une machine SGI origin 2000 et un Cray C196!!! Le record en décembre 2004 est d'une clef RSA de 174 chiffres décimaux factorisée le 3 décembre 2003. Pour des nombres spéciaux, cet algorithme est même parvenu le 4 avril 2004 à factoriser un nombre de *248* chiffres[7].

[5] http ://www.loria.fr/∼zimmerma/ecmnet.html
[6] http ://www.npac.syr.edu/factoring/overview.html
[7] ftp ://ftp.cwi.nl/pub/herman/SNFSrecords

1.2.7 Logarithme discret

Si $g \in \mathbb{Z}_n^*$ est d'ordre $\phi(n)$, alors $\mathbb{Z}_n^* = \{g^i \pmod n / i \in \mathbb{N}\}$. On dit que g est un *générateur* ou une *racine primitive* de $\mathbb{Z}_n^*$. Un groupe qui admet un élément générateur est dit *cyclique*. Les valeurs de n pour lesquelles $\mathbb{Z}_n^*$ admet un élément générateur (i.e. $\mathbb{Z}_n^*$ est cyclique) sont : 2, 4, p^e et $2p^e$ pour tout nombre premier $p \geq 3$ et tout entier positif e.
Si g est un générateur de $\mathbb{Z}_n^*$, alors, pour tout $a \in \mathbb{Z}_n^*$, $\exists z \in \mathbb{N}$ tel que $g^z = a$ modulo n.

Définition 1. *Pour a dans $\mathbb{Z}_n^*$, le plus petit entier positif z tel que $g^z = a$ modulo n est appelé le* logarithme discret *(ou encore l'*index*) en base g de a modulo n ; on note $z = \log_g a$ modulo n.*

Le résultat suivant est connu sous le nom de théorème du logarithme discret.

Théorème 6 (du logarithme discret). *Si g est une racine primitive de $\mathbb{Z}_n^*$, alors $\forall x, y \in \mathbb{N}$:*

$$g^x \equiv g^y \ mod\ n \Longleftrightarrow x \equiv y \ mod\ \phi(n)$$

Preuve.

de $\Rightarrow$)
Comme la séquence de puissance de g est périodique de période $\phi(n)$, alors $g^x \equiv g^y \bmod n \Longrightarrow x \equiv y \bmod \phi(n)$.
de $\Leftarrow$)
Si $x \equiv y \bmod \phi(n)$, on a $x = y + k * \phi(n)$. Or $g^{\phi(n)} = 1 \bmod n$, d'où $g^x \equiv g^y \bmod n$. □

On admet ici que la fonction $f_g : x \mapsto g^x \bmod p$ est une fonction à sens unique :

- Il est facile de calculer $f_g(x)$. Le paragraphe 1.2.4 montre que cela peut être fait en $O(\log_2^3 p)$ avec un algorithme trivial.
- Etant donné y, il est difficile de calculer x tel que $g^x = y$. La seule méthode simple consiste en l'énumération exhaustive de tous les x possibles, et prend un temps $O(p)$. Aucun algorithme polynomial en $\log_2 p$ n'est connu pour ce problème.

Ainsi, si $p = 10^{100}$, le calcul de $f_g(x)$ demande moins de 10^8 opérations, soit moins de 1 seconde sur un PC. Par contre, l'énumération exhaustive pour calculer $f_g^{-1}(y)$ demande 10^{100} opérations !!!

1.2.8 Résolution du logarithme discret

Comme nous venons de le mentionner, la seule méthode simple pour résoudre le logarithme discret consiste en l'énumération exhaustive de tous les x possibles, et prend un temps $O(p)$, beaucoup trop important par rapport

à $\log_2 p$. En pratique, les algorithmes de factorisation peuvent être pour la pluspart adaptés pour résoudre également le logarithme discret :

- L'algorithme rho de Pollard pour le logarithme discret est adapté de l'algorithme "Baby Step/Giant Step" de Shanks et nécessite de l'ordre de $O(\sqrt{p})$ opérations et $O(\log_2 p)$ espaces mémoire.
- Les algorithmes d'index s'apparentent aux algorithmes de crible, et nécessitent de l'ordre de $O(p^{\frac{1}{3}})$ opérations.

1.3 Corps finis.

1.3.1 Groupes, Anneaux, Corps, Polynômes

Groupes

Nous commençons par énoncer quelques définitions et théorèmes essentiels sur les groupes. Nous les notons multiplicativement, puisque, par la suite, nous nous intéresserons au groupe multiplicatif des inversibles d'un corps fini.

Définition 2. *Un* groupe $(G, *)$ *est un ensemble muni d'un opérateur binaire interne vérifiant les propriétés suivantes :*

1. $*$ *est associative :* $\forall a, b, c \in G, a * (b * c) = (a * b) * c$.
2. *Il existe un élément neutre* $e \in G$, *tel que* $\forall a \in G, a * e = e * a = a$.
3. *Tout élément a un inverse :* $\forall a, \exists a^{-1} \in G, a * a^{-1} = a^{-1} * a = e$.

De plus, si la loi est commutative ($\forall a, b \in G, a * b = b * a$), *alors* G *est dit* abélien.

Définition 3. *Un groupe est dit* cyclique *si il possède un générateur. Autrement dit, si* $\exists g \in G, \forall a \in G, \exists i \in \mathbb{Z}, a = g^i$.

Par exemple, l'ensemble des entiers $\{0, 1, \ldots, m-1\}$, muni de la loi d'addition modulo m est un groupe cyclique généré par exemple par 1. De même l'ensemble des entiers $\{1, \ldots, 6\}$ muni de la loi de multiplication modulo 7 est un groupe cyclique généré par exemple par 3, car $1 = 3^0$, $2 \equiv 3^2 = 9$, $3 = 3^1$, $4 \equiv 3^4 \equiv 2 * 2$, $5 \equiv 3^5$, $6 \equiv 3^3$.

Proposition 1. *Le* cardinal *d'un groupe fini est le nombre de ses éléments. Si* G *est fini, alors le cardinal de tout sous-groupe de* G *divise le cardinal de* G.

Proposition 2. *L'ensemble des puissances d'un élément* a *d'un groupe* G *est un sous groupe de* G, *noté* $< a >$. *Si ce sous-groupe est fini son cardinal est l'*ordre de a.

Anneaux

Définition 4. *Un* anneau $(A, +, \times)$ *est un ensemble muni de deux opérateurs binaires internes vérifiant les propriétés suivantes :*

1. $(A, +)$ *est un groupe abélien.*
2. $\times$ *est associative :* $\forall a, b, c \in A, a \times (b \times c) = (a \times b) \times c$.
3. $\times$ *est distributive sur* $+$ *:* $\forall a, b, c \in A, a \times (b + c) = (a \times b) + (a \times c)$ *et* $(b + c) \times a = (b \times a) + (c \times a)$.

- *Si de plus* $\times$ *possède un neutre dans* A*,* A *est unitaire.*
- *Si de plus* $\times$ *est commutative,* A *est commutatif.*
- *L'ensembles des éléments de* A *inversibles par* $\times$ *est noté* A^*.

Pour un élément a d'un anneau A, on note $n \cdot a$ (ou plus simplement na) la somme $a + \cdots + a$ portant sur n termes égaux à a pour tout $n \in \mathbb{N}^*$ et $0 \cdot a$ pour désigner 0.

Définition 5. *Si l'ensemble* $\{k \in \mathbb{N}^* \ : \ k \cdot 1 = 0\}$ *(noyau de l'homomorphisme de groupe* $n \to n.1_{\mathbb{F}}$ *de* $\mathbb{Z}$ *dans* $\mathbb{F}$*) est vide, on dit que l'anneau* V *est de* caractéristique 0. *Dans le cas contraire, le plus petit élément de cet ensemble est appelé la caractéristique de* V.

Par exemple, $(\mathbb{Z}, +, \times)$ est un anneau unitaire commutatif de caractéristique 0 tandis que $(\mathbb{Z}/n\mathbb{Z}, +, \times)$ pour $n \in \mathbb{N} \setminus \{0, 1\}$ est un anneau unitaire commutatif qui est de caractéristique n.

Pour éviter la surchage dans les notations, nous noterons toujours (tant que cela ne nuit pas à la clarté de l'exposé) par $+$ (respectivement par $\times$) l'addition (respectivement la multiplication) de tout anneau. Il reste entendu que l'opération $+$ (respectivement $\times$) n'a de sens que si elle est considérée par rapport à l'ensemble auquel appartiennent les éléments qu'elle engage.

Définition 6. *Deux anneaux* $(V, +, \times)$ *et* $(W, +, \times)$ *sont* isomorphes *lorsqu'il existe une bijection* $f \ : \ V \longrightarrow W$ *vérifiant pour tous* x *et* y *dans* V *:*

$$f(x + y) = f(x) + f(y) \tag{1.1}$$
$$f(x \times y) = f(x) \times f(y). \tag{1.2}$$

Remarque 1.

-) *L'isomorphisme entre anneaux est une relation d'équivalence en ce sens qu'elle est reflexive, symétrique et transitive.*

-) *Lorsque deux anneaux sont isomorphes, on peut identifier l'un quelconque d'entre eux à l'autre. C'est ce que nous ferons le plus souvent.*

-) *Si* E *est un ensemble quelconque et* $(V, +, \times)$ *un anneau tel qu'il existe une bijection* $\phi \ : \ E \longrightarrow V$ *de* E *sur* V*, alors la structure d'anneau*

de V et la bijection ϕ permettent de définir une structure d'anneau sur E si l'on pose pour tous x et y dans E :

$$x + y = f^{-1}(f(x) + f(y)) \tag{1.3}$$
$$x \times y = f^{-1}(f(x) \times f(y)) \ . \tag{1.4}$$

L'anneau $(E, +, \times)$ ainsi défini est évidemment isomorphe à V.

Corps

Définition 7. *Un* corps $(A, +, \times)$ *est un ensemble muni de deux opérateurs binaires internes vérifiant les propriétés suivantes :*

1. $(A, +, \times)$ *est un anneau.*
2. $(A \setminus \{0\}, \times)$ *est un groupe.*

Ainsi, un anneau unitaire V dans lequel tout élément $x \in V \setminus \{0\}$ a un inverse pour la multiplication est appelé *corps commutatif*; l'inverse de x est noté x^{-1}. Un corps est dit commutatif lorsqu'il est commutatif en tant qu'anneau unitaire. La *caractéristique* d'un corps est sa caractéristique en tant qu'anneau.

Par exemple, $(\mathbb{Q}, +, \times)$ est un corps commutatif de caractéristique 0 tandis que pour tout nombre premier p, l'anneau $(\mathbb{Z}/p\mathbb{Z}, +, \times)$ est un corps commutatif de caractéristique p ; on le note $\mathbb{F}_p$.

Puisque tous les anneaux unitaires et corps qui nous interessent dans ce cours sont commutatifs, nous dirons désormais anneau (respectivement corps) pour désigner anneau unitaire commutatif (respectivement corps commutatif).

On dit qu'un sous ensemble W d'un corps V est un sous-corps de V lorsque les restrictions des opérations de V à W confèrent à W une structure de corps.

Par exemple, si V est un corps de caractéristique $p \neq 0$, alors l'ensemble $\{k \cdot 1 \ : \ k = 1, \ldots, p\}$ est un sous-corps de V ; on l'appelle *sous-corps premier* de V.

Définitions 1.
- *Le corps des nombres rationnels $\mathbb{Q}$ et les corps $\mathbb{Z}/p\mathbb{Z}$, pour p premier, sont appelés* **corps premiers**.
- *Les corps finis sont appelés* **corps de Galois**. *Ils sont notés $\mathbb{F}_q$ ou $GF[q]$, avec q le cardinal du corps.*

Donnons ici quelques propriétés élémentaires.

Propriétés 2. *i.) Si la caractéristique d'un corps est non nulle, c'est un nombre premier.*

ii.) Si un corps W est un sous-corps d'un corps V, alors V a une structure d'espace vectoriel sur le corps W (la multiplication d'un élément de F par un élément de V est considérée comme le produit d'un scalaire par un vecteur).

iii.) Si la caractéristique d'un corps fini est non nulle, le cardinal du corps est une puissance de la caractéristique.

iv.) Tous les corps finis de même cardinal sont isomorphes (deux corps sont isomorphes s'ils le sont en tant qu'anneaux).

v.) Le cardinal de tout sous-corps d'un corps fini est un diviseur du cardinal du corps.

vi.) L'ordre de tout élément d'un corps fini est un diviseur du cardinal du groupe des inversibles du corps.

vii.) Le groupe des inversibles, $\mathbb{F}^*$, d'un corps fini $\mathbb{F} = GF[q]$ est cyclique de cardinal $q-1$.

Preuve. Nous ne donnons ici que quelques unes des preuves, classiques.
de i.) Si (ab).1 = 0, alors (a.1)(b.1)=0, mais comme on est dans un corps, soit (a.1), soit (b.1) doit être nul.
de iii.) Soit V un corps fini. Le cardinal de V est noté $|V|$. Alors V est de caractéristique $p \neq 0$ et p est un nombre premier. Le corps V est un espace vectoriel sur son sous-corps premier $F = \{k \cdot 1 \ : \ k = 1, \ldots, p\}$ et cet espace vectoriel est de dimension finie (sinon V ne serait pas fini). Si l'on prend $d = \dim_F V$, alors $|V| = p^d$.
de iv.) si d=1
On suppose $|V| = p$ (c'est à dire $d = 1$). L'application $\phi \ : \ \mathbb{F}_p \longrightarrow V$ qui à tout $k \in \mathbb{F}_p$ associe $\phi(k) = k \cdot 1$ est une bijection et elle vérifie les conditions (1.1) et (1.2) de la page 30.
de iv.) en général
Soit W un corps fini de cardinal p^d. Le sous-corps premier de W ainsi que celui de V sont tous les deux isomorphes à $\mathbb{F}_p$ (d'après l'énoncé **d=1** précédent) donc V et W s'identifient chacun à un $\mathbb{F}_p$-espace vectoriel. Et puisque les $\mathbb{F}_p$-espaces vectoriels V et W ont la même dimension d, ils sont isomorphes.
v. et vi.) découlent directement du théorème de Lagrange dont la preuve est identique à celle que nous avons donnée du théorème d'Euler page 19.
vii.) Il s'agit de démontrer l'existence q'un générateur. Nous avons tout d'abord besoin du lemme suivant :

Lemme 1. *Soient x et y d'ordres m et n. Il esite un élément d'ordre $ppcm(m,n)$.*

Preuve. [Preuve du lemme] Soient $d = pgcd(m,n)$ et $m = dm'$, $n = dn'$. Posons $x' = x^d$ et $y' = y^d$, alors x' ety' sont d'ordres m' et n' premiers entre eux. Il s'en suit que $(x'y')^{m'n'} = 1_{\mathbb{F}_q}$. Réciproquement si $(x'y')^r = 1$, alors $(x'y')^{rm'} = 1$ et donc $(y')^{rm'} = 1$. Ce qui prouve que n', l'ordre de y', divise rm', et donc r puisque n' et m' sont premiers entre eux. De même m' divise r, et l'on conclut que l'ordre de $x'y'$ est bien $m'n' = ppcm(m,n)$. □ À partir de là, nous posons $\omega = ppcm$(ordres d'éléments de $\mathbb{F}_q$). Donc ω est le plus petit entier tel que $x^\omega = 1_{\mathbb{F}_q}$ pour tous les éléments du corps. Par suite ω doit

diviser $q-1$ puisque par Lagrange, $x^{q-1} = 1_{\mathbb{F}_q}$ pour tout inversible. Ensuite, puisque le polynôme $x^\omega - 1$ ne peut avoir qu'au plus ω racines dans un corps et que tous les inversibles du corps en sont racines, il faut nécessairement que $\omega = q-1$. Enfin, le lemme nous permet de construire un élément d'ordre ω. □

Enfin, on dit que deux corps V et W sont isomorphes lorsqu'ils sont isomorphes en tant qu'anneaux.

Proposition 3. *Soit V un corps fini tel que $|V| = p^d$ où p est un nombre premier et d un entier positif.*
i) *Si $d = 1$, alors V est isomorphe à $\mathbb{Z}/p\mathbb{Z}$.*
ii) *Tout corps fini de cardinal p^d est isomorphe à V.*

Preuve.
de i)
On suppose $|V| = p$ (c'est à dire $d = 1$). L'application $\phi : \mathbb{F}_p \longrightarrow V$ qui à tout $k \in \mathbb{F}_p$ associe $\phi(k) = k \cdot 1$ est une bijection et elle vérifie les conditions (1.1) et (1.2) de la page 30.
de ii)
Soit W un corps fini de cardinal p^d. Le sous-corps premier de W ainsi que celui de V sont tous les deux isomorphes à $\mathbb{F}_p$ (d'après l'énoncé **i)** précédent) donc V et W s'identifient chacun à un $\mathbb{F}_p$-espace vectoriel. Et puisque les $\mathbb{F}_p$-espaces vectoriels V et W ont la même dimension d, ils sont isomorphes. □

1.3.2 Racines primitives

Un générateur du groupe des inversibles de $\mathbb{Z}/m\mathbb{Z}$ est appelé racine primitive de m. Nous allons voir qu'il existe des racines primitives pour tous les nombres premiers, ainsi que pour quelques autres nombres particuliers, même si dans ce cas $\mathbb{Z}/m\mathbb{Z}$ n'est plus un corps.

Définition 8. *Une* **racine primitive** *d'un entier m est un entier premier avec m et d'ordre $\varphi(m)$ modulo m. La* **plus petite racine primitive** *de m est notée $\chi(m)$.*

Propriétés 3. *[4, Théorèmes 8.4 & 8.10]*
- *Si m possède une racine primitive, alors il en a exactement $\varphi(\varphi(m))$.*
- *Un entier $m > 1$ possède une racine primitive si et seulement si $m = 2, 4, p^k$ ou $2p^k$ avec p un nombre premier impair et $k > 0$ entier.*

Les propriétés précedentes nous indiquent donc dans quels cas nous pouvons espérer avoir une racine primitive. Il faut maintenant en calculer au moins une. Pour trouver une racine primitive d'un nombre, nous avons tout d'abord besoin de tester si un entier donné satisfait cette propriété. Pour ce faire, il suffit (!) de posséder des algorithmes de test de primalité et de

factorisation d'entiers, pour calculer $\varphi(m)$ puis tester si l'ordre de l'élément est bien $\varphi(m)$:

Algorithme 2 *Test-Racine-Primitive*

Entrées Un entier $m > 0$.
Entrées Un entier $a > 0$.
Sorties Oui, si a est une racine primitive de m ; Non dans le cas contraire.
  Si a et m ne sont pas premiers entre eux **Alors**
    Renvoyer "Non".
  Fin Si
  $\varphi_m = \varphi(m)$ {Factorisation de m et calcul par les propriétés 1}
  Pour tout p, premier et divisant φ_m, **Faire** {Factorisation de $\varphi(m)$}
    Si $a^{\frac{\varphi_m}{p}} \equiv 1[m]$ **Alors**
      Renvoyer "Non".
    Fin Si{Calcul récursif par carrés}
  Fin Pour
  Renvoyer "Oui".

Théorème 7. *L'algorithme* Test-Racine-Primitive *est correct.*

Démonstration. On utilise [4, Théorème 8.1] : soit un entier a, d'ordre k modulo m. Alors $a^h \equiv 1[m]$ si et seulement si $k|h$. On en déduit que si l'ordre de a est plus petit que $\varphi(m)$, comme il doit diviser $\varphi(m)$, nécessairement l'une des valeurs $\frac{\varphi_m}{p}$ sera un multiple de l'ordre de a. Dans le cas contraire, la seule valeur possible pour l'ordre de a est $\varphi(m)$. □

Une première méthode de calcul est alors d'essayer un à un tous les entiers plus petits que m, qui ne soient ni 1, ni -1, ni une puissance sur les entiers, et de trouver ainsi la plus petite racine primitive de m. De nombreux résultats théoriques existent [21, 11] prouvant qu'en général, il ne faut pas trop d'essais pour la trouver, de l'ordre de

$$\chi(m) = \left(r^4(\log(r)+1)^4 \log^2(m)\right)$$

avec r le nombre de facteurs premiers distincts de m [29]. En pratique, $\chi(m)$ semble être encore plus petit ; d'après Tomás Oliveira e Silva [23], il apparaît qu'environ 80% des nombres premiers inférieurs à 8910000000000 ont une racine primitive plus petite que 6 et, même, 306841261647 des plus petites racines primitives, sur ces 309582581120 premiers nombres premiers, sont plus petites que 23. Ainsi, dans plus de 99% de ces cas, il suffira de 18 tests pour découvrir une racine primitive d'un nombre premier.

Une autre méthode est de tirer aléatoirement des entiers plus petits que m et de tester si ceux-ci sont une racine primitive ou non. Étant donné qu'il y a $\varphi(p-1)$ racines primitives dans $\mathbb{Z}/p\mathbb{Z}$ pour p premier, la probabilité

d'en trouver une est de $\frac{\varphi(p-1)}{p-1}$ et donc l'espérance du nombre de tirages pour tomber sur une racine primitive est de $\frac{p-1}{\varphi(p-1)}$. Ce qui nous donne une meilleure chance que la force brute puisque Rosser et Schoenfeld [26, Théorème 15] ont montré l'inégalité suivante où C est la constante d'Euler, $C \approx 0.5772156649\ldots$:

$$\frac{m}{\varphi(m)} < e^C \log(\log(m)) + \frac{5}{2\log(\log(m))}, \forall m \geq 3, m \neq 223092870 \quad (1.5)$$

En outre, comme $223092871 = 317 \times 703763$ n'est pas premier, nous pouvons utiliser cette inégalité pour toutes les racines primitives de nombres premiers. Il est de plus conjecturé que $e^C \log(\log(m)) < \frac{m}{\varphi(m)}$ pour un nombre infini de m, cette borne semble donc très bonne. Ainsi, pour $7 \leq m \leq 8910000000000$, elle donne une valeur maximale d'environ 6.78330. En pratique c'est encore mieux, puisqu'il y a seulement 36 nombres premiers inférieurs à 10000000000 avec $\frac{p-1}{\varphi(p-1)} > 6.0$.

En conclusion, l'algorithme le plus rapide nous semble donc être une combinaison de ces deux méthodes. Pour 80% des nombres premiers inférieurs à 8910000000000, il fera moins de 4 tests et dans les autres cas, il fera en moyenne moins de 11 tests.

Enfin, le cas des entiers non premiers se ramène au cas premier à l'aide des théorèmes suivants :

Théorème 8. *[4, Théorème 8.9 et Corollaire]*

- *Si a est une racine primitive de p premier impair, alors a ou $a+p$ est une racine primitive de p^k pour $k \geq 2$.*
- *Si a est une racine primitive de p^k avec p premier impair, alors celui de a et $a+p^k$ qui est impair est une racine primitive de $2p^k$.*

Nous en déduisons l'algorithme général 3.

En pratique, cet algorithme, exécuté sur un pentium cadencé à 333 MHz, et implémenté en utilisant *Gnu Multiprecision Package*[8] comme arithmétique, n'a jamais mis plus d'un dixième de seconde pour trouver une racine primitive d'entiers aléatoires bornés par 2^{64} !

Pour des nombres plus grands, le calcul de $\varphi(m)$ et sa factorisation vont être les facteurs limitant la production de racines primitives. Il faut alors de calculer des *pseudo racines primitives* ou *racines primitives industrielles* : l'idée est de ne factoriser $\varphi(p)$ que partiellement, pour trouver ses petits facteurs (si p est premier, $\varphi(p)$ est au moins divisible par 2). Si la factorisation devient trop difficile, c'est-à-dire si $\varphi(p) = kQ$, avec k entièrement factorisé et Q ne contenant que des gros facteurs premiers, on arrête la factorisation et l'on effectue le test de racine primitive avec les facteurs de k et en faisant comme si Q était premier. On peut alors montrer qu'un nombre passant ce

[8] http ://www.swox.com/gmp

Algorithme 3 *Racine-Primitive*

Entrées Un entier $m = 2$, 4, p^k ou $2p^k$ avec p premier impair.
Sorties a une racine primitive de m.

Si $m == 2$ **Alors**
  Renvoyer 1.
Fin Si
Si $m == 4$ **Alors**
  Renvoyer 3.
Fin Si
Pour $b = 2, 3, 5, 6$ **Faire**
  Si *Test-Racine-Primitive* (p, b) == "Oui" **Alors**
    $a = b$
  Fin Si
Fin Pour
Tant que *Test-Racine-Primitive* (p, a) == "Non" **Faire**
  Sélectionner a au hasard entre 7 et $p - 1$.
Fin Tant que
Si *Test-Racine-Primitive* (p^k, a) == "Non" **Alors**
  $a+ = p$
Fin Si
Si $m == p^k$ ou a est impair **Alors**
  Renvoyer a.
Sinon
  Renvoyer $a + p^k$.
Fin Si

"test industriel" a une très grande probabilité d'être une racine primitive[9]. En particulier, il est ainsi relativement facile d'obtenir des nombres étant racine primitive avec une probabilité de se tromper de l'ordre de seulement $\frac{1}{2^{50}}$ (une sur un million de milliards !). Cette probabilité est beaucoup plus faible que par exemple celle qu'un processeur ait perdu un bit dans un calcul à cause de radiations cosmiques ...

1.3.3 Anneau des polynômes sur un corps commutatif.

Soit V un corps. L'anneau des polynômes à coefficients dans V est

$$V[X] = \left\{ P = \sum_{i=0}^{d} a_i X^i / d \in \mathbb{N}, (a_0, \dots a_d) \in V^{d+1} \right\}.$$

On note $d = \deg(P)$ le degré du polynôme P. Dans cet anneau, il existe une division euclidienne (on dit alors que $V[X]$ est un anneau *euclidien*) qui se

[9] exactement une probabilité $\frac{\varphi(Q)}{Q-1}$, qui vaut donc 1 si Q et premier et se rapproche de 1 très vite quand les facteurs premiers de Q grandissent.

traduit comme suit : $\forall A, B \in V[X] : \exists!(Q, R) \in V[X]$ avec $\deg(R) < \deg(B)$ tels que :

$$A = B.Q + R \ .$$

Le polynôme Q est le *quotient* dans la division euclidienne de A par B ; le *reste* R est aussi noté $A \mod B$.

La notion de *pgcd* est donc définie ; l'*algorithme d'Euclide* appliqué à deux polynômes non nuls A et B est valide et fournit un polynôme de degré maximal (unique si on le choisit unitaire) qui divise à la fois A et B. Par ailleurs, l'idendité de Bezout est valide. Autrement dit, si A et B sont deux polynômes dans $V[X]$ qui admettent un polynôme $D \in V[X]$ comme pgcd, alors il existe deux polynômes S et T dans $V[X]$ tels que

$$A.S + B.T = D \ .$$

De plus, l'*algorithme d'Euclide étendu* donne un algorithme qui permet de calculer effectivement deux polynômes S et T dont les degrés respectifs sont strictement inférieurs à ceux de A/D et B/D.

Deux polynômes A et B sont dits *premiers entre eux* s'ils admettent le polynôme 1 comme pgcd ; autrement dit, A et B n'admettent aucun facteur commun de degré non nul. On dit qu'un polynôme P de $V[X]$ est un *polynôme irréductible* sur V lorsque P est premier avec tout polynôme de degré inférieur à $\deg P$.

Tout polynôme de degré non nul admet une décomposition unique en produit de puissances de facteurs irréductibles sur V ; on dit que $V[X]$ est un *anneau principal*. Autrement dit, on peut décomposer tout polynôme de degré non nul A de $V[X]$ sous la forme

$$A = P_1^{d_1} \dots P_k^{d_k}$$

où les d_i sont des entiers non nuls et les polynômes P_i sont irréductibles sur V. Si A est unitaire, les facteurs P_i peuvent être choisis unitaires : la décomposition est alors unique à une permutation d'indices près.

Le théorème fondamental de l'algèbre montre que tout polynôme de degré n a exactement n racines – distinctes ou confondues – dans une extension algébrique suffisante (la clôture du corps V, i.e. le plus petit corps qui contient toutes les racines de tous les polynômes irréductibles à coefficients dans V). Si α est une racine de A, alors $A(\alpha) = 0$ donc $(X - \alpha)$ divise A. Soit B le polynôme tel que $A = (X - \alpha).B$. On dit que α est racine *simple* de A si α n'est pas racine de B, i.e. $B(\alpha) \neq 0$. Sinon, dans le cas où $B(\alpha) = 0$, on dit que α est racine *multiple* de A.

Lorsque toutes les racines de A sont simples, on a $A = P_1 \dots P_n$ car les d_i sont alors tous égaux à 1 (en effet, $d_i \geq 2$ indique que A a au moins une racine multiple dans la clôture algébrique de V).

En particulier, le polynôme $A = X^n - 1$ de $V[X]$ a toutes ses racines simples ; donc, il s'écrit comme produit :

$$X^n - 1 = P_1 \dots P_k$$

où les P_i sont des polynômes unitaires, irréductibles sur V et distincts.

Exemple : Dans $\mathbb{F}_2[X]$ le polynôme $X^3 - 1$ se décompose en :

$$X^3 - 1 = (X - 1) \cdot (X^2 + X + 1) \ .$$

On vérifie aisément que $X^2 + X + 1$ est irréductible (les seuls polynômes de $\mathbb{F}_2[X]$ de degré non nul et inférieur à 2 sont X et $X - 1$ et aucun d'entre eux ne divise $X^2 + X + 1$).

1.3.4 Anneau quotient

Soit $(V, +, \times)$ un corps et soit P un polynôme de degré $d \geq 1$. Pour tout polynôme $A \in V[X]$, désignons par $A \mod P$ le reste de la division euclidienne de A par P. L'ensemble $\mathcal{P}_d$ des polynômes de degré inférieur strictement à d muni des opérations arithmétiques d'addition et de multiplication suivantes :

$$\forall A \in \mathcal{P}_d \, \forall A \in \mathcal{P}_d \ \left(A +_{\mathcal{P}_d} B = (A +_{V[X]} B) \mod P \right)$$
$$\forall A \in \mathcal{P}_d \, \forall A \in \mathcal{P}_d \ \left(A \times_{\mathcal{P}_d} B = (A \times_{V[X]} B) \mod P \right)$$

est un anneau commutatif unitaire.

En effet, on vérifie aisément que $(\mathcal{P}_d, +_{\mathcal{P}_d}, \times_{\mathcal{P}_d})$ est un anneau commutatif unitaire qui admet les polynômes 0 et 1 comme éléments neutres respectivement pour les lois $+_{\mathcal{P}_d}$ et $\times_{\mathcal{P}_d}$.

Cet anneau est noté $V[X]/(P)$ et appelé anneau quotient de $V[X]$ par P. Par abus de langage, nous notons dans la suite les opérations $+_{\mathcal{P}_d}$ et $\times_{\mathcal{P}_d}$ respectivement par $+$ et $\times$.

Si P est un polynôme irréductible, et Q un polynôme non nul de degré inférieur à $\deg P$, alors Q est premier avec P et l'identité de Bezout s'écrit :

$$\exists! A_P, B \in V[X], \quad A_P.Q + B.P = 1 \ ,$$

ce qui implique que $A_P \cdot Q \mod P = 1$ autrement dit, que Q est *inversible* dans l'anneau quotient $V[X]/(P)$. On en déduit que $V[X]/(P)$ est un corps.

Exemple : Sur le corps $V = \{0, 1\}$:

-) Si l'on prend $P = (X+1)(X^2+X+1)$, l'anneau $V[X]/(P)$ est :

$$V[X]/(P) = \left\{0, 1, X, 1+X, X^2, 1+X^2, X+X^2, 1+X+X^2\right\} .$$

Cet anneau n'est pas un corps car $(1+X)(1+X+X^2) = 0$ montre que la classe $1+X$ n'est pas inversible.

-) Si l'on prend $P = X^2+X+1$, l'anneau $V[X]/(P)$ est :

$$V[X]/(P) = \{0, 1, X, 1+X\} .$$

Cet anneau est un corps puisque $X(X+1) = 1$.

1.3.5 Polynômes irréductibles

Pour fabriquer des corps finis, nous allons donc avoir besoin de polynômes irréductibles, nous commençons donc par donner un test permettant de les reconnaitre.

Proposition 4. *Pour p premier et $d \geq 1$, dans $\mathbb{Z}/p\mathbb{Z}[X]$, le polynôme $X^{p^d} - X$ est le produit de tous les polynômes unitaires irréductibles dont le degré divise d.*

Preuve. Soit P est irréductible de degré r avec r divisant d, alors $V = \mathbb{Z}_p/P$ est un corps. Tout d'abord, si $r|d$ dans le corps V on a $X^{p^r-1} \equiv 1$, puisque l'ordre de tout élément divise $p^r - 1$, or $p^d \equiv (p^r)^k \equiv 1[p^r-1]$, donc $p^r - 1 | p^d - 1$, et donc $X^{p^d} \equiv X[P]$.
Reciproquement, si $P|X^{p^d} - X$ est irréductible, alors $X^{p^d-1} \equiv 1[P]$. Dans ce cas on a $p^d - 1 \equiv 0[p^r-1]$. Supposons que $d = qr + s$, avec $s < r$, on a alors $p^d - 1 \equiv p^{qr}p^s - 1 \equiv p^s - 1 \equiv 0[p^r-1]$. Comme $s < r$, alors forcément on obtient $p^s - 1 = 0$, ce qui implique que $s = 0$. donc r divise d.
Il reste à montrer qu'aucun carré ne divise $X^{p^d} - X$. En effet, son polynôme dérivé est $p^d X^{p^d-1} - 1 \equiv -1 \mod p$ et donc ces deux polynômes sont bien entendu premiers entre eux. □

Notation 1. *Nous notons $m_r(p)$, le nombre de polynômes unitaires irréductibles de degré r dans $\mathbb{Z}/p\mathbb{Z}[X]$.*

Corollaire 1. *Si f est premier, alors $m_f(p) = \frac{p^f - p}{f}$.*

Preuve. $\frac{p^f-p}{f}$ est un entier d'après le théorème de Fermat modulo f. Comme f est premier, les seuls diviseurs de f sont f et 1 et donc les seuls diviseurs de $X^{p^f} - X$ sont les polynômes irréductibles de degré f ou 1. Donc $p^f = m_f(p) * f + p * 1$ □

C'est à partir de cette proposition que Ben-Or [3] a proposé un test d'irréductibilité, qui est simplement une variante de l'algorithme de factorisation de polynômes de Cantor et Zassenhaus [5], simplifiée pour nos besoins.

Algorithme 4 *Test-Irréductibilité*

Entrées Un polynôme, $P \in GF_q[X]$.
Sorties "Oui" si P est irréductible, "Non" sinon.

```
Si pgcd (P, P') ≠ 1 Alors
  Renvoyer "Non".
Fin Si{P est-il sans carré ?}
W = X
Pour d = 1 à (d∘P)/2 Faire
  W ≡ W^q [P]
  Si pgcd (W − X, P) ≠ 1 Alors
    Renvoyer "Non".
  Fin Si
Fin Pour
```

Le test de V. Shoup [30], par exemple, est plus efficace quand le polynôme est irréductible. En effet, il n'est pas nécessaire de calculer toutes les puissances successives W^q, il suffit de vérifier les $X^{q^{\frac{d}{p_i}}}$ pour tous les p_i facteurs premiers distincts de d. Néanmoins, si il faut tester plusieurs polynômes avant de tomber sur un polynôme irréductible, cet algorithme sera, en général, plus performant puisqu'il découvrira que les précédents polynômes sont réductibles avec des puissances moins grandes, et donc avec moins de calculs.

Ainsi, en utilisant ce test, il est possible de tirer des polynômes au hasard et d'avoir une bonne chance de tomber sur un polynôme irréductible. En effet, si $m_r(p)$ est le nombre de polynômes unitaires irréductibles de degré r dans $\mathbb{Z}/p\mathbb{Z}[X]$, alors comme $X^{p^r} - X$ est le produit de tous les polynômes irréductibles de degré divisant r, on obtient d'après [9, Corollaire 8.19] et [20, Proposition II.1.8] :

$$\frac{1}{r}(p^r - p^{\lfloor \frac{r}{2} \rfloor + 1}) \leq m_r(p) = \frac{1}{r}\big(p^r - \sum_{d|r; d<r} d m_d(p)\big) \leq \frac{1}{r}p^r \qquad (1.6)$$

Preuve. p^r est le degré de $X^{p^r} - X$, donc $p^r = \sum_{d|r} d m_d$, donc $m_r \leq p^r/r$. En appliquent cette majoration, à l'égalité précédente on obtient $p^r - r m_r \leq \sum_{d|r} p^d \leq \sum_{d \leq \lfloor r/2 \rfloor} p^d$. Cette dernière est une série géométrique qui vaut $\frac{p^{\lfloor r/2 \rfloor + 1} - 1}{p-1} < p^{\lfloor r/2 \rfloor + 1}$. □

Cela montre que parmi les polynômes, tirés au hasard, de degré r, environ un sur r est irréductible.

Corollaire 2. *Pour tout nombre premier p et tout entier $d > 0$, il existe un corps K à $q = p^d$ éléments et ce corps est unique à un isomorphisme près.*

Preuve. Soit p un nombre premier et $\mathbb{F}_p[X]$ l'anneau des polynômes à coefficients dans $\mathbb{F}_p$. D'après l'équation 1.6, il existe au moins un polynôme

P dans $\mathbb{F}_p[X]$, qui est irréductible sur $\mathbb{F}_p$ et de degré d. L'anneau quotient $\mathbb{F}_p[X]/(P)$ est alors un corps. Posons $K = \mathbb{F}_p[X]/(P)$.

Les éléments de K peuvent être vus comme les restes dans la division euclidienne par le polynôme P. Puisque P est de degré d et que $|\mathbb{F}_p| = p$, alors il y a p^d restes possibles et par conséquent $|K| = p^d$.

Le fait que tout corps fini à q soit isomorphe au corps K résulte de la proposition 3 de la page 33. □

Puisqu'à un isomorphisme près il existe un et un seul corps fini de cardinal cardinal $q = p^d$ (où p est un nombre premier et d un entier positif) on note généralement $\mathbb{F}_q$ (où $q = p^d$) pour un corps à q éléments.

Pour construire ce corps fini, il faut donc trouver un polynôme irréductible de degré d dans $\mathbb{Z}/p\mathbb{Z}$. Comme un polynôme sur d est irréductible, il suffit donc d'en choisir un au hasard, puis de tester son irréductibilité jusqu'à tomber sur un polynôme irréductible. En moyenne il faudra donc d tirages pour trouver un polynôme adéquat. Cependant, pour faciliter les calculs avec ces polynômes, il est intéressant d'obtenir des polynômes creux, c'est-à-dire avec très peu de coefficients non nuls. Dans ce cas, de même que pour les racines primitives, la recherche systématique peut s'avérer plus efficace en pratique.

Nous proposons donc l'algorithme hybride suivant produisant de préférence un polynôme irréductible creux ou un polynôme irréductible au hasard, à l'aide du test 4, si les deux précédents s'avèrent trop difficile à calculer. Par exemple, Coppersmith propose de prendre des polynômes du type $X^r + g(X)$ avec $g(X)$ au hasard de faible degré par rapport à r [6].

1.3.6 Constructions des corps finis

Ainsi, comme les corps finis de même cardinal $q = p^n$ sont tous isomorphes, il suffit d'en connaître un pour les connaître tous. Par exemple, soit P un polynôme irréductible de degré n dans $\mathbb{Z}/p\mathbb{Z}[X]$, l'ensemble $\frac{\mathbb{Z}/p\mathbb{Z}[X]}{P}$ peut être muni d'une structure de corps et est de cardinal p^n. Une construction classique de l'arithmétique dans un corps fini est donc d'implémenter $\mathbb{Z}/p\mathbb{Z}$, de chercher un polynôme irréductible, P, dans $\mathbb{Z}/p\mathbb{Z}[X]$ de degré n, puis de représenter les éléments de $GF[p^n]$ par des polynômes, ou des vecteurs, et d'implémenter les opérations arithmétiques comme des opérations modulo p et P. Cette méthode est utilisée dans la bibliothèque NTL [31], par exemple.

Une autre idée consiste à utiliser la dernière propriété 2 : le groupe multiplicatif des inversibles d'un corps fini est cyclique, c'est-à-dire qu'il existe au moins un générateur et que le corps est généré par les puissances de ce générateur. Il s'agit de représenter les éléments inversibles par un indice, leur puissance, et zéro par un indice spécial. Ainsi, si g est un générateur du groupe multiplicatif d'un corps fini $GF[q]$, tous les inversibles peuvent s'écrire g^i. De plus, comme ce groupe est cyclique, il suffit de considérer,

Algorithme 5 *Polynôme-Irréductible*

Entrées Un corps fini $\mathbb{F}_q$.
Entrées Un entier $r > 0$.
Sorties Un polynôme irréductible de $\mathbb{F}_q[X]$, de degré r.
  Pour tout $a \in \mathbb{F}_q, a \neq 0$ **Faire**
    Si *Test-Irréductibilité* $(X^r + a) ==$ "Oui" **Alors**
      Renvoyer $X^r + a$
    Fin Si
  Fin Pour
  Pour $d = 2$ à $r - 1$ **Faire**
    Pour tout $a \in \mathbb{F}_q, a \neq 0, b \in \mathbb{F}_q, b \neq 0$ **Faire**
      Si *Test-Irréductibilité* $(X^r + bX^d + a) ==$ "Oui" **Alors**
        Renvoyer $X^r + bX^d + a$
      Fin Si
    Fin Pour
  Fin Pour
  Répéter
    Sélectionner P, unitaire de degré r, au hasard dans $\mathbb{F}_q[X]$.
  Jusqu'à ce que *Test-Irréductibilité* $(P) ==$ "Oui"
  Renvoyer P.

pour i, un intervalle de taille q pour générer l'ensemble des inversibles. Les opérations arithmétiques classiques sont alors grandement simplifiées, en utilisant la proposition suivante :

Proposition 5. *Soit $GF[q]$ un corps fini. Soit g un générateur de $GF[q]^*$. Alors $g^{q-1} = 1_{GF[q]}$. En outre, si la caractéristique de $GF[q]$ est impaire, alors $g^{\frac{q-1}{2}} = -1_{GF[q]}$ et dans le cas contraire, $1_{GF[2^n]} = -1_{GF[2^n]}$.*

Démonstration. L'ordre de g divise $q-1$ donc $g^{q-1} = 1_{GF[q]}$. Si le corps est de caractéristique 2, alors, comme dans $\mathbb{Z}/2\mathbb{Z}[X]$, $1 = -1$. Sinon $\frac{q-1}{2} \in \mathbb{Z}$ et donc $g^{\frac{q-1}{2}} \in GF[q]$. Or comme on est dans un corps, l'équation $X^2 = 1$ possède au plus deux racines, 1 et -1. Or g est un générateur donc l'ordre de g est $q-1$ et non pas $\frac{q-1}{2}$. La seule possibilité restante est $g^{\frac{q-1}{2}} = -1$. □

Cela donne le codage suivant pour un élément $x \in GF[q]$ si GF[q] est généré par g :

$$\begin{cases} 0 & \text{si } x = 0 \\ q-1 & \text{si } x = 1 \\ i & \text{si } x = g^i \end{cases}$$

En particulier, dans notre codage, notons $\overline{q} = q-1$ le représentant de $1_{GF[q]}$. Nous notons aussi i_{-1} l'indice de $-1_{GF[q]}$; il vaut $\frac{q-1}{2}$ si $GF[q]$ est de caracté-

ristique impaire et $q-1$ sinon. Ce qui donne la possibilité d'écrire simplement les opérations arithmétiques.

- La multiplication et la division d'inversibles sont respectivement une addition et une soustraction d'indices modulo $q-1$.
- La négation est donc simplement l'identité en caractéristique 2 et l'addition de $\frac{q-1}{2}$ modulo $q-1$ en caractéristique impaire.
- L'addition est l'opération la plus complexe. Il faut l'implémenter en utilisant les autres opérations, par exemple de cette manière : si g^i et g^j sont deux éléments non nuls d'un corps fini, $g^i + g^j = g^i(1 + g^{j-i})$. En construisant une table, $t\_plus1[]$, de taille q, des successeurs de chaque élément du corps, l'addition est implémentée par une soustraction d'indice, un accès à une table et une addition d'indices.

Opération	Éléments	Indices	Coût		
			+/-	Tests	Accès
Multiplication	$g^i * g^j$	$i + j\ (-\overline{q})$	1.5	1	0
Division	g^i / g^j	$i - j\ (+\overline{q})$	1.5	1	0
Négation	$-g^i$	$i - i_{-1}\ (+\overline{q})$	1.5	1	0
Addition	$g^i + g^j$	$k = j - i\ (+\overline{q})$ $i + t\_plus1[k]\ (-\overline{q})$	3	2	1
Soustraction	$g^i - g^j$	$k = j - i + i_{-1}\ (\pm\overline{q})$ $i + t\_plus1[k]\ (-\overline{q})$	3.75	2.75	1

TAB. 1.2 – Opérations sur les inversibles avec générateur en caractéristique impaire

Dans le tableau 1.2, nous montrons le calcul de ces opérations sur les indices, en considérant une seule table de taille q, celle des successeurs. Nous nous intéressons ici à la complexité du calcul en utilisant le moins de mémoire possible, en considérant des éléments aléatoires. Nous indiquons le coût des calculs en nombre d'additions et de soustractions (+/-), en nombre de tests et en nombre d'accès dans une table.
Nous avons compté 0.5 additions ou soustractions pour un modulo $\overline{q}$ dans le cas de la multiplication, la division, la négation et l'addition dans le corps fini ; en effet, dans la moitié des cas, il n'est pas nécessaire d'ajouter ni de retirer $\overline{q}$ pour que l'indice résultant soit compris entre 0 et $\overline{q}$. Le cas de la soustraction dans le corps fini nécessite une étude plus fine. Dans ce cas, $j - i + i_{-1}$ est compris entre $-\frac{\overline{q}}{2}$ et $\frac{3\overline{q}}{2}$ et ne nécessite un ajout ou une soustraction de $\overline{q}$ que dans 2 cas sur 8 comme le montre la figure 1.5. On en déduit un total de $2 + 0.25 + 1 + 0.5 = 3.75$ additions ou soustractions et $1 + 0.75 + 1 = 2.75$ tests. Il est possible de réduire le nombre d'additions et soustractions, par exemple en remplaçant $i + x$ par $i + x - \frac{q-1}{2}$ et en

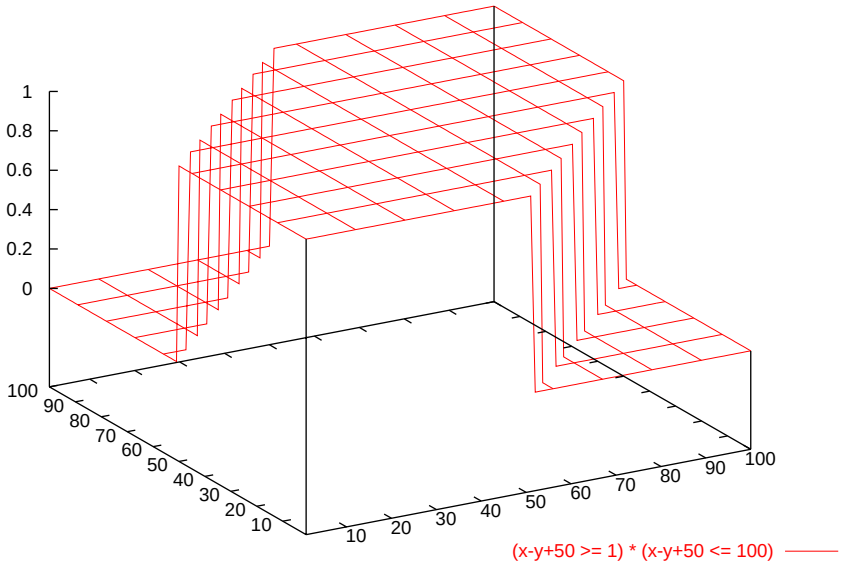

FIG. 1.5 – Répartition de $j - i + \frac{q-1}{2}$ lorsque i et j varient de 1 à $q-1$, pour $q = 101$

ajoutant un test. Cependant, pour de nombreuses architectures, un test du type $a > q$? peut être aussi coûteux que l'opération $a - q$. Nous proposons donc l'implémentation qui minimise le nombre total d'opérations (additions, soustractions, tests, accès), tout en n'utilisant qu'une seule table.

Ces opérations sont signées et restent valides tant que les indices sont plus petits que la moitié de la valeur maximale d'un entier machine. Alors, si un mot machine est codé sur 32 bits, la mantisse signée est codée sur 31 bits et ces opérations sont donc valables pour un corps de taille jusqu'à 30 bits, c'est-à-dire jusqu'à $2^{30} - 1$ éléments. Notons qu'il est possible de travailler avec des corps de taille 31 bits en utilisant l'arithmétique non signée et en ajoutant des tests de dépassement de capacité. En fait, la nécessité d'une table des successeurs implique souvent une plus petite limite du fait de l'espace mémoire disponible. Si l'on travaille sur des entiers codés sur 32 bits, c'est-à-dire 4 octets, un corps de taille 2^{26} nécessitera déjà au moins 256 Méga octets.

De cette manière, si un générateur est connu, les opérations et le stockage dans un corps fini peuvent être simples. Pour cela, il faudra générer le corps fini ($\mathbb{Z}/p\mathbb{Z}$ ou $\frac{\mathbb{Z}/p\mathbb{Z}[X]}{P}$) puis calculer un générateur. Il est alors possible de précalculer les puissances successives de ce générateur pour identifier les indices de chaque élément et stocker ainsi les correspondances entre éléments

et indices à l'aide de deux autres tables de taille q. Une autre possibilité, si la mémoire disponible est restreinte, ou si il y a peu de conversions entre représentation interne (les indices) et représentation externe (les éléments), est de faire ces calculs à chaque fois qu'une conversion est nécessaire.

Nous savons donc maintenant trouver un générateur pour un corps premier, et plus généralement pour tout sous-groupe $\mathbb{Z}/p^k\mathbb{Z}$ de $\mathbb{Z}$: il faut trouver une racine primitive. Mais si $k > 1$ un tel sous-groupe n'est pas un corps. Nous nous intéressons maintenant aux corps finis : les $GF[p^k]$ extensions des corps $GF[p] = \mathbb{Z}/p\mathbb{Z}$. Pour construire une telle extension, il faut tout d'abord construire le corps fini $GF[p]$, par la méthode de la section précédente, par exemple. Ensuite, il est possible d'utiliser un polynôme de degré k, irréductible sur ce corps, qui définit une extension de degré k, et donc de taille p^k, comme nous l'avons vu au début de la section.Nous commençons donc par construire l'extension avec des polynômes puis cherchons un *polynôme* générateur de cette extension pour pouvoir coder les éléments non plus par des polynômes, mais par leurs indices. Le codage et les opérations sont alors les *mêmes* que ceux des sous-groupes de $\mathbb{Z}$. Nous étudions ici les algorithmes permettant de trouver un polynôme irréductible possédant un générateur simple.

Nous utilisons de nouveau un algorithme probabiliste. Nous montrons tout d'abord un algorithme testant si un polynôme est générateur dans $GF[q][X]$. Cet algorithme est similaire à celui développé pour les racines primitives dans $\mathbb{Z}$.

Algorithme 6 *Test-Polynôme-Générateur*

Entrées Un polynôme $A \in GF[q][X]$.
Entrées Un polynôme F irréductible de degré d dans $GF[q][X]$.
Sorties Oui, si A est générateur de l'extension $\frac{GF[q][X]}{F}$; Non dans le cas contraire.
  Si A et F ne sont pas premiers entre eux **Alors**
    Renvoyer "Non".
  Fin Si
  Pour tout p, premier et divisant $q^d - 1$ **Faire** {Factorisation de $q^d - 1$}
    Si $A^{\frac{q^d-1}{p}} \equiv 1[F]$ **Alors**
      Renvoyer "Non".
    Fin Si{Calcul récursif par carrés}
  Fin Pour
  Renvoyer "Oui".

Un algorithme cherchant au hasard un générateur, une fois le corps construit, est alors aisé. En outre, V. Shoup [29, Théorème 1] a montré que l'on peut restreindre l'ensemble de recherche à des polynômes de faible de-

gré $((\log(n)))$. Toutefois, toujours par souci d'obtenir des polynômes creux, nous montrons à l'aide de la proposition suivante qu'il est possible de rapidement trouver, sur un corps premier, un polynôme irréductible pour lequel X est une racine primitive. Un tel polynôme est appelé **X-Irréductible**, ou **primitif**. En pratique, pour les corps finis de taille comprise entre 4 et 2^{32}, il est possible de montrer que plus d'un polynôme irréductible sur 12 est X-Irréductible ! Un algorithme recherchant un polynôme X-irréductible au hasard nécessite ainsi moins de $12r$ essais en moyenne. L'algorithme pour trouver un polynôme irréductible ayant X comme générateur est donc une simple modification de l'algorithme 5 ne selectionnant pas le polynôme irréductible trouvé si *Test-Polynôme-Générateur* répond que X n'est pas un générateur.

Exemple : Construction du corps $\mathbb{F}_{16}$.

1. *Détermination de P.*
   Le polynôme irréductible P s'écrit $P = X^4 + aX^3 + bX^2 + cX + 1$ avec a, b et c dans $\mathbb{Z}/2\mathbb{Z}$. Pour déterminer P, examinons les différentes valeurs possibles pour le triplet (a, b, c). On ne peut avoir $(a, b, c) \in \{(0, 1, 1), (1, 0, 1), (1, 1, 0), (0, 0, 0)\}$ puisque dans chacun de ces cas, 1 est racine de P. Le triplet (a, b, c) est donc à chercher dans $\{(0, 0, 1), (0, 1, 0), (1, 0, 0), (1, 1, 1)\}$. Les seuls polynômes irréductibles sur $\mathbb{F}_2$ de degré au plus égal à deux sont X, $1 + X$ et $X^2 + X + 1$. Pour chacun des quatre cas restants, il suffit de chercher le pgcd de P et $(1 + X)(X^2 + X + 1)$. Le calcul (par l'algorithme d'Euclide par exemple) de ces pgcd montre que les seules valeurs de (a, b, c) pour lesquelles P est irréductible sont $(0, 0, 1)$; $(1, 0, 0)$ et $(1, 1, 1)$. Donc, $P = X^4 + X^3 + 1$ est un choix possible de P. Faisons ce choix.

2. *Identification des éléments de $\mathbb{F}_{16}$ et règles pour les opérations.*
   Désignons par x la classe de X modulo P. Le calcul des puissances successives de x donne
   $x^1 = X \mod P$; $x^2 = X^2 \mod P$; $x^3 = X^3 \mod P$; $x^4 = 1 + X^3 \mod P$; $x^5 = 1 + X + X^3 \mod P$; $x^6 = 1 + X + X^2 + X^3 \mod P$; $x^7 = 1 + X + X^2 \mod P$; $x^8 = X + X^2 + X^3 \mod P$; $x^9 = 1 + X^2 \mod P$; $x^{10} = X + X^3 \mod P$; $x^{11} = 1 + X^2 + X^3 \mod P$; $x^{12} = 1 + X \mod P$; $x^{13} = X + X^2 \mod P$; $x^{14} = X^2 + X^3 \mod P$; $x^{15} = 1 \mod P$.
   Il en résulte que x est un élément générateur du groupe multiplicatif $\mathbb{F}_{16}$. Donc :
   $\mathbb{F}_{16} = \{0, 1, x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^{10}, x^{11}, x^{12}, x^{13}, x^{14}\}$.
   Avec $\mathbb{F}_{16}$ donné sous cette forme, la multiplication et le calcul de l'inverse dans $\mathbb{F}_{16}$ se font plus aisément. L'addition s'en trouve aussi simplifiée si l'on tient compte du fait que $x^k + x^t = x^t(1 + x^{k-t})$ pour tout

k et t dans $\{1, \ldots, 14\}$ tels que $k > t$.

1.3.7 Isomorphisme entre $V[X]/(P)$ et $V^{\deg(P)}$.

Soit V un corps fini, P un polynôme dans $V[X]$ de degré d et $V[X]/(P)$ l'anneau quotient associé.

A tout vecteur $u = [u_0, \ldots, u_{d-1}]$ du $V-$espace vectoriel V^d, on peut associer de manière bijective le polynôme $\psi(u) = \sum_{i=0}^{d-1} u_i X^i$ et en plus, on a la propriété suivante :

$$\forall u, v \in V^d, \forall \lambda \in V : \psi(u + \lambda \cdot v) = \psi(u) + \lambda \cdot \psi(v) \quad .$$

Ainsi, ψ est un isomorphisme entre V^d et $\psi(V^d) = V[X]/(P)$.

Ceci confère à V^d une structure d'anneau dans laquelle la multiplication est définie, comme en (1.4) par :

$$\forall u, v \in V^d \ : \quad u \cdot v = \psi^{-1}(\psi(u) \cdot \psi(v)) \ .$$

Si P est irréductible, alors $V[X]/(P)$ est un corps et il est en de même pour V^d.

1.3.8 Factorisation de $X^n - 1$ dans $\mathbb{F}_q[X]$: classes cyclotomiques

Soit p un nombre premier, d un entier positif et $\mathbb{F}_q$ le corps fini de cardinal $q = p^d$. Les facteurs du polynôme $X^n - 1$ dans $\mathbb{F}_q[X]$ jouent un rôle important pour les codes correcteurs cycliques . Ce paragraphe est consacré à la caractérisation de ces facteurs.

Réduction au cas essentiel : n premier avec q.
On suppose que n n'est pas premier avec $q = p^d$. On a alors $n = n'.p^\mu$ avec n' premier avec p et donc q. Dans $\mathbb{F}_q$, on a pour tout entier positif k, l'idendité remarquable

$$(a + b)^{kp} = a^{kp} + b^{kp} \ .$$

En effet, $(a+b)^{kp}$ s'écrit : $(a+b)^{kp} = \sum_{j=0}^{kp} C_{kp}^j a^j + b^{kp-j}$. Or C_{kp}^j est multiple de p sauf pour $j = 0$ et $j = kp$. Comme $\mathbb{F}_q$ est de caractéristique p, $p.x = 0$. Les seuls termes non nuls dans le développement sont donc a^{kp} et b^{kp}.

On a alors $X^n - 1 = X^{n'p^\mu} - 1 = (X^{n'} - 1)^{p^\mu}$. Trouver les facteurs de $X^n - 1$ dans $\mathbb{F}_q$ se ramène donc à trouver ceux de $X^{n'} - 1$ dans le cas où n' est premier avec q ; ce qui nous ramène au cas essentiel.

Factorisation dans le cas essentiel.
Dans ce cas (n premier avec $q = p^d$), il existe une racine primitive n-ième de l'unité α qui vérifie :

$$X^n - 1 = \prod_{j=0}^{n-1} (X - \alpha^j)$$

En général, α n'appartient pas à $\mathbb{F}_q$ mais à une extension algébrique de ce corps. Tout diviseur de $X^n - 1$ s'écrit donc sous la forme $\prod_{j\in\Sigma}(X - \alpha^j)$ où Σ est une partie de $\{0, \ldots, n-1\}$. Le théorème suivant caractérise les diviseurs de $X^n - 1$ qui sont dans $\mathbb{F}_q[X]$, i.e. dont les coefficients sont dans $\mathbb{F}_q$.

Théorème 9. *Soit Σ une partie non vide de $\{0, \ldots, n-1\}$. Alors $g_\Sigma = \prod_{j\in\Sigma}(X - \alpha^j)$ est un diviseur de $X^n - 1$.*
De plus, $g_\Sigma \in \mathbb{F}_q[X]$ si et seulement si Σ est stable par multiplication par q modulo n.

Les parties Σ stables par multiplication par q et minimales sont de la forme

$$\Sigma_j = \left\{j,\ jq,\ jq^2,\ \ldots,\ jq^{s-1}\right\}$$

où s est le plus petit entier tel que $(jq^s) \bmod n = j$. L'ensemble Σ_j est appelée *classe cyclotomique* de j relative à q modulo n.
Tout diviseur élémentaire de $X^n - 1$ dans $\mathbb{F}_q[X]$ est associé à une telle classe cyclotomique. Le calcul des classes cyclotomiques permet alors de décomposer facilement $X^n - 1$ dans $\mathbb{F}_q[X]$ en facteurs irréductibles à partir d'une racine primitive α donnée.
Exemple : Reprenons l'exemple de la décomposition de $X^3 - 1$ dans $\mathbb{F}_2[X]$. Soit α une racine primitive de $X^3 - 1$ (dans une extension algébrique). Les classes cyclotomiques et les polynômes irréductibles associés sont alors :

- pour $i = 0$, on a $\Sigma_0 = \{0\}$ et $g_{\Sigma_0} = X - \alpha^0 = X - 1$.
- pour $i = 1$, on a $\Sigma_1 = \{1, 2\}$ et $g_{\Sigma_1} = (X - \alpha^1)(X - \alpha^2)$.

Chapitre 2

Texte et codage.

Ce chapitre donne les définitions de base et quelques propriétés utiles à la manipulation des codes, le déchiffrage et une introduction à l'efficacité des codes pour la compression.

2.1 Définitions et exemple fondamental.

2.1.1 Définitions

- Un *alphabet* V est un ensemble fini $\{v_1, \ldots, v_k\}$ de k éléments appelés *caractères*.
- Un *code* C sur un alphabet V est un sous-ensemble fini de V^+ (ensemble des chaînes sur V de longueur non nulle).
- Un élément c_i de C est appelé *mot de code*. Sa longueur est notée $l(c_i)$.
- L'*arité* du code est le cardinal de V.

Exemple : $C = \{0, 10, 110\}$ est un code d'arité 2 (on dit aussi binaire) sur l'alphabet $V = \{0, 1\}$.

Soit C un code sur un alphabet V et soit $S = \{s_1, \ldots, s_q\}$ un ensemble fini tel que $|S| = |C|$. On appelle *fonction d'encodage* de S dans C toute bijection de S dans C. Pour une telle bijection $f : S \to C$ donnée, l'ensemble S est appelé *alphabet source* ; et les chaînes sur S sont appelées *messages*. Le couple (C, f) est alors appelé *schéma de codage* pour S.

Exemple : Le code *ASCII*.
L'ordinateur ne travaille qu'avec des données sous forme numérique. Pour la représentation numérique des caractères d'un clavier étendu, il existe un schéma de codage standard appelé "*ASCII*" (American Standart Code for Information Interchange). Dans le schéma de codage ASCII, le code C est constitué par l'ensemble des mots de longueur 8 sur l'alphabet
$V = \{0, 1\}$; autrement dit :

$$C = \{00000000,\ 00000001,\ \ldots,\ 11111111\}\ .$$

Chacun des $2^8 = 256$ caractères (majuscules, minuscules, caractères spéciaux, caractères de contrôle) est représenté par un mot de longueur 8 sur V suivant une fonction de codage dont un extrait est donné par la table 2.1 ci-dessous.

A	01000001	J	01001010	S	01010011
B	01000010	K	01001011	T	01010100
C	01000011	L	01001100	U	01010101
D	01000100	M	01001101	V	01010110
E	01000101	N	01001110	W	01010111
F	01000110	O	01001111	X	01011000
G	01000111	P	01010000	Y	01011001
H	01001000	Q	01010001	Z	01011010
I	01001001	R	01010010	espace	00100000

TAB. 2.1 – Un extrait de schéma de codage ASCII.

Par exemple, le codage suivant ASCII du message : *UNE CLEF*
est la chaîne :
0101010101001110010001010010000001000011010011000100010101000110 .

Dans tout ce cours, dans un souci de simplicité et de par leur importance pratique en télécommunications, on s'intéresse plus particulièrement aux codes binaires. Cependant, la plupart des résultats sont généralisables à des codes d'arité quelconque.

2.2 Déchiffrabilité d'un code.

Pour communiquer un message $a_1...a_n$ via un canal permettant de transmettre les mots d'un code C, on le traduit sous la forme $c_1...c_n = f(a_1)...f(a_n)$. Le fait que f soit bijective ne suffit cependant pas pour que le message puisse être décodé sans ambiguïté par le récepteur.

Prenons l'exemple du codage des lettres de l'alphabet $S = \{A, \ldots, Z\}$ par les entiers $C = \{0, \ldots, 25\}$ écrits en base 10 :

$$f(A) = 0, f(B) = 1, \ldots, f(J) = 9, f(K) = 10, f(L) = 11, \ldots, f(Z) = 25\,.$$

Le mot de code 1209 peut alors correspondre à différents messages : par exemple, BUJ ou MAJ ou BCAJ.

Il est donc nécessaire d'ajouter des contraintes sur le code pour qu'un message quelconque puisse être déchiffré sans ambiguïté.

2.2.1 Code uniquement déchiffrable.

Un code C sur un alphabet V est dit *uniquement déchiffrable* (on dit parfois *non ambigü*) si et seulement si, pour tout $x = x_1 \dots x_n \in V^+$, il existe au plus une séquence $c = c_1 \dots c_m \in C^+$ telle que

$$c_1 \dots c_m = x_1 \dots x_n \,.$$

Propriété 1. *Un code C sur un alphabet V est uniquement déchiffrable si et seulement si pour toutes séquences $c = c_1 \dots c_n$ et $d = d_1 \dots d_m$ de C^+ :*

$$c = d \Longrightarrow (n = m \text{ et } c_i = d_i \text{ pour tout } i = 1, \dots, n) \,.$$

Exemple : Sur l'alphabet $V = \{0, 1\}$,
- le code $C = \{0, 01, 001\}$ n'est pas uniquement déchiffrable.
- le code $C = \{01, 10\}$ est uniquement déchiffrable.
- le code $C = \{0, 10, 110\}$ est uniquement déchiffrable.

Théorème 10 (Kraft). *Il existe un code uniquement déchiffrable sur un alphabet V dont les mots $\{c_1, \dots, c_n\}$ sont de longueur $l_1, \dots, l_n$ si et seulement si*

$$\sum_{i=1}^{n} \frac{1}{|V|^{l_i}} \leq 1 \,.$$

Preuve. 1/ ($\Rightarrow$)

Soit C un code uniquement déchiffrable, d'arité q.

Pour $1 \leq k \leq m = max(longueurs\ des\ mots\ de\ C)$, soit r_k le nombre de mots de longueur k.

Nous développons l'expression suivante, pour un entier quelconque u, avec $u \geq 1$:

$$(\sum_{i=1}^{n} \frac{1}{q^{l_i}})^u = (\sum_{k=1}^{m} \frac{r_k}{q^k})^u \,.$$

Chaque terme est de la forme

$$\frac{r_{i_1} \dots r_{i_u}}{q^{i_1 + \dots + i_u}} \,.$$

Et en regroupant pour chaque valeur $s = i_1 + \dots + i_u$, on obtient les termes

$$\sum_{i_1 + \dots + i_u = s} \frac{r_{i_1} \dots r_{i_u}}{q^s} \,.$$

Soit $N(s) = \sum_{i_1 + \dots + i_u = s} r_{i_1} \dots r_{i_u}$. L'expression initiale s'écrit :

$$(\sum_{i=1}^{n} \frac{1}{q^{l_i}})^u = \sum_{s=u}^{mu} \frac{N(s)}{q^s} \,,$$

N(s) est le nombre de combinaisons de mots de C de longueur totale s. Comme C est uniquement déchiffrable, deux combinaisons de mots de C,ne peuvent être égales au même mot sur le alphabet de C.Comme C est d'arité q, et que $N(s)$ est inférieur au nombre total de messages de longueur s sur cet alphabet, alors $N(s) \leq q^s$. Donc

$$(\sum_{i=1}^{n} \frac{1}{q^{l_i}})^u \leq mu - u + 1 \leq mu\,.$$

Et en prenant la u-ième racine,

$$\sum_{i=1}^{n} \frac{1}{q^{l_i}} \leq (mu)^{1/u}\,.$$

Si u tend vers l'infini,

$$\sum_{i=1}^{n} \frac{1}{q^{l_i}} \leq 1\,.$$

D'où le résultat, puisque $q = |V|$.
2/ ($\Leftarrow$)

Ce résultat est une conséquence du théorème de McMillan, que nous verrons plus loin dans ce chapitre. □

2.2.2 Propriété du préfixe

On dit qu'un code C sur un alphabet V a la *propriété du préfixe* (on dit parfois qu'il est *instantané*, ou *irréductible*) si et seulement si pour tout couple de mots de code distincts (c_1, c_2), c_2 n'est pas un préfixe de c_1.

Exemple : $a = 101000$, $b = 01$, $c = 1010$: b n'est pas un préfixe de a mais c est un préfixe de a.

Grâce à la propriété du préfixe, on peut déchiffrer les mots d'un tel code dès la fin de la réception du mot (instantanéité), ce qui n'est pas toujours le cas pour les codes uniquement déchiffrables : par exemple, si $V = 0, 01, 11$, et si on reçoit le message $m = 001111111111\ldots$, il faut attendre l'occurence suivante d'un 0 pour pouvoir déchiffrer le second mot (0 ou 01 ?).

Propriété 2. *Tout code possédant la propriété du préfixe est uniquement déchiffrable.*

Preuve. Soit un code C sur V qui n'est pas uniquement déchiffrable. Alors il existe une chaîne $a \in V^n$ telle que $a = c_1 \ldots c_l = d_1 \ldots d_k$, les c_i et d_i étant des mots de C et $c_i \neq d_i$ pour au moins un i. Choisissons le plus petit i tel que $c_i \neq d_i$ ($\forall j < i$, $c_j = d_j$). Alors $l(c_i) \neq l(d_i)$, sinon, vu le choix

de i, on aurait $c_i = d_i$, ce qui est en contradiction avec la définition de i. Si $l(c_i) < l(d_i)$, c_i est un préfixe de d_i et dans le cas contraire d_i est un préfixe de c_i. C n'a donc pas la propriété du préfixe. □

La réciproque est fausse : le code $C = \{0, 01\}$ est uniquement déchiffrable, mais ne possède pas la propriété du préfixe.

Propriété 3. *Tout code dont tous les mots sont de même longueur possède la propriété du préfixe.*

2.2.3 L'arbre de Huffman.

On donne ici les définitions dans le cas binaire, mais elles sont généralisables pour des codes d'arité quelconque.

On appelle un *arbre de Huffman* un arbre binaire tel que tout sous-arbre a soit 0 soit 2 fils (il est localement complet). Dans ce dernier cas, on assigne le symbole "1" à l'arête reliant la racine locale au fils gauche et "0" au fils droit.

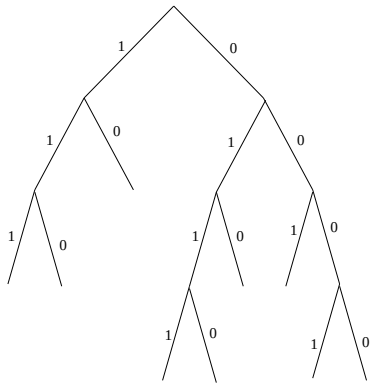

FIG. 2.1 – Exemple d'arbre de Huffman

A chaque feuille d'un arbre de Huffman, on peut associer un mot de $\{0,1\}^+$: c'est la chaîne des symboles marquant les arêtes d'un chemin depuis la racine jusqu'à la feuille. Le maximum de l'ensemble des longueurs des mots d'un arbre de Huffman est appelé *hauteur* de cet arbre On appelle *code de Huffman* l'ensemble des mots correspondant aux chemins d'un arbre de Huffman ; la hauteur de cet arbre est appelée aussi hauteur du code C.

Exemple : le code correspondant à l'arbre de la figure 1.1 est :

$$\{111, 110, 10, 0111, 0110, 010, 001, 0001, 0000\}.$$

2.2.4 Représentation des codes instantannés

Propriété 4. *Un code binaire de Huffman possède la propriété du préfixe.*

Preuve. Si un mot de code c_1 est un préfixe de c_2, alors le chemin représentant c_1 dans l'arbre de Huffman est inclus dans le chemin représentant c_2. Comme c_1 et c_2 sont, par définition, associés à des feuilles de l'arbre, $c_1 = c_2$. Il n'existe donc pas deux mots différents dont l'un est le préfixe de l'autre, et le code de huffman a la propriété du préfixe. □

Propriété 5. *Tout code qui possède la propriété du préfixe est contenu dans un code de Huffman.*

Preuve. On construit un arbre de Huffman à partir d'un code C quelconque possédant la propriété du préfixe :

Soit un arbre de Huffman complet (toutes les feuilles sont à distance constante de la racine) de hauteur l. Chaque mot c_i de C est associé à un chemin depuis la racine jusqu'à un noeud. On peut alors élaguer le sous-arbre dont ce noeud est racine (tous les mots pouvant être représentés dans les noeuds de ce sous-arbre ont c_i pour préfixe). Tous les mots de C sont toujours dans les noeuds de l'arbre résultant. On peut effectuer la même opération pour tous les mots. On a finalement un code de Huffman contenant tous les mots de C. □

2.2.5 Théorème de McMillan.

Théorème 11 (McMillan). *Sur un alphabet V, il existe un code qui possède la propriété du préfixe dont les mots $\{c_1, \ldots, c_n\}$ sont de longueur $l_1, \ldots, l_n$ si et seulement si*

$$\sum_{i=1}^{n} \frac{1}{|V|^{l_i}} \leq 1 \,.$$

Preuve. 1/ preuve de ($\Rightarrow$) : on suppose qu'il existe un code instantanné (c'est-à-dire qui vérifie la propriété du préfixe) avec les longueurs de mots $l_1, \ldots, l_n$. On a montré qu'on peut construire sa représentation par un arbre de Huffman. L'arbre complet initial a pour hauteur l, et $|V|^l$ feuilles. On compte à chaque opération d'élagage le nombre de feuilles de l'arbre initial complet qu'on enlève ainsi à l'arbre de Huffman. Pour un mot de longueur l_i, c'est le nombre de feuilles de l'arbre complet de hauteur $l - l_i$, soit $|V|^{l-l_i}$ (on suppose enlevée, car non réutilisée, une feuille d'un sous-arbre de hauteur 0). Pour toutes les opérations, on enlève $\sum_{i=1}^{n} |V|^{l-l_i}$ feuilles. Mais on ne peut en enlever au total plus que le nombre initial, soit :

$$\sum_{i=1}^{n} |V|^{l-l_i} \leq |V|^l$$

d'où :

$$\sum_{i=1}^{n} \frac{1}{|V|^{l_i}} \leq 1 \,.$$

2/ preuve de ($\Leftarrow$) : on sait que l'inégalité est satisfaite. On cherche à construire un arbre de Huffman dont les mots de code ont pour longueur $l_1, \ldots, l_n = l$, mots que l'on suppose classés par ordre croissant de leurs longueurs.

On part d'un arbre de hauteur l. On procède ensuite par induction.

Pour pouvoir choisir un noeud dans l'arbre, correspondant à un mot de longueur l_k, il faut qu'on puisse trouver dans l'arbre, un sous-arbre complet de hauteur $l - l_k$. Mais, toutes les opérations d'élagage précédentes consistent à enlever des sous-arbres de hauteur plus grande (correspondant à des mots plus courts). Donc, s'il reste au moins $|V|^{l-l_k}$ feuilles dans l'arbre restant, l'arbre restant contient un sous-arbre, de hauteur $l - l_k$. Montrons qu'il reste effectivement $|V|^{l-l_k}$ feuilles dans l'arbre restant.

Si on a déjà "placé" les mots de longeur $l_1, \ldots, l_{k-1}$, on a enlevé $\sum_{i=1}^{k-1} |V|^{l-l_i}$ feuilles de l'arbre initial par les successives opérations d'élagage. Il reste donc $|V|^l(1 - \sum_{i=1}^{k-1} |V|^{-l_i})$ feuilles. Or, on sait d'après l'inégalité de Kraft que :

$$\sum_{i=k}^{n} \frac{1}{|V|^{l_i}} \leq 1 - \sum_{i=1}^{k-1} \frac{1}{|V|^{l_i}}$$

soit :

$$|V|^l(1 - \sum_{i=1}^{k-1} |V|^{-l_i}) \geq \sum_{i=k}^{n} |V|^{l-l_i} \geq |V|^{l-l_k}$$

On peut donc placer le mot de longueur l_k, et en réitérant l'opération construire l'arbre de Huffman. Le code de Huffman, de longueurs de mots $l_1, \ldots, l_n$, vérifie la propriété du préfixe.

Remarque : Ceci prouve l'implication laissée en suspens du théorème de Kraft. □

Corollaire 3. *S'il existe un code uniquement déchiffrable dont les mots sont de longueur $l_1, \ldots, l_n$, alors il existe un code instantanné de mêmes longueurs de mots.*

C'est une conséquence des théorèmes de Kraft et McMillan. Les codes déchiffrables qui ne possèdent pas la propriété du préfixe ne produisent pas de code aux mots plus courts que les codes instantannés, auxquels on peut donc se restreindre pour la compression d'information (leurs propriétes les rendent plus maniables).

2.3 Code par blocs.

Historiquement, le codage n'était utilisé que pour garder le secret des messages ; les mots de codes consistaient principalement en des unités linguistiques. Les opérations utilisées pour coder se limitaient aux permutations et substitutions. L'apparition de nouveaux moyens de transports et de stockage de données (cables, satellites, ordinateurs) et la nécéssité d'adapter la représentation des données aux exigences des nouveaux supports (problème de lisibilité et d'implémentation), ont conduit au développement de la forme numérique pour les mots de code. L'alphabet V est alors composé de nombres et les mots de codes sont aussi des nombres.

Mieux, le codage et la détection d'erreur dans un code devient un travail fastudieux voir impossible si le code n'a pas une structure particulière. Pour bénéficier des possibilités de calcul qu'offrent les structures algébriques, on suppose généralement que les opérations arithmétiques d'addition $(+)$ et de multiplication $(\times)$ confèrent à alphabet V une structure d'anneau unitaire commutatif, voire même de corps commutatif.

Par exemple pour $V = \{0, 1\}$ et pour tout entier $n > 0$, si l'on pose $m = 2^n$, l'application $\phi : V^n \longrightarrow \mathbb{Z}/m\mathbb{Z}$ qui, à tout mot $c_0 \dots c_{n-1}$ de V^n associe l'entier naturel $c_0 + 2c_1 + \cdots + 2^{n-1}c_{n-1}$ est une bijection qui permet de munir V^n d'une structure d'anneau isomorphe à $\mathbb{Z}/m\mathbb{Z}$. Ce qui rend possibles les calculs modulaires sur les mots de tout code $C \subset V^n$.

Dans le même ordre d'idée, considérer V muni d'une structure de corps commutatif permet de considérer V^n (où $n > 0$ est un entier) comme un V-espace vectoriel et ainsi, on peut opérer sur les mots de tout code $C \subset V^n$ comme on opère sur des vecteurs d'un espace vectoriel.

Le fait de choisir l'alphabet V comme muni d'une structure de corps est d'un grand intérêt surtout lorsque le code C est choisi comme sous-ensemble de V^n ; c'est à dire un code dont tous les mots ont la même longueur. Un tel code est appelé *code par blocs*. Le code ASCII est un code binaire par blocs dans lequel chaque bloc est de longueur 8.

Une condition nécessaire et suffisante pour qu'on puisse définir une structure de corps sur V est (voir théorème 2 page 40) qu'il existe un nombre premier p et un entier $d > 0$ tels que $|V| = p^d$. Dans la pratique, si l'alphabet choisi au départ ne vérifie pas cette condition, on peut toujours lui ajouter (ou enlever) quelques éléments artificiellement de manière à ce que l'alphabet obtenu vérifie la condition exigée.

Chapitre 3

Théorie de l'information et compression.

Un premier intérêt de la théorie mathématique des codes réside dans l'optimisation possible de la taille d'un message qui doit transiter par un canal, ou être stocké sur un support. On effectue donc ce codage à la sortie de la source, et on décode à l'entrée de la destination. Nous supposons ici que le canal n'est pas soumis aux perturbations (on parle de codage sans bruit), le codage de canal et la gestion des erreurs sont étudiés au chapitre suivant. Il existe des techniques d'encodage permettant de choisir des codes efficaces ainsi qu'une théorie importante, originairement développée par Shannon en 1948 permettant de quantifier l'information contenue dans un message et de calculer la taille minimale d'un schéma de codage, et de connaître ainsi la "valeur" d'un code donné.

3.1 Source d'information

3.1.1 Source sans mémoire

Définition 9. *On définit une* source d'information *par un couple* $\mathcal{S} = (S, \mathcal{P})$ *où* $S = \{s_1, \ldots, s_n\}$ *est un alphabet source et* $\mathcal{P} = (p_1, \ldots, p_n)$ *est une distribution de probabilité. Pour tout* i, p_i *est la probabilité d'occurence de* s_i *dans une émission. .*

La source d'information $\mathcal{S} = (S, \mathcal{P})$ est dite *sans mémoire* lorsque les événements (occurences d'un symbole dans une émission) sont indépendants et que leur probabilité reste stable au cours de l'émission (la source est *stationnaire*).

Définition 10. *Une source* $\mathcal{S}$ *est dite* sans redondance *si tous les symboles de* S *apparaissent avec la même probabilité* ($p_1 = \ldots = p_n = \frac{1}{n}$).

Nous allons montrer dans ce chapitre qu'une source redondante peut être codée de manière à éliminer la redondance dans l'alphabet lisible par le canal. Nous précisons donc le schéma de codage défini en introduction de ce cours :

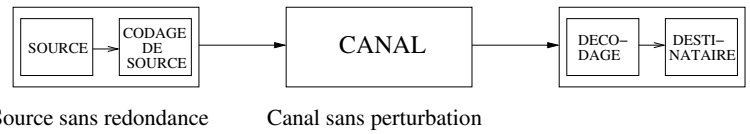

FIG. 3.1 – Codage de source

3.1.2 Longueur moyenne d'un code.

Définition 11. *Soient $\mathcal{S} = (S, \mathcal{P})$ une source, (C, f) un schéma de codage de S. La* longueur moyenne *de (C, f) est :*

$$l(C, f) = \sum_{i=1}^{n} l(f(s_i))P(s_i)$$

où $S = \{s_1, \ldots, s_n\}$.

Exemple : $S = \{a, b, c, d\}$, $\mathcal{P} = (\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$, $V = \{0, 1\}$.

Si $C = \{f(a) = 00, f(b) = 01, f(c) = 10, f(d) = 11\}$, la longueur moyenne du schéma est 2.

Si $C = \{f(a) = 0, f(b) = 10, f(c) = 110, f(d) = 1110\}$, la longueur moyenne du schéma est : $1 * \frac{1}{2} + 2 * \frac{1}{4} + 3 * \frac{1}{8} + 4 * \frac{1}{8} = 1.875$.

Nous utilisons la longueur moyenne d'un schéma de codage pour mesurer son efficacité.

3.1.3 Extension d'une source.

Définition 12. *Soit une source $\mathcal{S}$ sans mémoire. La* k-ième extension *$\mathcal{S}^k$ de $\mathcal{S}$ est le doublet $(S^k, \mathcal{P}^k)$, où S^k est l'ensemble des mots de longueur k sur S, et $\mathcal{P}^k$ est la distribution de probabilité ainsi définie : pour un mot $s = s_{i_1} \ldots s_{i_k} \in S^k$, $P^k(s) = P(s_{i_1} \ldots s_{i_k}) = P(s_{i_1}) \ldots P(s_{i_k})$.*

Exemple : $S = (s_1, s_2)$, $\mathcal{P} = (\frac{1}{4}, \frac{3}{4})$

alors $S^2 = (s_1s_1, s_1s_2, s_2s_1, s_2s_2)$ et $P^2 = (\frac{1}{16}, \frac{3}{16}, \frac{3}{16}, \frac{9}{16})$.

3.1.4 Limitations de la compression sans perte

Une compression est dite sans perte si la source est reconstituée parfaitement par le décodage, c'est-à-dire si absolument aucune différence n'existe entre la source et ce que le destinataire reçoit au final.

Par compression, on entend que le codage de la source doit prendre moins de place que la source elle-même. Dans ce chapitre, nous allons voir des algorithmes qui permettent de compresser des fichiers particuliers avec des gains de place importants dans certains cas. Il est important de voir que la compression dépend fortement de la source et que certaines sources ne pourront pas être compressées.

Un premier argument, pour comprendre la compression est l'argument de comptage suivant :

Théorème 12. *Aucun algorithme ne peut compresser sans perte TOUS les fichiers de taille N bits.*

Preuve. Supposons qu'un tel algorithme existe ; sur N bits, il existe 2^N fichiers distincts. L'algorithme doit fournir pour chacun de ces fichiers un résultat compressé, c'est-à-dire sur strictement moins de N bits. Combien y-a-t-il de tels fichiers ? 1 de 1 bit, 4 de 2 bits, 8 de 3 bits etc. Au total : $\sum_{i=1}^{n-1} 2^i = \frac{2^N-1}{2-1} = 2^N - 1$ fichiers. Cela veut dire qu'il y a forcément au moins un fichier codé exactement comme un autre, autrement dit il y a perte. □

Dans la suite, nous allons voir différents algorithmes et mesurerons leur efficacité. Avec la longueur moyenne d'un code, cette efficacité est totalement liée à une mesure de probabilité, c'est-à-dire qu'elle varie donc pour chaque fichier. L'argument précédent nous confirme bien que cette efficacité ne pourra être générale.

3.2 Algorithme de codage de Huffman

Cette méthode permet de trouver le meilleur schéma d'encodage d'une source sans mémoire $\mathcal{S}$.

3.2.1 Description de l'algorithme

La source à coder est $\mathcal{S} = (S, \mathcal{P})$, l'alphabet de codage V. Il est nécessaire à l'optimalité du résultat de vérifier que $|V| - 1$ divise $|S| - 1$ (afin d'obtenir un arbre localement complet). Dans le cas contraire, il est facile de rajouter des symboles à S, de probabilités d'occurence nulle, jusqu'à ce que $|V| - 1$ divise $|S| - 1$. Les mots de codes associés (les plus longs) ne seront pas utilisés.

On construit avec l'alphabet source S un ensemble de noeuds isolés auxquels on associe les probabilités de P.

Soient $p_{i_1}, \ldots, p_{i_{|V|}}$ les $|V|$ symboles de plus faibles probabilités. On construit un arbre (sur le modèle des arbres de Huffman), dont la racine est un nouveau noeud et auquel on associe la probabilité $p_{i_1} + \ldots + p_{i_{|V|}}$, et dont les branches sont incidentes aux noeuds $p_{i_1}, \ldots, p_{i_{|V|}}$. La figure 2.3 montre un exemple de cette opération pour $|V| = 2$.

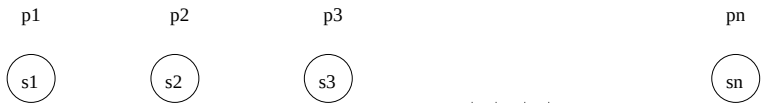

FIG. 3.2 – algorithme de Huffman : départ

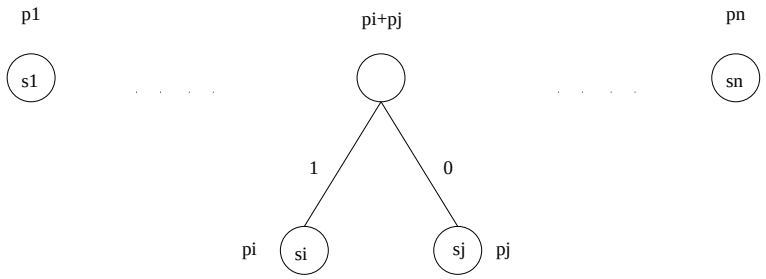

FIG. 3.3 – algorithme de Huffman : première étape ($| V |= 2$)

On recommence ensuite avec les $|V|$ plus petites valeurs parmi les noeuds du plus haut niveau (les racines), jusqu'à n'obtenir qu'un arbre (à chaque itération, il y a $|V| - 1$ éléments en moins parmi les noeuds de plus haut niveau), dont les mots de S sont les feuilles, et dont les mots de code associés dans le schéma ainsi construit sont les mots correspondant aux chemins de la racine aux feuilles.

Exemple : Soit la source à coder sur $V = \{0, 1\}$

Symbole	Probabilité
a	0,35
b	0,10
c	0,19
d	0,25
e	0,06
f	0,05

Les étapes successives de l'algorithme sont décrites par la figure 2.4.
Le code de Huffman construit est alors :

Symbole	Mot de code
a	11
b	010
c	00
d	10
e	0111
f	0110

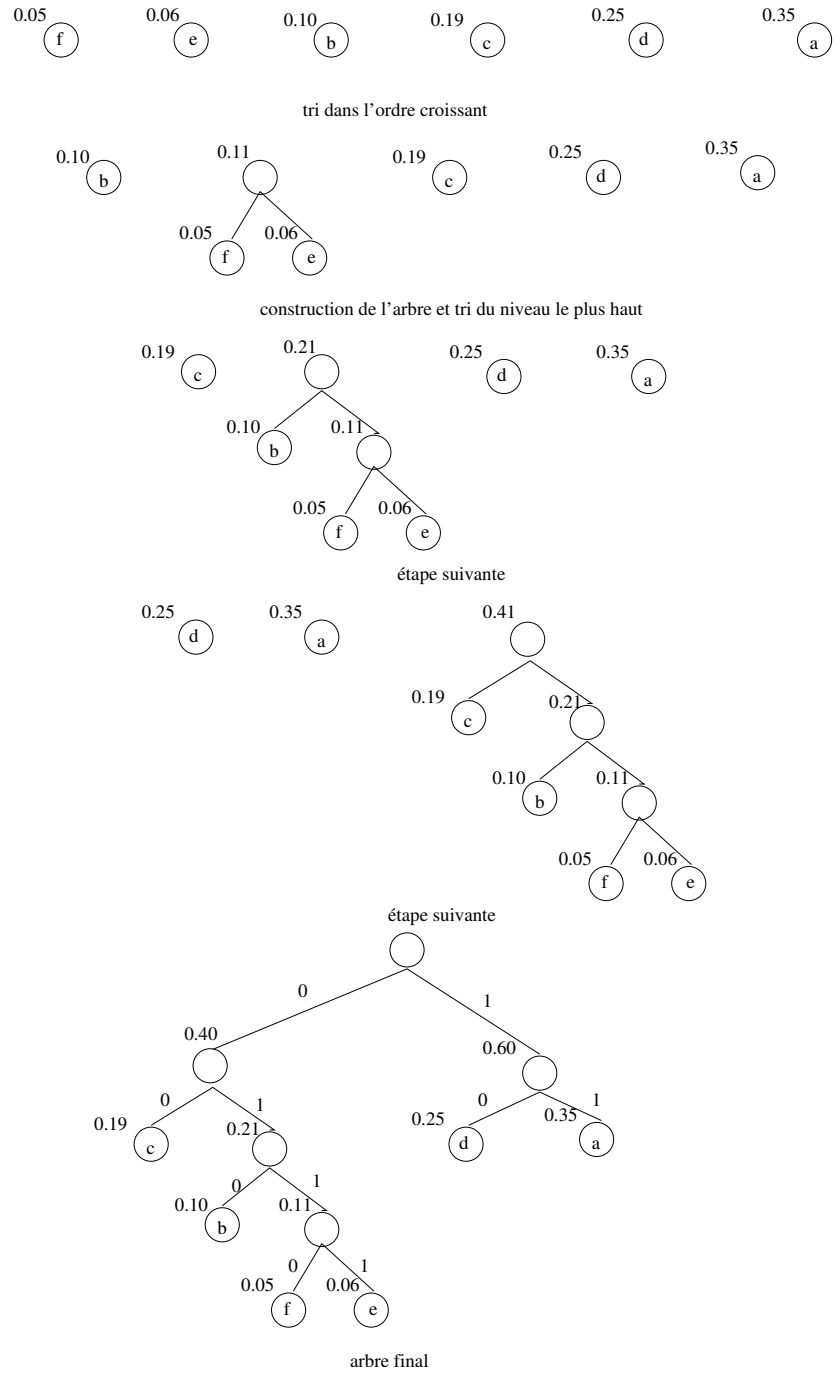

FIG. 3.4 – Exemple de construction d'un code de Huffman

3.2.2 L'algorithme de Huffman est optimal

Théorème 13. *Le code issu de l'algorithme de Huffman est optimal parmi tous les codes instantanés de $\mathcal{S}$ sur V.*

Preuve. On suppose dans la preuve pour alléger les notations que $|V| = 2$, mais on peut à toutes les étapes généraliser automatiquement.

On sait qu'un code instantané peut être représenté par un arbre de Huffman. Soit A l'arbre représentant un code optimal, et H l'arbre représentant le code issu de l'algorithme de Huffman.

Remarquons que dans A, il n'existe pas de noeud avec un seul fils dont les feuilles contiennent des mots du code (en effet, on remplace un tel noeud par son fils et on obtient un meilleur code).

Remarquons maintenant que dans A, si pour des mots c_1, c_2, les probabilités d'apparition respectives p_1, p_2 satisfont $p_1 < p_2$, alors les hauteurs respectives des feuilles représentant c_1, c_2 : l_1, l_2 satisfont $l_1 \geq l_2$ (en effet, dans le cas contraire, on remplace c_1 par c_2 dans l'arbre et on obtient un code meilleur).

On peut donc supposer que A représente un code optimal pour lequel les deux mots de plus faibles probabilités sont deux feuilles "soeurs" (elles ont le même père).

On raisonne maintenant par récurrence sur le nombre de feuilles n dans A. Pour $n = 1$, le résultat est évident. Pour $n \geq 2$ quelconque, on considère les deux feuilles soeurs correspondant aux mots c_1, c_2 de plus faibles probabilités d'apparition p_1, p_2 dans A. D'après le principe de construction de Huffman, c_1 et c_2 sont des feuilles soeurs dans H. On définit alors $H' = H \setminus \{c_1, c_2\}$. C'est un codage de Huffman pour le code $C' = C \setminus \{c_1, c_2\} \cup \{c\}$, c étant un mot de probabilité d'apparition $p_1 + p_2$. H' représente par le principe de récurrence le meilleur code instantané sur C', donc de longueur moyenne inférieure à $A' = A \setminus \{c_1, c_2\}$. Donc, et d'après les premières remarques, la longueur moyenne de H est inférieure à la longueur moyenne de A. □

Il faut bien comprendre la signification de ce théorème : il ne dit pas que l'algorithme de Huffman est le meilleur pour coder une information dans tous les cas ; mais qu'en fixant pour modèle une souce $\mathcal{S}$ sur un alphabet V, il n'y a pas de code plus efficace qui a la propriété du préfixe. Dans la pratique (voir à la fin de ce chapitre), les codes jouent sur le modèle en choisissant un modèle de source adapté. En particulier, on peut obtenir des codes plus efficaces à partir des extensions de la source, comme on peut le voir sur l'exemple suivant :

Soit $\mathcal{S} = (S, \mathcal{P})$, $S = (s_1, s_2)$, $\mathcal{P} = (1/4, 3/4)$. Un codage de Huffman pour $\mathcal{S}$ donne évidemment $s_1 \to 0$ et $s_2 \to 1$, et sa longueur moyenne est 1.

Un codage de Huffman pour $\mathcal{S}^2 = (S^2, \mathcal{P}^2)$ donne :

$$\begin{aligned} s_1 s_1 &\rightarrow 010 \\ s_1 s_2 &\rightarrow 011 \\ s_2 s_1 &\rightarrow 00 \\ s_2 s_2 &\rightarrow 1 \end{aligned}$$

et sa longueur moyenne est $3 * \frac{1}{16} + 3 * \frac{3}{16} + 2 * \frac{3}{16} + \frac{9}{16} = \frac{27}{16} = 1,6875$

La longueur moyenne de ce code est donc $l = 1,6875$, et en comparaison avec le code sur $\mathcal{S}$ (les mots de $\mathcal{S}^2$ sont de longueur 2), $l = 1,6875/2 = 0,84375$, ce qui est meilleur que le code sur la source originelle.

Nous pouvons encore améliorer ce codage en examinant la source $\mathcal{S}^3$. Il est aussi possible d'affiner le codage par une meilleure modélisation de la source : souvent, l'occurence d'un symbole n'est pas indépendante des symboles précédemment émis par une source (dans le cas d'un texte, par exemple). Dans ce cas, les probabilités d'occurence sont conditionnelles et il existe des modèles (le modèle de Markov, en particulier) qui permettent un meilleur codage. Mais ces procédés ne conduisent pas à des améliorations infinies. Il existe un seuil pour la longueur moyenne, appelé entropie, ne dépendant que de la source, en deçà duquel on ne peut pas trouver de code. Ce résultat théorique important fait l'objet de la section 3.3.

3.3 Entropie d'une source

3.3.1 Quantité d'information

Nous arrivons aux notions fondamentales de la théorie de l'information. Il est important, au-delà des définitions mathématiques de la quantité d'information et de l'entropie (que seul le théorème de Shannon pourrait justifier), de saisir la signification de ces grandeurs, et c'est pourquoi nous commençons par cette approche intuitive. Soit une source $\mathcal{S} = (S, \mathcal{P})$. Nous ne connaissons de cette source qu'une distribution de probabilité, mais nous cherchons à mesurer quantitativement à quel point nous ignorons le comportement de $\mathcal{S}$. Par exemple, cette *incertitude* est plus grande si le nombre de symboles dans S est plus grand. Elle est faible si une probabilité p_i est proche de 1, et plus forte en cas d'équiprobabilité.

Afin de choisir une fonction qui quantifie l'incertitude, nous cherchons une mesure de l'information I contenue dans un événement x (pour une source, c'est l'occurence d'un symbole s apparaissant avec une probabilité p). C'est une fonction croissante de l'*improbabilité* de cet événement.

Par exemple, imaginons une source pouvant émettre deux symboles, l'un avec une probabilité $p_1 = \frac{1}{100}$, l'autre avec une probabilité $p_2 = \frac{99}{100}$. L'apparition du premier symbole fournira au destinataire plus d'information sur la source (il ne lui manque que le second, qui apparaîtra avec une forte proba-

bilité) que l'apparition du second, qui laissera le destinataire pratiquement aussi peu informé sur la source qu'avant l'apparition du symbole.

Nous devons donc choisir $I(x)$ fonction inverse de la probabilité d'occurence de x. $I(x) = f(\frac{1}{P(x)})$. Soit $I(p) = f(\frac{1}{p})$. D'autres axiomes sont imposés par cette approche intuitive :
- la fonction quantité d'information $I(p)$ est positive et continue.
- l'information apportée par un événement certain est nulle.
- pour deux événements indépendants x, y, la quantité d'information apportée par x et y sera la somme des quantités d'information $I(x)$ et $I(y)$; puisque la probabilité d'occurence de deux événements est le produit des probabilités, et que la quantité d'information I est fonction inverse de la probabilité de x, I vérifie $I(x, y) = I(x) + I(y)$, soit $f(\frac{1}{P(x)P(y)}) = f(\frac{1}{P(x)}) + f(\frac{1}{P(y)})$.

Les seules fonctions satisfaisant ces axiomes sont :

$$I(x) = K * \log(\frac{1}{P(x)}) ,$$

ce qui est un résultat classique que nous ne redémontrons pas. Choisir une constante K revient à choisir une base pour le logarithme, ce qui est équivalent à choisir une unité de mesure. Par convention, on choisit le $\log_2$ et l'unité *bit* pour "binary unit", ou parfois *shannon*. Nous arrivons à la définition :

Définition 13. *Pour une source $\mathcal{S} = (S, \mathcal{P})$, La* quantité d'information *fournie par l'occurence d'un symbole $s \in S$ de probabilité $p \in]0, 1]$ est :*

$$I(p) = \log_2(\frac{1}{p}) .$$

3.3.2 Entropie

L'entropie d'une source est la quantité moyenne d'information contenue dans cette source.

Cette quantité peut-être illustrée par l'exemple suivant : sur un jet de dé non pipé, combien faut-il de questions pour déterminer quelle est la valeur obtenue ? En procédant par dichotomie, il suffit de $3 = \lceil log_2(6) \rceil$ questions. Supposons maintenant que le dé est pipé, le 1 sort une fois sur deux et les cinq autres une fois sur 10. Si la première question est : "est-ce 1 ?" dans la moitié des cas une question aura suffit, dans l'autre moitié il faudra 3 questions supplémentaires. Ainsi le nombre moyen de questions nécessaires sera $\frac{1}{2} * 1 + \frac{1}{2} * 4 = -\frac{1}{2} log_2(\frac{1}{2}) - 5 * \frac{1}{10} \lceil log_2(\frac{1}{10}) \rceil = 2.5$. En fait, il est encore possible d'affiner ce résultat en remarquant qu'il n'est pas toujours nécessaire de poser 3 questions pour déterminer lequel de $2, 3, 4, 5, 6$ a été tiré : si la dichotomie sépare ces 5 possibilités en deux groupes $2, 3$ et $4, 5, 6$, deux questions supplémentaires seulement seront nécessaires pour trouver 2 ou 3, et seuls 5 et 6 ne pourront être départagés qu'en trois questions. Sur un

grand nombre de tirage, il est encore possible de faire mieux si les questions ne séparent pas toujours de la même façon, par exemple en $2, 3, 5$ et $4, 6$, pour que deux questions soient alternativement nécessaires, etc. Poussé à l'infini, ce raisonnement nous induit que la moyenne des questions nécessaires pour les 5 possibilités est entre 2 et 3, et vaut donc $log_2(10)$!

La quantité moyenne d'information est ainsi définie intuitivement par le nombre moyen de questions ; la notion d'entropie suivante est donc naturelle :

Définition 14. *L'*entropie *d'une source* $\mathcal{S} = (S, \mathcal{P})$, $S = (s_1, \ldots, s_n)$, $\mathcal{P} = (p_1, \ldots, p_n)$ *est :*

$$H(\mathcal{S}) = H(p_1, \ldots, p_n) = -\sum_{i=1}^{n} p_i \log_2(p_i) \ = \sum_{i=1}^{n} p_i \log_2(\frac{1}{p_i}) \ .$$

C'est une mesure de *l'incertitude* liée à une loi de probabilités, ce qui est toujours illustré par l'exemple du dé : On considère la variable aléatoire (source) issue du jet d'un dé à n faces. Nous avons vu qu'il y a plus d'incertitude dans le résultat de ce jet si le dé est normal que si le dé est biaisé. Ce qui se traduit par $\forall p_1, \ldots, p_n$, $H(p_1, \ldots, p_n) \leq H(\frac{1}{n}, \ldots, \frac{1}{n}) = \log_2 n$.

En généralisant, cela nous donne :

Propriété 6. *Soit* $\mathcal{S} = (S, \mathcal{P})$ *une source, de loi* $(p_1, \ldots, p_n)$.

$$0 \leq H(\mathcal{S}) \leq \log_2 n \ .$$

Nous commençons par montrer un lemme dont nous aurons plusieurs fois besoin.

Lemme 2 (de Gibbs). *Soient* $(p_1, \ldots, p_n)$, $(q_1, \ldots, q_n)$ *deux lois de probabilité discrètes.*

$$\sum_{i=1}^{n} p_i * \log \frac{q_i}{p_i} \leq 0 \ .$$

Preuve. [du lemme] On sait que $\forall x \in \mathbb{R}_*^+$, $\ln(x) \leq x - 1$. Donc

$$\sum_{i=1}^{n} p_i * \ln \frac{q_i}{p_i} \leq \sum_{i=1}^{n} p_i * (\frac{q_i}{p_i} - 1) \ .$$

Soit puisque $\sum_{i=1}^{n} p_i = \sum_{i=1}^{n} q_i = 1$, alors

$$\sum_{i=1}^{n} p_i * \ln \frac{q_i}{p_i} \leq 0 \ .$$

Par suite,

$$\sum_{i=1}^{n} p_i * \log \frac{q_i}{p_i} \leq 0 \, .$$

□

Preuve. [du théorème] Appliquons ensuite ce lemme à la distribution $(q_1, \ldots, q_n) = (\frac{1}{n}, \ldots, \frac{1}{n})$, on obtient

$$H(\mathcal{S}) \leq \log_2 n \,.$$

Enfin, la positivité découle directement du fait que les probabilités p_i sont inférieures à 1. □

Propriété 7. *Soient $\mathcal{S}$ une source, et $\mathcal{S}^k$ sa k-ième extension.*

$$H(\mathcal{S}^k) = kH(\mathcal{S}) \,.$$

Preuve. On développe l'expression de $H(\mathcal{S}^k)$:

$$H(\mathcal{S}^k) = \sum_{i_1,\ldots,i_k, 0 \leq i_j \leq |V|} p_{i_1} \ldots p_{i_k} \log \frac{1}{p_{i_1} \ldots p_{i_k}} \,.$$

Soit

$$H(\mathcal{S}^k) = \sum_{i_1,\ldots,i_k} p_{i_1} \ldots p_{i_k} \log \frac{1}{p_{i_1}} + \ldots + \sum_{i_1,\ldots,i_k} p_{i_1} \ldots p_{i_k} \log \frac{1}{p_{i_k}} \,.$$

Le premier terme de cette somme est :

$$\sum_{i_1,\ldots,i_k} p_{i_1} \ldots p_{i_k} \log \frac{1}{p_{i_1}} = \sum_{i_1=1}^{|V|} p_{i_1} \log \frac{1}{p_{i_1}} * \sum_{i_2=1}^{|V|} p_{i_2} * \ldots * \sum_{i_k=1}^{|V|} p_{i_k}$$

Puisque la somme des probabilités p_j vaut toujours 1, alors

$$\sum_{i_1,\ldots,i_k} p_{i_1} \ldots p_{i_k} \log \frac{1}{p_{i_1}} = \sum_{i_1=1}^{|V|} p_{i_1} \log \frac{1}{p_{i_1}} = H(\mathcal{S}) \,.$$

Et en replaçant ce terme dans l'expression précédente de $H(\mathcal{S}^k)$, on obtient

$$H(\mathcal{S}^k) = H(\mathcal{S}) + \ldots + H(\mathcal{S}) = kH(\mathcal{S}) \,.$$

□

3.4 Théorème de Shannon

Ce théorème fondamental de la théorie de l'information est connu sous le nom de *théorème de Shannon* ou *théorème du codage sans bruit.*

Nous commençons par énoncer le théorème dans le cas d'une source sans mémoire .

Théorème 14. *Soit une source $\mathcal{S}$ sans mémoire d'entropie H.*
Tout code uniquement déchiffrable de $\mathcal{S}$ sur un alphabet V de taille Q (i.e $Q = |V|$), de longueur moyenne l, vérifie :

$$l \geq \frac{H}{\log_2 Q}\,.$$

De plus, il existe un code uniquement déchiffrable de $\mathcal{S}$ sur un alphabet de taille Q, de longueur moyenne l, qui vérifie :

$$l < \frac{H}{\log_2 Q} + 1\,.$$

Preuve.

première partie : Soit $C = (c_1, \ldots, c_n)$ un code de $\mathcal{S}$ uniquement déchiffrable sur un alphabet de taille Q, et $(l_1, \ldots, l_n)$ les longueurs des mots de C. Soit $K = \sum_{i=1}^{n} \frac{1}{Q^{l_i}}$, $K \leq 1$ d'après le théorème de Kraft (voir à la page 51). Soient $(q_1, \ldots, q_n)$ tels que $q_i = \frac{Q^{-l_i}}{K}$ pour tout $i = 1, \ldots, n$. On a $\forall i, q_i \in [0,1]$ et $\sum_{i=1}^{n} q_i = 1$, donc $(q_1, \ldots, q_n)$ est une distribution de probabilités. Le lemme de Gibbs (voir page 65) s'applique, et on obtient

$$\sum_{i=1}^{n} p_i \log \frac{Q^{-l_i}}{K p_i} \leq 0\,;$$

autrement dit

$$\sum_{i=1}^{n} p_i \log \frac{1}{p_i} \leq \sum_{i=1}^{n} p_i l_i \log Q + \log K\,.$$

Et comme $\log K \leq 0$, donc

$$H(\mathcal{S}) \leq l * \log Q\,.$$

D'où le résultat.

seconde partie : Soient $l_i = \lceil \log_Q \frac{1}{p_i} \rceil$. Comme $\sum_{i=1}^{n} \frac{1}{Q^{l_i}} \leq 1$ (car $Q^{l_i} \geq \frac{1}{p_i}$), il existe un code de $\mathcal{S}$ sur un alphabet de taille Q, uniquement déchiffrable, avec des longueurs de mots égales à $(l_1, \ldots, l_n)$. Comme $l_i < \lceil \log_Q \frac{1}{p_i} \rceil + 1$, $p_i < Q^{-l_i+1}$ et par suite,

$$\sum_{i=1}^{n} p_i \log \frac{1}{p_i} > \sum_{i=1}^{n} p_i l_i \log Q - \log Q\,.$$

Soit

$$l < \frac{H(\mathcal{S})}{\log Q} + 1\,.$$

□

On en déduit le théorème pour la k-ième extension de $\mathcal{S}$:

Théorème 15. *Soit une source $\mathcal{S}$ stationnaire d'entropie H.*

Tout code uniquement déchiffrable de $\mathcal{S}^k$ sur un alphabet de taille Q, de longueur moyenne l_k, vérifie :

$$\frac{l_k}{k} \geq \frac{H}{\log_2 Q}.$$

De plus, il existe un code uniquement déchiffrable de $\mathcal{S}^k$ sur un alphabet de taille Q, de longueur moyenne l_k, qui vérifie :

$$\frac{l_k}{k} < \frac{H}{\log_2 Q} + \frac{1}{k}.$$

Preuve. La preuve de ce théorème est immédiate d'après la propriété 7 selon laquelle $H(\mathcal{S}^k) = k * H(\mathcal{S})$. □

Pour une source stationnaire quelconque, le théorème peut s'énoncer :

Théorème 16. *Pour toute source stationnaire d'entropie H, il existe un procédé de codage uniquement déchiffrable sur un alphabet de taille Q, et de longueur moyenne l, aussi proche que l'on veut de sa borne inférieure $H/\log_2(Q)$.*

En théorie, il est donc possible de trouver un code s'approchant indéfiniment de l'entropie. En pratique pourtant, si le procédé de codage consiste à coder les mots d'une extension de la source, on est limité évidemment par le nombre de ces mots ($|\mathcal{S}^k| = |\mathcal{S}|^k$, ce qui peut représenter un très grand nombre de mots).

3.5 Codage arithmétique

Le codage arithmétique est une méthode statistique souvent meilleure que le codage de Huffman ! Or nous avons vu que Huffman était optimal, quelle optimisation est donc possible ? En codage arithmétique, chaque caractère peut être codé sur un nombre non entier de bits : des chaines entières de caractères plus ou moins longue sont encodées sur un seul entier ou réel machine . Par exemple, si un caractère apparaît avec une probabilité de 90% la taille optimale du codage de ce caractère pourrait être de $\frac{H}{\log_2 Q} = \frac{9/10\log_2(10/9)+1/10*\log_2(10/1)}{\log_2 10} \approx 0.14$ bits alors qu'un algorithme de type Huffman utiliserait sûrement 1 bit entier. Nous allons donc voir comment le codage arithmétique permet de faire mieux.

3.5.1 Arithmétique flottante

L'idée du codage arithmétique est d'encoder les caractères par intervalles. La sortie d'un code arithmétique est un simple réel entre 0 et 1 construit de la manière suivante : à chaque symbole on associe une portion de l'intervalle

Symbole	a	b	c	d	e	f
Probabilité	0,1	0,1	0,1	0,2	0,4	0,1
Intervalle	]0,0.1]	]0.1,0.2]	]0.2,0.3]	]0.3,0.5]	]0.5,0.9]	]0.9,1]

TAB. 3.1 – Intervalles de codage arithmétique

$[0, 1]$ de taille sa probabilité d'occurence. L'ordre d'association n'importe pas pourvu que soit le même pour le codage et le décodage.

Prenons l'exemple de la section 3.2.1 mais avec des probabilités plus simples sur la table 3.1. L'idée est alors de se placer à chaque caractère dans la portion de l'intervalle qui lui correspond et de continuer ensuite avec cette nouvelle portion comme intervalle de base. Sur l'exemple "bebecafdead" cela donne l'intervalle réel de la table 3.2 : tous les réels entre 0.15633504384

Symbole	Borne inférieure	Borne supérieure
b	0.1	0.2
e	0.15	0.19
b	0.154	0.158
e	0.1560	0.1576
c	0.15632	0.15648
a	0.156320	0.156336
f	0.1563344	0.1563360
d	0.15633488	0.15633520
e	0.156335040	0.156335168
a	0.156335040	0.1563350528
d	0.15633504384	0.15633504640

TAB. 3.2 – Codage arithmétique de "bebecafdead"

et 0.15633504640 correspondent au début de chaine "bebecafdead", choisissons par exemple "0.15633504500". Le programme de codage est donc très simple et peut s'écrire shématiquement sur l'algorithme 7. Pour le décodage, il suffit alors de repérer dans quel intervalle se trouve "0.15633504500" : c'est $]0.1, 0.2]$, donc la première lettre est un 'b'. Il faut ensuite se ramener au prochain intervalle, on retire la valeur inférieure et on divise par la taille de l'intervalle de 'b', à savoir $(0.0.15633504500 - 0.1)/0.1 = 0.5633504500$. Ce nouveau nombre nous indique que la valeur suivante est un 'e'. La suite du décodage est indiqué table 3.3, et le programme est décrit dans l'algorithme 8 :

En résumé, l'encodage réduit progressivement l'intervalle proportionnellement aux probabilités des caractères. Le décodage fait l'inverse en augmentant cet intervalle.

Algorithme 7 Codage arithmétique

```
binf = 0.0 ;
bsup = 1.0 ;
Tant que il y a des symboles à coder Faire
   C = symbole à coder ;
   intervalle = Table_Intervalles( C ) ;
   taille = bsup-binf ;
   bsup = binf + taille * intervalle.bsup ;
   binf = binf + taille * intervalle.binf ;
Fin Tant que
Retourner bsup ;
```

Réel	Intervalle	Symbole	Taille
0.15633504500	]0.1,0.2]	b	0,1
0.5633504500	]0.5,0.9]	e	0,4
0.158376125	]0.1,0.2]	b	0,1
0.58376125	]0.5,0.9]	e	0,4
0.209403125	]0.2,0.3]	c	0,1
0.09403125	]0.0,0.1]	a	0,1
0.9403125	]0.9,1.0]	f	0,1
0.403125	]0.3,0.5]	d	0.2
0.515625	]0.5,0.9]	e	0,4
0.0390625	]0.0,0.1]	a	0,1
0.390625	]0.3,0.5]	d	0.2

TAB. 3.3 – Décodage arithmétique de "0.156335"

Algorithme 8 Décodage arithmétique

```
r = nombre en entrée ;
Tant que r ≠ 0.0 Faire
   C = symbole dont l'intervalle contient r ;
   Afficher C ;
   intervalle = Table_Intervalles( C ) ;
   taille = intervalle.bsup-intervalle.binf ;
   r = r - intervalle.binf ;
   r = r / taille ;
Fin Tant que
```

3.5.2 Arithmétique entière

Le codage précéent est dépendat de la taille de la mantisse et n'est donc pas forcément très portable. Il n'est donc pas utilisé tel quel avec de l'arithmétique flottante. Une idée est de produire simplement les décimales une à une, puis de recombiner aux nombre de bits du mot machine. Une arithmétique entière plutôt que flottante est alors plus naturelle : au lieu d'un intervalle [0,1] flottant, c'est un intervalle du type [00000,99999] entier qui est utilisé, sachant que le nombre de 0 et de 9 est infini et donc que l'intervalle a en fait une taille ici de 100000. Ensuite, le codage est le même, par exemple 'b', donne [10000,19999], puis 'e' donnerait [15000,18999]. Il se trouve que dès que le chiffre le plus significatif est identique dans les deux bornes de l'intervalle, il ne change plus et peut donc être envoyé et supprimé du nombre entier : pour "be", cela donne [10000,19999], puis on affiche '1' que l'on retire et l'intervalle devient [00000,99999], ensuite passant à [50000,89999] pour 'e', la suite est dans le tableau 3.4. Le décodage suit quasiment la même pro-

Symbole	Borne inférieure	Borne supérieure	Sortie
b	10000	19999	
shift 1	00000	99999	1
e	50000	89999	
b	54000	57999	
shift 5	40000	79999	5
e	60000	75999	
c	63200	64799	
shift 6	32000	47999	6
a	32000	33599	
shift 3	20000	35999	3
f	34400	35999	
shift 3	44000	59999	3
d	48800	51999	
e	50400	51679	
shift 5	04000	16799	5
a	04000	05279	
shift 0	40000	52799	0
d	43840	46399	
			43840

TAB. 3.4 – Codage arithmétique entier de "bebecafdead"

cédure en ne lisant qu'un nombre fixé de bit à la fois :à chaque étape, on trouve l'intervalle (et donc le caractère) contenant l'entier courant. Ensuite, si le chiffre de poids fort est identique pour l'entier courant et les bornes, on décale le tout comme dans le tableau 3.5.

Shift	Entier	Borne inférieure	Borne supérieure	Sortie
	15633	$>$ 10000	$<$ 19999	b
1	56335	00000	99999	
	56335	$>$ 50000	$<$ 89999	e
	56335	$>$ 54000	$<$ 57999	b
5	63350	40000	79999	
	63350	$>$ 60000	$<$ 75999	e
	63350	$>$ 63200	$<$ 64799	c
6	33504	32000	47999	
	33504	$>$ 32000	$<$ 33599	a
3	35043	20000	35999	
	35043	$>$ 34400	$<$ 35999	f
3	50438	44000	59999	
	50438	$>$ 48800	$<$ 51999	d
	50438	$>$ 50400	$<$ 51679	e
5	04384	04000	16799	
	04384	$>$ 04000	$<$ 05279	a
0	43840	40000	52799	
	43840	$>$ 43840	$<$ 46399	d

TAB. 3.5 – Décodage arithmétique entier de "156335043840"

3.5.3 En pratique

En pratique, à cause de la précision machine, le schéma d'encodage de la section précédente ne fonctionne pas si les deux chiffres de poids fort ne deviennent pas égaux mais que la précision augmente. C'est le cas si l'on obtient un intervalle de type $[59992, 60007]$, après quelques itérations supplémentaires, l'intervalle converge vers $[59999, 60000]$, et plus rien n'est affiché ! Pour pallier ce problème, il faut, outre comparer les chiffres de poids fort, comparer également les suivants si les poids forts diffèrent seulement de 1. Alors, si les suivants sont 9 et 0, il faut décaler ceux-ci et conserver en mémoire que le décalage est au niveau du deuxième chiffre de poids fort. Par exemple, $[59123, 60456]$ est décalé en $[51230, 64569]_1$; l'indice 1 indiquant que l'on a décalé le deuxième chiffre. Ensuite, lorsque les chiffres de poids fort deviennent égaux (après k décalages de 0 et de 9), il faudra afficher ce chiffre suivi de k zéros ou k neufs (en décimal ou en hexa, il faudrait donc stocker un bit supplémentaire indiquant si l'on a convergé vers le haut ou le bas ; en binaire, cette information se déduit directement).

Exemple : Avec les probabilités $('a' : 4/10; 'b' : 2/10; 'c' : 4/10)$, le codage de la chaine "bbbbba" donne :

Symbole	Intervalle	Sortie
b	$[40000; 49999]$	
b	$[48000; 51999]$	
b	$[49600; 50399]$	
shift 9 et 0	$[46000; 53999]_1$	
b	$[49200; 50799]_1$	
shift 9 et 0	$[42000; 57999]_2$	
b	$[48400; 51599]_2$	
a	$[48400; 49679]_2$	
shift 4	$[84000; 96799]$	499
		84000

3.6 Heuristiques de réduction d'entropie

L'algorithme de Huffman (ou des variantes) est utilisé en pratique, souvent en même temps que d'autres procédés de codages, pas toujours optimaux en théorie, mais qui font des hypothèses raisonnables sur la forme des fichiers à compresser (ce sont des modéles de source) pour diminuer l'entropie, ou accepter une destruction d'information que l'on suppose sans conséquence pour l'utilisation des donnés.

Nous donnons ici trois exemples de tels précalculs.

3.6.1 Run Length Encoding

Le codage statistique tire parti des caractères apparaissant souvent, mais absolument pas de leur position dans le texte. Si un même caractère apparait souvent répété plusieurs fois d'affilée, il peut être utile de coder simplement le nombre de fois où il apparait. Le codage RLE, fait justement cela très simplement : il affiche chaque caractère à la suite, simplement lorsque 3 ou plus caractères identiques sont rencontrés, ces trois caractères sont affichés suivis d'un compteur indiquant le nombre d'apparitions supplémentaires du caractère. Il faut bien sûr convenir d'un nombre de bits fixe à attribuer à ce compteur. Si le compteur dépasse son maximum, la suite sera codée par plusieurs blocs des 3 mêmes caractères suivi d'un compteur. Par exemple, la chaine "aabbbcddddd" est codée par "aabbb0cddd2". Dans ce cas, si une chaine de n octets contenant m répetitions de longueur moyenne L, le gain de compression est $\frac{n-m(L-4)}{n}$ si le compteur est sur un octet. Ce type de codge est très utile par exemple pour des images noir et blanc ou des pixels de même couleur sont très souvents accolés. Ce type de codage est encore utilisé quasiment directement dans les modems ou les fax pour la transmission d'images noir et blanc (séries de nombres de pixels noirs et de nombre de pixels blancs). Enfin, il est clair qu'une compression statitisque peut être effectuée ultérieurement sur le résultat d'un RLE.

3.6.2 Move-to-Front

Un autre précalcul est possible lorsque que l'on transforme un caractère ASCII en sa valeur entre 0 et 255 : en modifiant à la volée l'ordre de la table. Par exemple, dès qu'un caractère est rencontré il passe en début de liste, et sera dorénavant codé par 0, tous les autres caractères étant décalés de une unité. Ce "Move-to-front" permet d'avoir plus de codes proches de 0 que de 255, quand on imagine que différentes occurences d'un même caractère sont souvent proches : ainsi, l'entropie est modifiée. Exemple : la chaine "aaaaffff" peut être codée par "00006666", d'entropie 1, si la table est (a,b,c,d,e,f) ou par "00006000", d'entropie $H = 7/8 \log_2(8/7) + log_2(8)/8 \approx 0.55$, réduite de moitié.

Une variante de cette méthode peut être de ne déplacer le caractère dans la liste que de k caractères vers le début. Ainsi, les caractères les plus fréquents vont se retrouver vers le début de la liste. Cette idée préfigure donc les codages adaptatifs et par dictionnaires des sections suivantes.

Exercice 1. *Soit A l'alphabet sur 8 symboles suivant : A = ('a', 'b', 'c', 'd', 'm', 'n', 'm', 'o', 'p').*

1. *On associe à chaque symbole un numéro. Quels nombres représentent "abcddcbamnopponm" et "abcdmnopabcdmnop" ?*
2. *Codez les deux chaines précédentes en utilisant la technique "Move-to-Front". Que constatez-vous ?*
3. *Combien de bits sont nécessaires pour coder les deux premiers nombres, par Huffman par exemple ?*
4. *Combien de bits sont nécessaires pour coder les deux nombres obtenus après "Move-to-Front" ? Comparer les entropies.*
5. *Que donne Huffman étendu à deux caractères sur le dernier nombre ? Quelle est alors la taille de la table des fréquences ?*
6. *Comment pourrait-on qualifier l'algorithme composé d'un "Move-to-Front" suivi d'un code statistique ?*
7. *Le "Move-ahead-k" est une variation du "Move-to-Front" où le caractère est seulement avancé de k positions au lieu du sommet de la pile. Encoder les deux chaines précédentes avec $k = 1$ puis $k = 2$ et comparer les entropies.*

3.6.3 Transformation de Burrows-Wheeler

L'idée de l'heuristique de Michael Burrows et David Wheeler est de trier les caractères d'une chaîne afin que le move-to-front et le RLE soient les plus efficaces possible. Le problème est bien sûr qu'il est impossible de retrouver la chaîne initiale à partir d'un tri de cette chaîne ! L'astuce est donc de trier la chaîne, mais d'envoyer plutôt une chaîne intermédiaire, de meilleure

entropie que la chaîne initiale, et qui permette cependant de retrouver la chaîne initiale. Prenons par exemple, la chaîne "*COMPRESSE*", il s'agit de créér tous les décalages possibles de la chaîne et de trier ceux-ci comme sur la figure 3.5. Bien sûr, pour calculer **L** (la première colonne) et **F** (la

												F		**L**
S_0	C	O	M	P	R	E_1	S_1	S_2	E_2		S_0	C		E_2
S_1	O	M	P	R	E	S	S	E	C		S_8	E_2		S_2
S_2	M	P	R	E	S	S	E	C	O		S_5	E_1		R
S_3	P	R	E	S	S	E	C	O	M	tri	S_2	M		O
S_4	R	E	S	S	E	C	O	M	P	$\longrightarrow$	S_1	O	...	C
S_5	E_1	S	S	E	C	O	M	P	R		S_3	P		M
S_6	S_1	S	E	C	O	M	P	R	E		S_4	R		P
S_7	S_2	E	C	O	M	P	R	E	S		S_7	S_2		S_1
S_8	E_2	C	O	M	P	R	E	S	S		S_6	S_1		E_1

FIG. 3.5 – BWT sur COMPRESSE

dernière colonne), il n'est pas nécessaire de stocker toute la matrice des décalages, un simple pointeur se déplaçant dans la chaîne est suffisant. La phase de transformation est donc assez simple. Mais comment effectuer la réciproque ?

La solution est alors d'envoyer la chaîne **L** au lieu de la chaîne **F** : si **L** n'est pas triée, elle est cependant un préfixe d'une chaîne triée et, sur un grand nombre de caractères, elle restera sans doute "triée par morceaux". La magie vient de ce que la connaissance de cette chaîne, conjuguée à celle de l'**index primaire** (le numéro de ligne contenant le premier caractère de la chaîne initiale, sur l'exemple 3.5, c'est 4 – en numérotant de 0 à 8) permet de récupérer la chaîne initiale. En effet, à partir de **L**, il suffit de trier pour récupérer **F** ! La deuxième étape de décompression est ensuite de calculer un vecteur de transformation **H** contenant la correspondance des indices entre **L** et **F** : c'est-à-dire l'indice dans **L** de chaque caractère pris dans l'ordre trié de **F**. Pour l'exemple 3.5, cela donne $H = \{4, 0, 8, 5, 3, 6, 2, 1, 7\}$, car C est en position 4, l'index primaire, dans **L**, puis E_2 est en position 0 dans **L**, puis E_1 est en position 8, etc.[1] Ensuite, il faut se rendre compte qu'en plaçant **L** avant **F** en colonnes, dans chaque ligne deux lettres se suivant doivent se suivre dans la chaîne initiale : en effet, par décalage, la dernière lettre devient la première et la première devient la deuxième. Ceci se traduit également par le fait que $\forall j, L[H[j]] = F[j]$. Il ne reste plus alors qu'a suivre cet enchaînement de lettres deux à deux pour retrouver la chaîne initiale :

En sortie de la transformation BWT, on obtient donc en général une

[1] sur la figure et dans le texte nous mettons des indices pour les caractères qui se répètent ; ces indices permettent de simplifier la vision de la transformation mais n'interviennent pas dans l'algorithme de décodage : en effet l'ordre des caractères est forcément conservé entre **L** et **F**.

Algorithme 9 Réciproque de la BWT

```
Récupérer L
Récupérer index_primaire
F = tri de L ;
Calculer le vecteur de transformation H tel que L[H[j]] = F[j]
int index = index_primaire ;
Pour i de 0 à Taille de L Faire
  Afficher L[index] ;
  index = H[index] ;
Fin Pour
```

chaîne d'entropie plus faible, bien adaptée à un move-to-front suivi d'un RLE. L'utilitaire de compression `bzip` détaillé sur la figure 3.6 utilise cette suite de réductions d'entropie avant d'effectuer un codage de Huffman. Nous verrons section 3.8.2 que cette technique est parmi les plus efficaces actuellement.

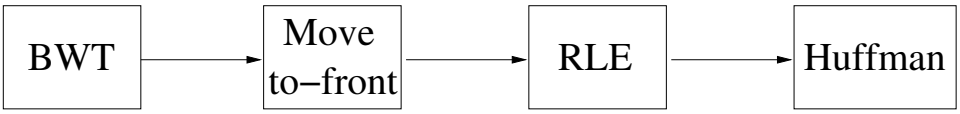

FIG. 3.6 – bzip

3.7 Codes adaptatifs

Le codage présenté par arbre de Huffman (avec paramétrage de l'extension de source) ou le codage arithmétique sont des codages statistiques qui nécessitent de connaître la fréquence d'apparition des caractères dans le fichier. Il existe des variantes dynamique (à la volée) qui permettent de s'affranchir de ce précalcul. Ce sont ces variantes qui sont les plus utilisées dans les utilitaires de compression.

3.7.1 Algorithme d'Huffman dynamique – `pack`

L'algorithme de Huffman dynamique permet de compresser un flot à la volée en faisant une seule lecture de l'entrée ; à la différence de l'algorithme statique d'Huffman il évite donc de faire deux parcours du fichier d'entrée (un pour le calcul des fréquences, l'autre pour le codage). La table des fréquences est élaborée au fur et à mesure de la lecture du fichier ; ainsi l'arbre de Huffman est modifié à chaque fois que l'on lit un nouveau caractère.

La commande "pack" de Unix implémente cet algorithme dynamique.

L'algorithme de Huffman dynamique est décrit dans la figure 10 ; le principe de la compression d'une part et de la décompression d'autre part est

Algorithme 10 Algorithmes de Huffman dynamique : compression et décompression.

Compression

```
int nbOccurences[256];
Arbre AH;
tant que (S != eof) {
    c = S.lireChar();
    si (nbOccurences[c] != 0) {
        envoyerCode(AH, c);
    }
    sinon {
        envoyerCode(AH, 'φ');
        nbOccurences[φ] += 1;
        envoyerOctet(c);
    }
    nbOccurences[c] += 1;
    maj(AH);
}
```

Décompression

```
int nbOccurences[256];
Arbre AH;
tant que (C_b != eof) {
    b = C_b.lireBits();
    char c = decoderCode(AH, b);
    si (c != φ) {
        envoyerChar(c);
    }
    sinon {
        nbOccurences[φ] += 1;
        C_b.lireOctet();
    }
    nbOccurences[c] += 1;
    maj(AH);
}
```

décrit ci-dessous.

Compression dynamique.

Le fichier est lu à la volée par bloc de k bits (k est souvent un paramètre) ; dans la suite, un tel bloc est appelé "caractère". A l'initialisation, on définit un caractère symbolique (noté ϕ dans la suite) et codé initialement par un symbole prédéfini (par exemple 0). Lors du codage, à chaque fois que l'on rencontre un nouveau caractère non encore rencontré, on le code sur la sortie par le code de ϕ suivi des k bits du nouveau caractère. Le nouveau caractère est alors entré dans l'arbre de Hufmann.
Pour construire l'arbre de Huffman et le mettre à jour, on compte le nombre d'occurences de chaque caractère et le nombre de caractères déja lus ; on connait donc, à chaque top, la fréquence de chaque caractère depuis le début du fichier jusqu'au caractère courant ; les fréquences sont donc calculées dynamiquement.
Après avoir écrit un code (soit celui de ϕ, soit celui d'un caractère déjà rencontré, soit les k bits non compressés d'un nouveau caractère), on incrémente de 1 le nombre d'occurences du caractère écrit. En prenant en compte les modifications de fréquence, on met à jour l'arbre de Huffman à chaque top de manière déterministe.
L'arbre existe donc pour la compression (et la décompression) mais n'a pas

besoin d'être envoyé au décodeur.
Enfin, il y a plusieurs choix pour le nombre d'occurences de ϕ : dans le code 10 c'est le nombre de caractères distincts (cela permet d'avoir peu de bits pour ϕ au début), il est aussi possible de lui attribuer par exemple une valeur constante très proche de zéro (dans ce cas le nombre de bits pour *phi* évolue comme la profondeur de l'arbre de Huffmann, en laissant aux caractères très fréquents les codes les plus courts).

Décompression dynamique.

A l'initialisation, le décodeur connaît un seul code, celui de ϕ (par exemple 0). Il lit alors 0 qui est le code associé à ϕ. Il en déduit que les k bits suivants contiennent un nouveau caractère. Il écrit ces k bits et met à jour l'arbre de Huffman, qui contient déja ϕ, avec ce nouveau caractère.
Le point critique est que le codeur et le décodeur maintiennent chacun leur propre arbre de Hufmann, mais utilisent tous les deux le même algorithme déterministe pour le mettre à jour à partir des occurences (fréquences) des caractères déjà lus. Ainsi, les arbres de Huffman calculés séparément par le codeur et le décodeur sont exactement les mêmes.
Puis, le décodeur lit le code suivant et le décode via son arbre de Huffman. Si il s'agit du code de ϕ, il lit les k bits correspondants à un nouveau caractère, les écrit sur la sortie et ajoute le nouveau caractère à son arbre de Huffman (le nouveau caractère est désormais associé à un code). Sinon, il s'agit du code d'un caractère déjà rencontré ; via son arbre de Huffman, il trouve les k bits du caractère associé au code, et les écrit sur la sortie. Il incrémente alors de 1 le nombre d'occurences du caractère qu'il vient d'écrire (et de ϕ si c'est un nouveau caractère) et met à jour l'arbre de Huffmann.

Au niveau estimation des fréquences, cette méthode dynamique est un peu moins efficace que la méthode statique. Cependant, elle évite le stockage de l'arbre (la table des fréquences) et le résultat final est souvent plus court.

3.7.2 Codage arithmétique adaptatif

L'idée générale du codage adaptatif, développée dans la sous section précédente pour l'algorithme de Huffmann,est la suivante :

- à chaque étape le code courant correspond au code statique que l'on aurait obtenu en utilisant les occurences déjà connues comme fréquences.
- après chaque étape, l'encodeur mets à jour sa table d'occurence avec le nouveau caractère qu'il a reçu et crée un nouveau code statique correspondant. Ceci doit être fait toujours de la même manière afin que le décodeur puisse faire de même à son tour.

Cette idée s'implémente très bien également dans le cadre du codage arithmétique : le code est construit essentiellement comme le code arithmétique, sauf que la distribution de probabilité est calculée au vol par l'enco-

deur sur les symboles qui ont déjà été traités. Le décodeur peut faire les mêmes calculs et reste ainsi synchronisé. Le codeur arithmétique adaptatif travaille comme suit en arithmétique flottante : à la première itération, l'intervalle $[0, 1[$ est divisé en n segments de longueurs égales. Chaque fois qu'un symbole est reçu, l'algorithme calcule la nouvelle distribution de probabilité et le segment du symbole reçu est lui-même re-divisé en n nouveaux segments. Cependant, ces nouveaux segments sont maintenant de longueurs correspondantes aux probabilités, mises à jour, des symboles. Plus le nombre de symboles reçus est grand, plus la probabilité calculée sera proche de la probabilité réelle. Comme pour Huffman dynamique, il n'y a aucun surcoût dû à l'envoi préliminaire de la table des fréquences et pour des sources variables l'encodeur est capable de s'adapter dynamiquement aux variations de probabilités.

Enfin, outre la compression souvent meilleure par un codage arithmétique, il est à noter que si les implémentations de Huffman statique sont plus rapide que celles du codage arithmétique statique, c'est en général l'inverse dans le cas adaptatif.

3.8 Codes compresseurs usuels

3.8.1 Algorithme de Lempel-Ziv et variantes `gzip`

L'algorithme de Lempel Ziv est un algorithme de compression par dictionnaire (ou encore par substitution de facteurs) ; il consiste à remplacer une séquence de caractères (appelée facteur) par un code plus court qui est l'indice de ce facteur dans un dictionnaire. Come pour l'algorithme d'Huffman dynamique, les méthodes par dictionnaire ne nécessitent qu'une lecture du fichier.

Il existe plusieurs variantes de cet algorithme ; parmi celles-ci, LZ77 et LZ78 sont libres de droits.

- LZ77 (publié par Lempel et Zif en 1977) est le premier algorithme de type dictionnaire. Alternative efficace aux algorithmes de Huffman, il a relancé les recherches en compression. LZ77 est basé sur une fenêtre coulissante qui coulisse sur le texte de gauche à droite. Cette fenêtre est divisée en deux parties : la première partie constitue le dictionnaire ; la seconde partie (tampon de lecture) rencontre le texte en premier. Initialement, la fenêtre est située de façon à ce que le tampon de lecture soit positionné sur le texte, et que le dictionnaire n'y soit pas. A chaque top, l'algorithme cherche, dans le dictionnaire le plus long facteur qui se répète au début du tampon de lecture ; ce facteur est codé par le triplet (i, j, c) où :
  - i est la distance entre le début du tampon et la position de la répétition dans le dictionnaire ;
  - j est la longueur de la répétition ;

- c est le premier caractère du tampon différent du caractère correspondant dans le dictionnaire.

Après avoir codé cette répétition, la fenêtre coulisse de $j+1$ caractères vers la droite. Dans le cas où aucune répétition n'est trouvée dans le dictionnaire, le caractère c qui a provoqué la différence est codé alors $(0, 0, c)$.

- LZ78 est une amélioration qui consiste à remplacer la fenêtre coulissante par un pointeur qui suit le texte et un dictionnaire indépendant, dans lequel on cherche les facteurs situés au niveau du pointeur. Supposons que, lors du codage, on lit la chaîne sc où s est une chaîne qui est à l'index n dans le dictionnaire et c est un cacactère tel que la chaîne sc n'est pas dans le dictionnaire. On écrit alors sur la sortie le couple (n, c). Puis, le caractère c concaténé au facteur numéro n nous donne un nouveau facteur qui est ajouté au dictionnaire pour être utilisé comme référence dans la suite du texte. Seules deux informations sont codées au lieu de trois avec LZ77.
- LZW (Lempel-Zip-Welch) est une autre variante de Lempel-Zif. LZW est breveté par Unisys. Il consiste à ne coder que l'indice n dans le dictionnaire ; de plus le fichier est lu bit par bit. Enfin, il nécessaire d'avoir un dictionnaire initial (par exemple, la table ASCII). La compression, indiquée à la figure 11 est alors directe. Seule la décompression est un plus technique, il faut traiter en outre le cas particulier où la même chaine est reproduite deux fois de suite (le prochain code à mettre dans le dictionnaire est alors justement celui que l'on rencontre et celui-ci est donc inconnu, mais le caractère de fin de la nouvelle chaîne est alors le même que celui du début de chaine). Enfin, deux variantes simples ont été proposées, LZMW où 'chaîne + prochain mot' est ajoutée au dictionnaire au lieu de seuelemnt 'chaîne + prochain caractère' et LZAP dans laquelle tous les préfixes de 'chaîne + prochain mot' sont ajoutés au dictionnaire.

Exercice 2. *En utilisant la table ASCII hexadécimale (7bits) ci dessous comme dictionnaire initial, compresser la chaîne "BLEBLBLA" avec LZW.*

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Décompresser le résultat sans le traitement du cas particulier, que constatez-vous ?

Algorithme 11 LZW : compression et décompression.

Compression

```
chaîne = S.lireChar() ;

Tant que (S != eof) {
    char c = S.lireChar() ;
    Si ('chaîne|c' ∈ Dico) {
        chaîne='chaîne|c' ;
    }
    Sinon {
        Afficher le code de chaîne
        Ajouter 'chaîne|c' au Dico
        chaîne = c ;
    }
}

Afficher le code de chaîne
```

Décompression

```
int old_i = C_b.lireBits()
char c = ASCII(old_i) ;
Afficher c
Tant que (C_b != eof) {
    int new_i = C_b.lireBits() ;
    Si (new_i ∈ Dico) ;
        chaîne = Dico(new_i) ;
    }
    Sinon { // cas particulier
        chaîne = Dico(old_i) ;
        chaîne = 'chaîne|c' ;
    }
    Afficher chaîne
    c = chaîne.firstchar() ;
    Ajouter 'Dico(old_i)|c' au Dico
    old_i = new_i ;
}
```

La commande `compress` de Unix implémente LZ77. L'utilitaire `gzip` est basé sur une variante ; il utilise deux arbres de Huffman dynamiques : un pour les chaînes de caractères, l'autre pour les distances entre occurences. Les distances entre deux occurences d'une chaîne sont bornées : lorsque la distance devient trop grande, on réinitialise la compression en redémarrant une compression à partir du caractère courant, indépendamment de la compression déjà effectuée jusqu'à ce caractère.

Cette particularité fait qu'il n'y a pas de différence structurelle entre un seul fichier compressé par `gzip` et une séquence de fichiers compressés. Ainsi, soient `f1.gz` et `f2.gz` les résultats associés à la compression de deux fichiers sources `f1` et `f2` ; supposons que l'on concatène[2] dans un fichier `f3.gz`. En décompressant avec `gunzip f3.gz`, on obtient en résultat un fichier dont le contenu est identique à la concaténation de `f1` et `f2`.

3.8.2 Comparaison des algorithmes de compression

Dans cette section, nous comprarons les taux de compression et les vitesses de codage et décodage des utilitaires classiques sous unix/linux. `gzip` et `compress` utilisent des variantes de Lempel-Ziv et sont décrits à la section précédente, `bzip2` utilise des réductions d'entropies comme la Transformation de Burrows-Wheeler avant de faire un codage entropique et est décrit

[2]Par exemple avec la commande Unix : `$ cat f1.gz f2.gz > f3.gz`.

section 3.6.3, `pack`, enfin, implémente un codage de Huffman simple. En

Algorithme	Fichier compressé	Taux	codage	décodage
bzip2 (BWT)	6.47 Mo	55.75%	14.79s	5.09s
gzip -9 (LZ77)	6.67 Mo	54.38%	6.97s	0.46s
gzip -5 (LZ77)	6.74 Mo	53.90%	3.08s	0.51s
compress (LZW)	9.19 Mo	37.14%	4.18s	0.64s
pack (Huffman)	11.05 Mo	24.42%	0.33s	

TAB. 3.6 – Compression d'un fichier de courriels (14.62Mo), sur un PIII 1GHz

conclusion le comportement pratique reflète bien la théorie : LZ77 compresse mieux que LZW, mais est plus lent. Les algorithmes par dictionnaires compressent mieux que les codages entropiques simples, mais moins bien qu'un codage entropique avec une bonne heuristique de réduction d'entropie. Enfin, il serait intéressant de poursuivre cette étude avec des implémentations de codage arithmétique.

3.8.3 Formats GIF et PNG pour la compression d'images

Le format GIF (Graphic Interchange Format) est un format de fichier graphique bitmap proposé par la société Compuserve. Le format GIF est un format de compression pour une image pixellisée c'est-à-dire décrite comme une suite de points (pixels) contenus dans un tableau ; chaque pixel a une valeur qui décrit sa couleur.

Le principe de compression est en deux étapes. Tout d'abord, les couleurs pour les pixels (initialement, il y a 16,8 millions de couleur codées sur 24 bits RGB) sont tout d'abord limitées à une palette contenant de 2 à 256 couleurs (2, 4, 8, 16, 32, 64, 128 ou 256 qui est le défaut). La couleur de chaque pixel est donc approchée par la couleur la plus proche figurant dans la palette. Tout en gardant un nombre important de couleurs différentes avec une palette de 256, ceci permet d'avoir un facteur 3 de compression. Puis, la séquence de couleurs des pixels est compressée par l'algorithme de compression dynamique LZW (Lempel-Zip-Welch). Il existe deux versions de ce format de fichier développées respectivement en 1987 et 1989 :

- GIF 87a permet un affichage progressif (par entrelacement) et la possibilité d'avoir des images animées (appelées GIFs animés) en stockant plusieurs images au sein du même fichier.
- GIF 89a permet en plus de définir une couleur transparente dans la palette et de préciser le délai pour les animations.

Comme l'algorithme de compression LZW est breveté par Unisys, tous les éditeurs de logiciel manipulant des images GIF doivent payer une redevance à Unisys. C'est une des raisons pour lesquelles le format PNG est de plus en

plus plébiscité, au détriment du format GIF. Le format PNG est un format analogue ; la différence est que l'algorithme de compression utilisé est LZ77 (une variante de Lempel-Ziv), qui lui est libre de droits.

3.8.4 Formats JPEG et MPEG

Le codage au format JPEG (Joint Photographic Experts Group) compresse des images fixes avec perte d'information. L'algorithme de codage est complexe, et se déroule en plusieurs étapes. Le principe de base est que les couleurs des pixels voisines dans une image diffèrent peu. On code donc le décalage par rapport à la pixel voisine. De plus, une image est un signal : plutôt que les valeurs des pixels, on calcule les fréquences (transformée de Fourier DFT, transformée en cosinus discrète DCT). En ne gardant que les premiers termes de la décomposition (les plus importants), on perd un peu d'information mais l'image reste visible. L'image est finalement codée comme une suite de nombres (une valeur de DCT suivie du nombre de pixels ayant cette valeur qui sont consécutives selon un balayage en zigzag de l'image). En fin d'algorithme, un codage de Hufmann compresse le fichier de nombres ainsi obtenus.

FIG. 3.7 – Compression JPEG

Le format MPEG (Motion Picture Experts Group) assure la compression d'images animées). L'algorithme de codage utilise JPEG pour coder une image mais prend en compte le fait que deux images consécutives dans une séquence vidéo sont très voisines. Une des particularités des normes MPEG est de chercher à compenser le mouvement (ex. zoom etc) d'une image à la suivante. Une sortie MPEG(-1) contient 4 sortes d'images : des images au format I (JPEG), des images P codées par différences avec l'image précédente (par compensation du mouvement deux images consécutives sont très voisines), des images bidirectionnelles B codées par différence avec l'image précédente et la suivante (en cas de différences moins importantes avec une image future qu'avec une image précédente), enfin des images basse résolution utilisées pour l'avance rapide sur un magnétoscope. En pratique, avec les images P et B, il suffit d'une nouvelle image complète I toutes les 10 à 15 images.

Un film vidéo étant composé d'images et de son, il faut également compresser le son le groupe MPEG a donc définit trois formats les MPEG-1 Audio Layer I, II et III, le troisème étant le fameux MP3. Nous allons y revenir. Le MPEG-1 est donc la combinaison de compressions d'images et de

son, associées à une synchronisation par marquage temporel et une horloge de référence du sytème comme indiqué sur les figures 3.8 et 3.9.

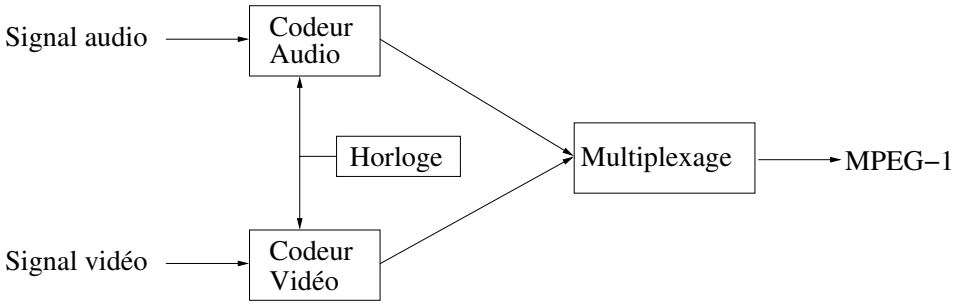

FIG. 3.8 – Compression MPEG-1

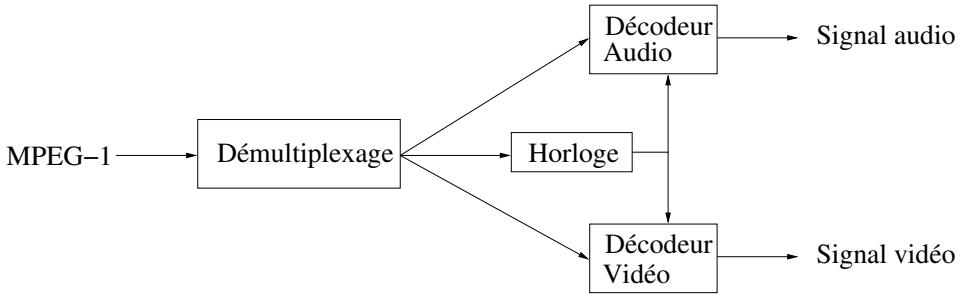

FIG. 3.9 – Décompression MPEG-1

Depuis 1992, le format MPEG à évolué de la façon suivante :

1992 MPEG-1 avec 320×240 pixels et un débit de 1.5 Mbits/s.

1996 MPEG-2 avec quatre résolutions de 352×288 à 1920×1152 pixels plus un flux de sous-titres et langues multiplexés est passé à un débit de 4 à 6 Mbits/s et est l'actuel format des DVD.

1999 MPEG-4 La vidéo est devenue une scène orientée objet, la compression est impressionante. C'est le format des vidéo conférences et du DivX.

2002 MPEG-7 XML a été ajouté pour permettre les recherches multimédia.

2004 MPEG-21 La sécurité est prévue.

Enfin, le son est donc décomposé en trois couches, MP1, MP2 et MP3, suivant les taux de compression (et donc les pertes) désirés. En pratique, les taux de compression par rapport à l'analogique sont d'un facteur 4 pour le MP1, de 6 à 8 pour le MP2, utilisé par exemple dans les vidéo CD et de 10 à 12 pour le MP3. Nous donnons une idée de la transformation MP3 sur

la figure 3.10 : le son analogique est d'abord filtré et modifié (avec pertes) par des transformées de Fourier et en cosinus (FFT, DCT) puis, après un contrôle, compressé avec un algorithme de Huffman classique.

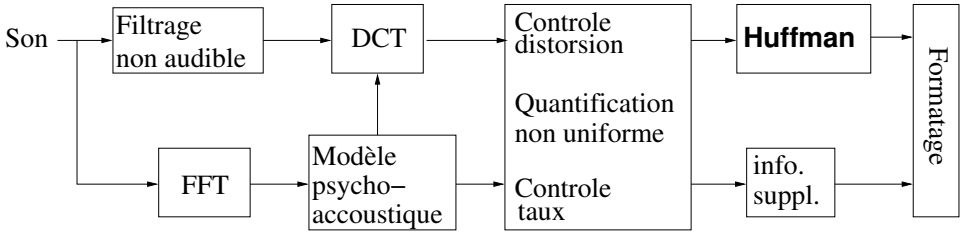

FIG. 3.10 – Compression du son MP3

Chapitre 4

Cryptographie.

Le chapitre 2 a permis de définir la notion de codage. Il s'agit maintenant d'étendre cette notion au cas où on souhaite cacher une information.

4.1 Terminologie

L'objectif fondamental de la cryptographie est de permettre à deux personnes, appelées traditionnellement Alice et Bob, de communiquer à travers un canal peu sûr de telle sorte qu'un opposant, Oscar, ne puisse pas comprendre ce qui est échangé. Le canal peut être par exemple une ligne téléphonique ou tout autre réseau de communication.

L'information qu'Alice souhaite transmettre à Bob est appelé *texte clair*. Il peut s'agir d'un texte en français, d'une donnée numérique ou de n'importe quoi d'autre, de structure arbitraire.

Le processus de transformation d'un message M de tel manière à le rendre incompréhensible est appelé *chiffrement*.

On génère ainsi un *message chiffré* C obtenu à partir d'une *fonction de chiffrement* E par $C = E(M)$. Le processus de reconstruction du message clair à partir du message chiffré est appelé *déchiffrement* et utilise une *fonction de déchiffrement* D.

Il est donc requis que : $D(C) = D(E(M)) = M$. Autrement dit, D et E sont injectives.

Un *algorithme cryptographique* est la définition des fonctions mathématiques utilisées pour le chiffrement et le déchiffrement.

En pratique, les fonctions E et D sont paramétrées par des clés K_e et K_d qui peuvent prendre l'une des valeurs d'un ensemble appelé *espace des clés*. On a donc :

$$\begin{cases} E_{K_e}(M) = C \\ D_{K_d}(C) = M \end{cases} \tag{4.1}$$

Le type de relation qui unit les clés K_e et K_d utilisées dans le chiffrement et le déchiffrement permet de définir deux grandes catégories de systèmes

cryptographiques :
- les systemes classiques à clé secrète d'une part (§4.3) ;
- les systèmes à clé publique d'autre part (§4.4).

En outre, les fontions E et D peuvent fonctionner de deux façons :
- *en continu* : dans ce cas, chaque nouveau bit est manipulé directement.
- *par bloc* : chaque message à traiter est d'abord partitionné en blocs de tailles fixes, en ajoutant éventuellement de l'information nulle en fin de message pour obtenir des blocs entiers. Les fonctions s'appliquent alors sur chaque bloc.

4.2 Principes généraux

Avant d'aller plus loin, il convient de présenter les différents types de menaces et les principales fonctionnalités offertes par la cryptographie.

4.2.1 Les grands types de menaces

Attaques passives/actives

Lorsque Alice et Bob communiquent sur un canal non sécurisé, les messages échangés peuvent être ménacés par un attaquant que nous appelerons Oscar. On distingue déjà deux types d'attaques :

1. **Les attaques passives** : dans ce cas, Oscar se contente d'écouter les messages échangés entre Alice et Bob, comme cela est présenté dans la figure 4.1.

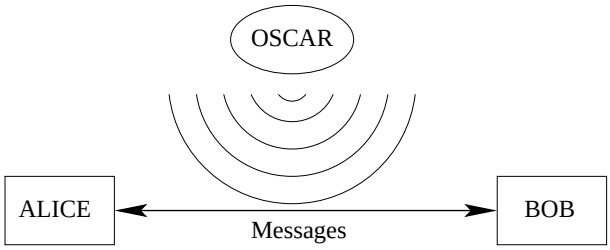

FIG. 4.1 – Principe de l'attaque passive Oscar se contente d'écouter le message

   Une telle attaque menace la *confidentialité* de l'information : une information sensible parvient également à une autre personne que son destinataire légitime.

2. **Les attaques actives** : ici, Oscar peut également modifier le contenu et/ou la forme des messages échangés entre Alice et Bob (voir figure 4.2)

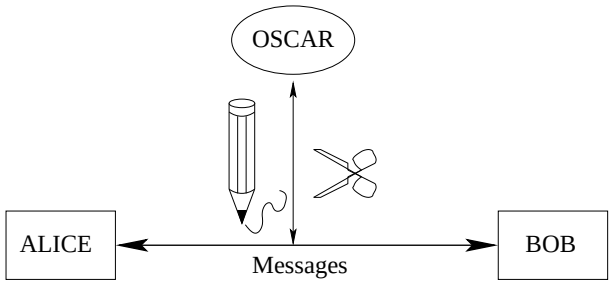

FIG. 4.2 – Principe de l'attaque active : Oscar peut modifier les messages

Dans ce cas, il y a une menace contre l'*intégrité* de l'information : l'information reçue est interprétée comme provenant d'une personne autre que son véritable auteur.

On peut énumérer un certains nombre d'attaques actives :
- l'usurpation d'identité (de l'émetteur ou du recepteur) ;
- l'altération/modification des données, du contenu des messages ;
- la destruction de messages ;
- le retardement de la transmission ;
- la répétition de messages (jusqu'à engorgement) ;
- la répudiation de message : l'émetteur nie avoir envoyé le message.

Les attaques sur un chiffrement

En supposant qu'Alice et Bob utilise un algorithme de chiffrement pour communiquer, Oscar tentera d'appliquer des méthodes pour déchiffrer les messages envoyés C.

On parle plus généralement de *cryptanalyse* : l'étude de la sécurité des procédés de chiffrement utilisés en cryptographie.

En supposant qu'Oscar connaisse le système cryptographique utilisé (principe de Kerckhoffs), il convient déjà de distinguer les niveaux d'attaques possibles :

Texte chiffré connu Oscar ne connaît que le message chiffré C.

Texte clair connu Oscar dispose à la fois d'un texte clair M et de sa correspondance chiffrée C.

Texte clair choisi Oscar peut choisir un texte clair M et obtenir le texte chiffré associé C.

Texte chiffré choisi Oscar peut choisir un texte chiffré C et obtenir le texte déchiffré associé M.

Dans tous les cas, garantir la confidentialité des communications entre Alice et Bob signifie qu'Oscar ne peut pas :
- trouver M à partir de $E(M)$; le système de chiffrement doit être résistant aux attaques sur le message codé,

– trouver la méthode de déchiffrement D à partir d'une séquence $\{E(M_i)\}$ pour une séquence quelconque de messages clairs $\{M_1, M_2, M_3, \ldots\}$, le système doit être sûr vis-à-vis des attaques avec du texte en clair.

Les fonctions de chiffrement sont paramétrées par des clés. Pour "casser" un algorithme de chiffrement, on cherche le plus souvent à découvrir la valeur de ces clés. En liaison avec le chapitre 2 du cours on peut dire qu'une clé est bonne si son entropie est élevée car cela signifie qu'elle ne contient pas de *patterns* répétés plusieurs fois.

Parallèlement aux niveaux d'attaques définis précédemment, il existe un certain nombre d'algorithmes généraux qu'Oscar va pouvoir utiliser :

Attaque brutale Elle consiste en l'énumération de toutes les valeurs possibles de la clé, c'est-à-dire à une exploration complète de l'espace des clés. La complexité de cette méthode apparaît immédiatement : une clé de 128 bits oblige à essayer 2^{128} combinaisons différentes.

Pour une clé de 64 bits, il existe $1.844 * 10^{19}$ combinaisons différentes, sur un ordinateur calculant un milliard de clés par seconde il faudra 584 ans pour être sûr de trouver la clé[1].

Attaque par séquences connues Ce type d'attaque consiste à supposer connue une certaine partie du message en clair (par exemple les entêtes standards dans le cas d'un message transmis par courrier électronique) et de partir de cette connaissance pour essayer de deviner la clé.

Cette attaque peut réussir si l'algorithme de chiffrement laisse apparaître des *patterns* du message original.

Attaque par séquences forcées Cette méthode d'attaque est basée sur la précédente, elle consiste à faire chiffrer par la victime un bloc dont l'attaquant connaît le contenu.

Attaque par analyse différentielle Cette attaque utilise les faibles différences existant entre des messages successifs (par exemple des *logs* de serveur) pour essayer de deviner la clé.

4.2.2 Les fonctionnalités offertes par la cryptographie

Nous venons de voir différents types de menaces qui pèsent sur les communications échangées entre deux personnes sur un canal non sécurisé.

La cryptographie apporte un certain nombre de fonctionnalités pour pallier à ces menaces résumé dans le sigle CAIN (Confidentialité - Authentification - Intégrité - Non-répudiation) :

1. **Pour assurer la confidentialité** : utilisation d'un algorithme de chiffrement. La confidentialité consiste à *empêcher l'accès* aux informations qui transitent pour ceux qui ne sont pas autorisés. Ils peuvent lire les messages transmis sur le canal mais ne peuvent pas le déchiffrer.

[1] ou un an avec 584 ordinateurs tournant en parallèle

2. **Contre l'usurpation d'identité** : utilisation d'algorithmes d'authentification. Il s'agit à Alice de s'identifier à Bob en prouvant qu'elle connaît un secret S, comme par exemple un mot de passe.
3. **Contre l'altération de données** : utilisation d'algorithmes de contrôle d'intégrité (voir figure 4.3).

   De tels algorithmes permettent de vérifier que le message n'a pas subi d'*altérations* lors de son parcours. Cette vérification ne se place pas au même niveau que celles vues au chapitre 5, elle concerne plutôt une modification volontaire par un tiers lors de son transfert sur le canal, qui aurait donc dans ce cas modifié les *checksums* pour masquer ses modifications.

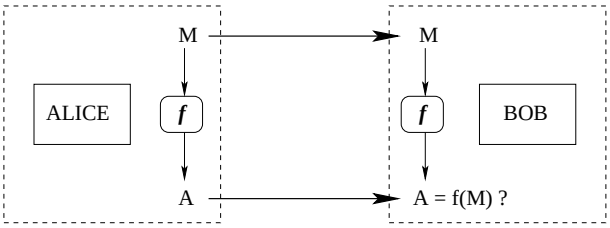

FIG. 4.3 – Principe d'un algorithme de contrôle d'intégrité

   Il doit être impossible de trouver deux messages M_1 et M_2 ayant la même image A par f.
4. **Contre la répudiation** : utilisation d'algorithmes de signatures (§ 4.7).

Le chiffrement classique ne s'intéresse généralement qu'au premier aspect et, jusqu'à ces dernières années, ne s'intéressait qu'aux clés secrètes. Les vingt dernières années ont vu émerger l'étude de nouvelles tendances :

- l'authentification devient aussi, voire même plus, importante que le secret[2],
- une partie de la clé ne doit pas être privée, afin de ne pas provoquer une explosion du nombre de clés nécessaires pour communiquer avec un grand nombre de personnes.

De plus, les opérations de chiffrement et déchiffrement portant sur de grandes quantités de données, le critère d'efficacité est très important afin de pouvoir chiffrer "à la volée" des flux audio ou vidéo par exemple.

Un système de chiffrement idéal devrait donc résoudre tous ces problèmes simultanément : utiliser des clés publiques, fournir du secret, de l'authentification et de l'intégrité. Malheureusement, il n'existe pas de technique unique qui satisfasse ces trois critères. Les systèmes conventionnels comme le DES (voir §4.3.4) sont efficaces mais utilisent des clés privées ; les systèmes à clé

[2]Cela est particulièrement vraie dans le cas du commerce électronique : il faut pouvoir prouver que la commande vient bien de la personne à qui la livraison est destinée pour éviter les contestations.

publique fournissent de l'authentification mais sont inefficaces pour le chiffrement de grandes quantités de données car trop coûteux. Cette complémentarité a motivé le développement de protocoles cryptographiques hybrides, à l'instar de PGP (Pretty Good Privacy) (§4.5) , qui sont basés à la fois sur des clefs publiques et des clefs secrètes.

4.3 Système cryptographique à clef privée.

4.3.1 Principe du chiffrement à clé secrète

Pour reprendre les notations vues dans l'équation 4.1, on a $K_e = K_d = K$. Autrement dit, Alice et Bob conviennnent secrètement d'une clé privée K. Ils conviennent également d'un algorithme cryptographique. Des exemples de tels algorithmes seront fournis dans les paragraphes suivants.

La clé K est donc utilisée à la fois pour le chiffrement et le déchiffrement.

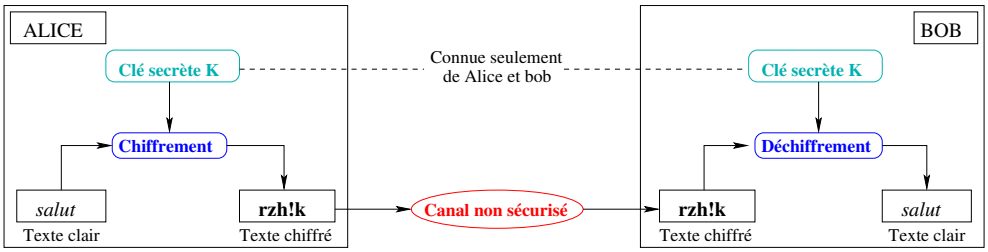

FIG. 4.4 – Principe du chiffrement a cle secrète

En pratique, on utilise principalement des chiffrements par bloc tel que le DES (Data Encryption Standard : cf § 4.3.4) ou plus récemment l'AES (Advanced Encryption Standard).

De tels systèmes ont l'avantage principal d'être efficaces en terme de temps de calcul, tant pour le chiffrement que pour le déchiffrement. En revanche, la faiblesse de ce système vient du secret absolu qui doit entourer la clé K. Si Oscar parvient par un moyen quelconque a obtenir cette clé, il pourra déchiffrer tous les messages échangés entre Alice et Bob.

Nous allons maintenant présenter plusieurs systèmes conventionnels à clé privée.

4.3.2 Au commencement était César...

Initialement, le secret echangé était la technique mise en oeuvre, ou l'algorithme de chiffrement E. Ainsi, le chiffrement de César substituait chaque lettre du message initial M par celle située 3 positions plus loin dans l'alphabet[3].

[3]Autrement dit, $E(x) = x + 3 \mod n$, si n est le cardinal de l'alphabet utilisé

Ensuite, E et D furent paramétrées par une simple clé K, choisie secrètement entre Alice et Bob. D'autres systèmes cryptographiques, plus élaborés virent alors le jour (chiffrement affine, par substitution etc...).

Pour ces systèmes dits à *substitution monoalphatbétique*, une cryptanalyse simple consiste à étudier la fréquence d'apparition des lettres au sein du texte chiffré et d'en déduire la correspondance avec les lettres du texte clair. Le tableau 4.1 présente la répartition statistique des lettres dans les textes français.

A	8.11 %	N	7.68 %
B	0.81 %	O	5.20 %
C	3.38 %	P	2.92 %
D	4.28 %	Q	0.83 %
E	17.69 %	R	6.43 %
F	1.13 %	S	8.87 %
G	1.19 %	T	7.44 %
H	0.74 %	U	5.23 %
I	7.24 %	V	1.28 %
J	0.18 %	W	0.06 %
K	0.02 %	X	0.53 %
L	5.99 %	Y	0.26 %
M	2.29 %	Z	0.12 %

TAB. 4.1 – Fréquence d'apparition des lettres en français

Exercice 3. *Toto trouve un papier sur lequel se trouve le message chiffré suivant :* `YN PHEVBFVGR RFG HA IVYNVA QRSNHG!`

Aider toto à déchiffrer ce message.

Pour contrer cette cryptanalyse, Vigenère mit au point en 1586 un système pour lequel une clef définit le décalage pour chaque lettre du message (A : décalage de 0, B : 1, C : 2, ..., Z : 25). Ce chiffrement est illustré dans le tableau 4.2.

Clair	L	A	V	I	E	E	S	T	B	E	L	L	E
Clef	B	O	N	J	O	U	R	B	O	N	J	O	U
Décalage	1	14	13	9	14	20	17	1	14	13	9	14	20
Chiffré	M	O	I	R	S	Y	J	U	P	R	U	Z	Y

TAB. 4.2 – Chiffrement de Vigenère utilisant la clé BONJOUR

Finalement, en 1854, Charles Babbage proposa une cryptanalyse du chiffrement de Vigenère basée sur la répétition de la clé.

Cela nous permet d'introduire la notion de *sécurité inconditionnelle* où la connaissance du message chiffré n'apporte aucune information sur le message

clair. Ainsi, la seule attaque possible est la recherche exhaustive de clé secrète.

Pour qu'un système soit inconditionnellement sûr, il faut donc que la clé secrète soit au moins aussi longue que le texte clair. C'est le principe du système du chiffrement à clé jetable présenté maintenant.

4.3.3 Chiffrement à usage unique.

Le système à clef jetable (*One time pad*[4]) est un système à clef privée K dans lequel cette clef n'est utilisée qu'une fois. Il repose sur le fait que pour tous messages M et N de même longueur, on a :

$$(M \oplus N) \operatorname{xor} N = M$$

où $\oplus$ (qu'on note aussi xor) désigne l'opération logique "ou exclusif".

Pour envoyer un message M de n bits, il faut avoir une clef K secrète de n bits. Le message chiffré $\tilde{M}$ est donné par $\tilde{M} = M \oplus K$. Pour déchiffrer, il suffit de calculer $\tilde{M} \oplus K$.

Exercice 4. *Pourquoi faut-il jeter la clef après l'avoir utilisée, i.e. changer de clef pour chaque nouveau message ?*

Exercice 5. *Construire un protocole, à base de clefs jetables, permettant à un utilisateur de se connecter depuis un PC quelconque sur internet à un serveur sécurisé. Le mot de passe doit être crypté pour ne pas circuler visible sur internet.*

Sous réserve que la clé K ait bien été générée de façon totalement aléatoire et qu'elle n'ait été utilisée qu'une seule fois pour chiffrer le message, Oscar n'obtient aucune information sur le message clair M s'il intercepte C (hormis la taille de M).

Ce chiffrement est le seul algorithme cryptographique à clé secrète prouvé inconditionnellement sûr. Ainsi tous les autres systèmes sont théoriquement cassables. Pour ces systèmes, on utilise des chiffrements *pratiquement sûr* : la connaissance du message chiffré (où de certains couples message clair/message chiffré) ne permet de retrouver ni la clé secrète ni le message clair *en un temps humainement raisonnable.*

En 1977, un standard de chiffrement *pratiquement sûr* fut mis en place : il s'agit du système DES présenté dans le paragraphe suivant.

4.3.4 Le système DES (Data Encryption Standard).

Ce système de chiffrement à clef privée est le plus connu. Il est bien documenté ([28] [32]) et ne sera pas présenté en détails ici, il n'est donné qu'à

[4]En 1917, pendant la première guerre mondiale, AT&T a chargé le scientifique américain Gilbert Vernam d'inventer une méthode de chiffrage que les Allemands ne pourraient pas casser : le chiffrement jetable que celui-ci a conçu est le seul code mathématiquement prouvé sûr aujourd'hui connu.

titre de comparaison de son fonctionnement avec celui des autres systèmes présentés.

Ce système fonctionne par blocs de texte clair de 64 bits en utilisant une clef de 56 bits[5]. Il permet d'obtenir ainsi des blocs de texte chiffré de 64 bits. D'une manière générale, l'algorithme se déroule en trois étapes :

1. soit x un bloc de texte clair de 64 bits. On lui applique une permutation initiale IP fixée pour obtenir une chaîne x_0. On a donc :

$$x_0 = IP(x) = L_0 R_0$$

où L_0 contient les 32 premiers bits de x_0 et R_0 les 32 restants.

2. 16 itérations (où 16 tours) d'une certaine fonction sont effectuées. On calcule $L_i R_i, 1 \leq i \leq 16$ suivant la règle :

$$\begin{cases} L_i = R_{i-1} \\ R_i = L_{i-1} \oplus f(R_{i-1}, K_i) \end{cases} \tag{4.2}$$

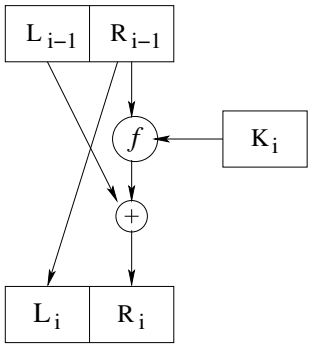

FIG. 4.5 – Un tour de DES

La figure 4.5 présente un tour de chiffrement. La fonction f est une fonction à deux variables, l'une de 32 bits (correspondant à R_{i-1} à la i^{eme} itération) et l'autre de 48 bits (il s'agit de K_i). Les éléments K_i sont obtenus par diversification des bits de la clé initiale K (de 56 bits).

3. La permutation inverse IP^{-1} est appliquée à $R_{16}L_{16}$ pour obtenir un bloc de texte chiffré $y = IP^{-1}(R_{16}L_{16})$ (à noter l'inversement de L_{16} et de R_{16}).

Le même algorithme (à quelques légères nuances près) est utilisé pour déchiffrer.

Exercice 6. *Détailler l'algorithme de déchiffrement du système DES.*

[5]en fait, la clef compte 64 bits mais parmi lesquels 1 bit sur 8 est utilisé comme bit de parité

La sûreté du DES vient de la fonction f qui est en fait une substitution suivie d'une permutations. De plus, après 16 tours, le résultat est statistiquement "plat", c'est-à-dire que les caractéristiques générales du message source (la fréquence des caractères, le nombre d'espaces, ...) seront indétectables. De plus il dispose d'une caractéristique très importante pour éviter les attaques par analyse différentielle : une légère modification de la clef ou du texte à chiffrer provoque des changements importants dans le texte chiffré.

Le gros avantage du DES est qu'il repose, tant pour le chiffrement que pour le déchiffrement, sur des opérations facilement implantables au niveau matériel, il est donc possible d'obtenir des taux de chiffrement très élevés, de l'ordre de 40Mo/s il y a une dizaine d'années, avec du matériel spécifique.

Comme applications de DES [28], citons : les cartes de crédit (par exemple *UEPS* pour *Universal Electronic Payment System* voir [28] pages 618 – 620), des protocoles d'authentification dans les réseaux tels que *Kerberos* (voir [28] pages 594 – 600),) et la messagerie électronique avec par exemple *PEM* pour *Privacy-Enhanced Mail* (voir [28] pages 606 – 612).

Avec les progrès réalisés dans le domaine de la cryptanalyse depuis 1997, en particulier la progression de la puissance de calcul des processeurs d'aujourd'hui, le DES avec sa longueur de clé fixe de 56 bits n'est plus considéré comme pratiquement sûr. Un petit calcul simple peut en donner une idée : il y a 2^{56} clefs possibles c'est-à-dire moins de $10^{17} = 10^8 10^9$; en supposant que l'on dispose de 1000 PC tournant à $1GHz = 10^9 Hz$, il faut 10^5 secondes, soit 30 heures pour donc effectuer $1000 * 10^9 * 10^5 = 10^{17}$ opérations ; le cassage par force brute n'est donc pas impossible en un temps raisonnable !

C'est pourquoi un nouveau standard a pris sa place depuis 2000. Il s'agit de l'AES (Advanced Encryption Standard [15, 7]), qui travaille sur des blocs de 128 bits et dont la clef de chiffrement admet une taille variable (128, 192 ou 256 bits).

Pour finir, on peut citer d'autres exemples de systèmes de chiffrement à clé secrète :

- IDEA (1992) : blocs de 64 bits, clef de 128 bits ;
- Triple DES à deux clefs : blocs de 64 bits, clef de 112 bits :

$$\begin{cases} C = E_{K_1}(D_{K_2}(E_{K_1}(M))) \\ M = D_{K_1}(E_{K_2}(D_{K_1}(C))) \end{cases}$$

4.3.5 Le nouveau standard AES (Rijndael).

Rijndael est le chiffrement à clef privée qui a été retenu par le NIST (National Institute of Standards and Technology) comme le nouveau standard américain de chiffrement (AES : Advanced Encryption Standard)[15, 8]. Ce système a été choisi après une compétition organisée par le NIST. Des cinq finalistes (Rijndael, Serpent, Twofish, RC6, Mars), c'est donc Rijndael [7]

qui s'est avéré le plus résistant et est donc devenu le nouveau standard américain, remplaçant le DES. C'est un code par blocs encodant 128 bits avec des clefs de 128, 192 ou 256 bits. Il est important de noter que Rijndael était parmi les proposition les plus rapides, comme on peut le voir dans la table 4.3 issue du rapport NESSIE-D21.

Algorithme	Taille de clef	Cycle/octet	
		Cryptage	Décryptage
IDEA	128	56	56
Khazad	128	40	41
Misty1	128	47	47
Safer++	128	152	168
CS-Cipher	128	156	140
Hierocrypt-L1	128	34	34
Nush	128	48	42
DES	56	59	59
3-DES	168	154	155
kasumi	128	75	74
RC5	64	19	19
Skipjack	80	114	120
Camellia	256	47	47
RC6	256	18	17
Safer++	256	69	89
Anubis	256	48	48
Grand cru	128	1250	1518
Hierocrypt-3	260	69	86
Nush	256	23	20
Q	256	60	123
SC2000	256	43	46
Rijndael	256	34	35
Serpent	256	68	80
Mars	256	31	30
Twofish	256	29	25

TAB. 4.3 – Vitesses comparées de quelques chiffrements par blocs sur un PIII/Linux

Principe de l'algorithme

Tous les opérations d'AES sont des opérations sur des octets considérés comme des éléments du corps fini à 2^8 éléments, $GF(256)$. Un octet est donc représenté comme un polynôme de degré au plus 7 : $b_7b_6b_5b_4b_3b_2b_1b_0 \leftarrow b_7X^7 + b_6X^6 + b_5X^5 + b_4X^4 + b_3X^3 + b_2X^2 + b_1X + b_0$. Toutes les opérations

se font alors modulo 2 et modulo le polynôme $\tilde{P} = X^8 + X^4 + X^3 + X + 1$, irréductible modulo 2.

Sur ce corps, le principe de l'AES est alors le suivant : le message est segmenté en blocs de 128 bits. Ces 128 bits sont arrangés sous la forme d'une matrice d'octets 4×4. 128 bits de la clef sont alors ajoutés à cette matrice initiale (l'addition modulo 2 équivaut à un XOR bit à bit). Chaque élément de la matrice est alors remplacé par l'élément correspondant à une table de substitution (ou "S-box", obtenue par inversion d'un polynôme dans $GF(256)$ – grâce à l'algorithme d'Euclide étendu –, puis par transformation affine de cette inverse). La matrice obtenue est alors soumise à deux autres transformations linéaires, l'une mélangeant les lignes, l'autre les colonnes. Cette procédure est répétée dix fois (10 rondes) en utilisant, à chaque ronde, une partie différente de la clef secrète.

Détail d'une ronde

Une ronde comporte donc quatre étapes appliquées sur la matrice appelée "state" [7] :

1. AddRoundKey(state,roundkey) : XOR bit à bit des octets de la matrice et de ceux de la partie de la clef concernée.
2. ByteSub(state) : La matrice $\begin{bmatrix} \mathcal{O}_{0,0} & \mathcal{O}_{0,1} & \mathcal{O}_{0,2} & \mathcal{O}_{0,3} \\ \mathcal{O}_{1,0} & \mathcal{O}_{1,1} & \mathcal{O}_{1,2} & \mathcal{O}_{1,3} \\ \mathcal{O}_{2,0} & \mathcal{O}_{2,1} & \mathcal{O}_{2,2} & \mathcal{O}_{2,3} \\ \mathcal{O}_{3,0} & \mathcal{O}_{3,1} & \mathcal{O}_{3,2} & \mathcal{O}_{3,3} \end{bmatrix}$ est transformée en $\begin{bmatrix} S(\mathcal{O}_{0,0}) & S(\mathcal{O}_{0,1}) & S(\mathcal{O}_{0,2}) & S(\mathcal{O}_{0,3}) \\ S(\mathcal{O}_{1,0}) & S(\mathcal{O}_{1,1}) & S(\mathcal{O}_{1,2}) & S(\mathcal{O}_{1,3}) \\ S(\mathcal{O}_{2,0}) & S(\mathcal{O}_{2,1}) & S(\mathcal{O}_{2,2}) & S(\mathcal{O}_{2,3}) \\ S(\mathcal{O}_{3,0}) & S(\mathcal{O}_{3,1}) & S(\mathcal{O}_{3,2}) & S(\mathcal{O}_{3,3}) \end{bmatrix}$ où, si $y \equiv a^{-1} \mod \tilde{P}$ (0 n'ayant pas d'inverse, il est envoyé sur 0 pour cette étape) et si
$$z = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} y + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \mod 2$$
alors, $S(a) = z$.
3. ShiftRow(state) : une permutation cyclique est effectuée sur chaque ligne pour obtenir : $\begin{bmatrix} z_{0,0} & z_{0,1} & z_{0,2} & z_{0,3} \\ z_{1,1} & z_{1,2} & z_{1,3} & z_{1,0} \\ z_{2,2} & z_{2,3} & z_{2,0} & z_{2,1} \\ z_{3,3} & z_{3,0} & z_{3,1} & z_{3,2} \end{bmatrix}$
4. MixColumn(state) : pour cette étape, chaque colonne de la matrice est considérée comme un polynôme de degré 3 de $GF(256)[Y]$. Chaque

colonne est alors multipliée par le polynôme $C = [3]Y^3 + [1]Y^2 + [1]Y + [2]$ modulo $G = Y^4 + [1]$, où [3], par exemple, est $[3] = [0000011] = X + 1$ dans $GF(256)$. Comme les opérations se font modulo $Y^4 + [1]$, cette multiplication par C, correspond à une multiplication de chaque colonne par la matrice suivante :

$$M_j = \begin{bmatrix} [2] & [3] & [1] & [1] \\ [1] & [2] & [3] & [1] \\ [1] & [1] & [2] & [3] \\ [3] & [1] & [1] & [2] \end{bmatrix} C_j$$

En outre, comme C et G sont premiers entre eux, C est bien inversible modulo G et la transformation MixColumn est également inversible, ce qui permet de décrypter.

Effets sur la cryptanalyse

La "S-box" a été construite pour être résistante à la cryptanalyse. En particulier, elle ne possède pas de point fixe $S(a) = a$, ni de point opposé $S(a) = (!a)$, ni de point fixe inverse $S(a) = S^{-1}(a)$. Ensuite, l'opérateur ShiftRow permet de diffuser largement les données en séparant les octets originellement consécutifs. Enfin, combinée avec MixColumn, elle permet qu'après plusieurs rondes, chaque bit de sortie dépende de tous les bits en entrée. Par ailleurs, MixColumn est un code linéaire de distance maximale (voir section 5.2.1).

4.3.6 Les modes de chiffrement

Que ce soit pour le DES ou les nouveaux standards IDEA, AES, les clefs sont de longueur fixée. Or les messages peuvent être de longueur quelconque bien sûr. Il faut donc initier des chiffrements par blocs de taille fixe correspondants aux tailles des clefs. Pour cela 4 modes de chiffrement par blocs sont possibles : ECB, CBC, CFB ett OFB.

Le mode ECB

Ce mode est le plus simple, le message M est découpé en blocs m_i de taille fixe et chaque bloc est crypté séparément par

$$c_i = E_k(m_i) \tag{4.3}$$

suivant la figure 4.6.

Ainsi un bloc de message donné m_i sera toujours codé de la même manière. Ce mode de chiffrement ne présente donc aucune sécurité et n'est jamais utilisé !

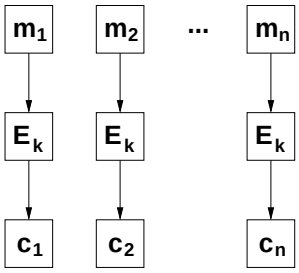

FIG. 4.6 – Mode ECB : Electronic Code Book

Le mode CBC

Le mode CBC a donc été introduit pour que un bloc ne soit pas codé de la même manière si il est dans deux messages différents. Il faut ajouter une

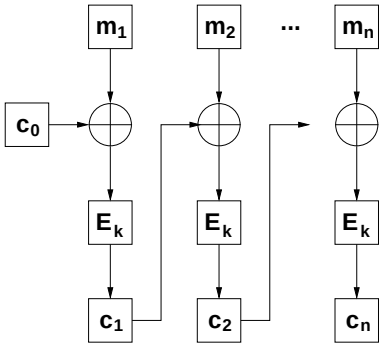

FIG. 4.7 – Mode CBC : Cipher Bloc Chaining

valeur initiale C_0, aléatoire par exemple. Chaque bloc est d'abord modifié par XOR avec le bloc crypté précédent avant d'être lui même crypté par

$$c_i = E_k(m_i \oplus c_{i-1}) \tag{4.4}$$

suivant la figure 4.7. C'est le mode de chiffrement le plus utilisé. Le déchiffrement nécessite l'inverse de la fonction de codage $D_k = E_k^{-1}$ pour décrypter :

$$m_i = c_{i-1} \oplus D_k(c_i) \tag{4.5}$$

Le mode CFB

Pour ne pas avoir besoin de la fonction inverse pour décrypter, il est possible de faire un XOR après le cryptage, c'est le mode CFB :

$$c_i = m_i \oplus E_k(c_{i-1}) \tag{4.6}$$

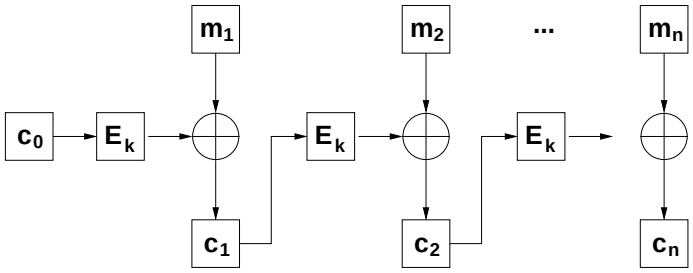

FIG. 4.8 – Mode CFB : Cipher FeedBack

suivant la figure 4.8. Ce mode est moins sûr que le CBC et est utilisé par exemple pour les cryptages réseaux. L'intérêt est que le déchiffrement ne nécessite pas D_k :

$$m_i = c_i \oplus E_k(c_{i-1}) \tag{4.7}$$

Le mode OFB

Enfin, un dernier mode permet d'avoir un cryptage et un décrypteage totalement symétrique, c'est le mode OFB :

$$z_i = E_k(z_{i-1}) \ ; \ c_i = m_i \oplus z_i \tag{4.8}$$

suivant la figure 4.9. Ce mode est utilisé par exemple pour les cryptages

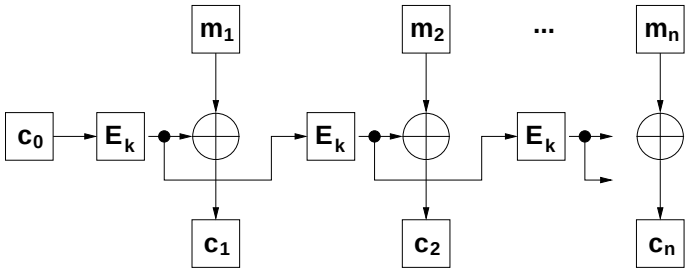

FIG. 4.9 – Mode OFB : Output FeedBack

satellite et se déchiffre par :

$$z_i = E_k(z_{i-1}) \ ; \ m_i = c_i \oplus z_i \tag{4.9}$$

4.4 Système cryptographique à clef publique.

4.4.1 Motivations

Nous avons vus que les systèmes cryptographiques à clé secrètes sont pratiquement sûrs et efficaces en termes de temps de calcul. Néanmoins, dès

le milieu des années 1970, de nouvelles interrogations furent soulevées :

- Avant d'utiliser un système de chiffrement à clé secrète, comment convenir d'une clé ?
- Comment établir une communication sécurisée entre deux entités sans échange préalable de clef ?

C'est pour répondre à ces questions qu'en 1976, Diffie et Hellman posèrent les bases des systèmes cryptographiques à clé publique, par analogie avec une boîte aux lettres dont Bob est le seul à posséder la clé :

- toute personne peut envoyer du courrier à Bob ;
- seul Bob peut lire le courrier déposé dans sa boîte aux lettres.

Pour comparaison, un système de chiffrement à clef secrète peut être vu comme un coffre-fort dont la clé est partagée par Alice et Bob.

4.4.2 Principe

Dans le cadre d'un tel système et toujours pour reprendre les notations de l'équation 4.1, on a cette fois $K_e \neq K_d$.

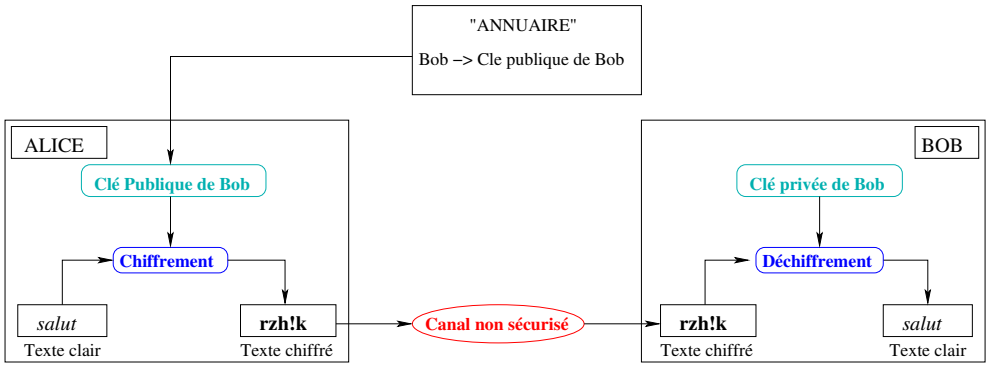

FIG. 4.10 – Principe du chiffrement à clef publique

Pour être plus précis, K_e est une clé publique qui est publiée dans une sorte d'annuaire (en fait sous la forme d'un certificat :voir §4.8) de tel sorte que n'importe qui puisse récupérer cette clé, en tester l'origine et chiffrer un message avec.

K_d est une clé secrète que Bob garde pour lui et qui lui servira à dechiffrer.

Evidemment, les deux clés K_e et K_d sont mathématiquement liés mais choisies de telle sorte qu'il soit pratiquement infaisable de calculer la valeur de la clé K_d à partir de K_e.

Le principe du chiffrement avec un tel système est illustré dans la figure 4.10. L'exemple le plus connu d'algorithme de chiffrement à clé publique est le système RSA, exposé dans la section 4.4.5.

4.4.3 Réalisation : notion de Fonctions à Sens Unique

Dans un système de chiffrement à clef publique, le fait que la fonction de chiffrement E soit publique alors que la fonction de déchiffrement D soit secrète semble à priori contradictoire : si l'on connaît E, on connaît forcément D puisque $D = E^{-1}$. En fait, en remplaçant "inconnu" par "extrèmement long à calculer sur un ordinateur" (i.e. plusieurs années par exemple), les fonctions E et D d'un système de cryptographie à clef publique doivent vérifier :

- $D = E^{-1}$ pour garantir $D(E(M)) = M$;
- il est facile (i.e. très rapide) de calculer $\tilde{M} = E(M)$ à partir de M ;
- il est difficile (i.e. très long) de retrouver M à partir de $\tilde{M}$ si l'on ne connaît pas K_d. Par contre, D_{K_d} doit être facile (rapide) à évaluer pour pouvoir déchiffrer le message puisque $M = D(\tilde{M})$.

Autrement dit, il faut trouver une fonction E de chiffrement qui soit rapide à calculer mais longue à inverser. On parle de fonction "à sens unique" (one way function, FSU dans la suite).

De bonnes FSU sont des fontions telles que la recherche de x à partir de $F(x)$ soit un problème mathématique réputé difficile (par exemple NP-complet).

Pour obtenir de tels fonctions, il a fallu changer la vision des messages et des clés comme de simples séquences de bits. Les méthodes modernes de cryptographie, et tout particulièrement les méthodes à clefs publiques, sont basées sur cette vision d'un message comme un entier, ou plutôt comme une suite d'entiers compris entre 0 et n. Si l'on pose $L = \lfloor \log_2 n \rfloor$, on crypte alors le message par blocs de L bits. Chaque bloc M représente alors un nombre entre 0 et $n-1$ et donc, un élément de l'anneau des classes résiduelles modulo n. Le bloc M est chiffré en $\tilde{M} = F(M) \pmod{n}$ où F est une fonction dans les entiers.

L'intérêt de calculer en arithmétique modulaire est double. D'une part, les calculs "modulo n" sont très rapides : ils prennent un coût $O(\log^2 n)$ avec les algorithmes les plus naïfs. D'autre part, les itérations d'une fonction F même simple calculées en arithmétique modulo n tendent à avoir un comportement aléatoire[6]. Connaissant F et n, il apparaît difficile de résoudre l'équation : trouver x tel que $F(x) \equiv a \pmod{n}$, donc d'inverser la fonction F.

4.4.4 Un exemple de FSU : l'Exponentiation Modulaire

Le théorème d'Euler (section 1.2.2, page 17) peut être directement utilisé pour fournir une fonction à sens unique : l'exponentiation modulaire.

- Soient p et q deux entiers premiers et $n = p.q$.
- Soit e un entier inférieur à n, premier avec $\phi(n) = (p-1)(q-1)$.

[6]Ce type de calcul est d'ailleurs utilisé dans la plupart des générateurs de nombres aléatoires.

La fonction d'exponentiation modulaire est alors définie par :

$$\begin{array}{rccc} F_e : & \mathbb{Z}_n & \longrightarrow & \mathbb{Z}_n \\ & x & \longrightarrow & x^e \bmod n \end{array}$$

Cette fonction est rapide à calculer (en fait en temps linéaire de e et donc de n) grâce à un algorithme de type exponentiation rapide, encore appelé *élévation récursive au carré* (voir section 1.2.4, page 21).

La fonction inverse de F_e n'est autre que la fonction F_d, ou $d = e^{-1} \mod \phi(n)$. Autrement dit, on a : $d.e \equiv 1 \mod (p-1)(q-1)$.

En effet, si $y = F_e(x)$, alors $y^d = x^{e.d} \bmod n = x^{1+\lambda\phi(n)} \bmod n = x$ d'après le théorème RSA (voir §4, page 20).

Le calcul de $d = e^{-1} \mod \phi(n)$ peut être fait facilement à partir de l'algorithme d'Euclide Etendu EE exposé dans la section 1.2.1, page 16, puisque e et $\phi(n)$ sont premiers entre eux. Après l'appel :

$$EE(e, \phi(n), g, d, v)$$

on obtient le pgcd $g = 1$ ainsi que les valeurs de d et v telles que

$$e.d + v.\phi(n) = g = 1 \Rightarrow e.d \equiv 1 \bmod \phi(n)$$

.

Par exemple, calculons $17^{-1} \pmod{2668}$; si l'on écrit la valeur du coefficient dans la colonne de droite, on obtient :

(a)	2668	0	
(b)	17	1	
(c)	$2668 - 140.17 = 16$	$(a - 156b)$	-156
(d)	$17 - 16 = 1$	$(b - c)$	157

La valeur de $17^{-1} \pmod{2668}$ est donc 157.

En fait, la véritable difficulté pour calculer d est liée à l'évaluation de $\phi(n) = (p-1)(q-1)$.

Celle-ci est aisée si on connait la valeur de p et de q. Sinon, le problème se ramène à factoriser l'entier n en produits de nombres premiers. Cependant, malgré tous les travaux menés sur le problème de la factorisation d'entiers depuis 300 ans, aucun algorithme rapide n'a été publié (donc a priori découvert) à ce jour[7].

C'est sur cette constatation qu'est fondée la confiance portée au sytème cryptographique RSA qui est maintenant exposé.

[7]Le record actuel fut obtenu en janvier 2002 par F. Bahr, J.Frankle et T.Kleinjung avec la factorisation d'un nombre de 158 chiffres en produit de 2 nombres premiers

4.4.5 Le système cryptographique à clé publique RSA

Ce sytème, proposé par Rivest, Shamir et Adleman en 1978 est aujourd'hui le plus connu et le plus utilisé de part sa simplicité.

Description de RSA.

Dans le système RSA un utilisateur crée son couple (clé plublique, clé privée) en utilisant la procédure suivante :

1. Choisir au hasard deux grands nombres premiers p et q. Il faut que p et q contiennent au moins 100 chiffres décimaux chacun.
2. Calculer $n = pq$
3. Choisir un petit entier e qui est premier avec $\phi(n) = (p-1)(q-1)$
4. Calculer d, l'inverse par la multiplication de e modulo $\phi(n)$.
5. Publier la paire $K_e = (e, n)$ comme sa clé publique RSA.
6. Garder secrète la paire $K_d = (d, n)$ qui est sa clé privée RSA.

On a alors :

Chiffrement RSA : $E_{K_e}(M) = M^e \ mod \ n$

Déchiffrement RSA : $D_{K_d}(\tilde{M}) = \tilde{M}^d \ mod \ n$

Exercice 7. *Calculer une clé publique et une clé privée pour $p = 47$ et $q = 59$. Pour simplifier on prendra $e = 17$. Chiffrer le message B en système ASCII avec la clé publique et vérifier que la clé privée permet bien de retrouver le message initial.*

Efficacité et robustesse de RSA.

Grâce aux algorithmes vus dans la section précédente :

- Il est facile de générer des grands nombres premiers, tout au moins en acceptant un taux d'erreur (cf test de Miller-Rabin §1.2.5, page 22). Dans le cas de RSA, l'erreur n'est pas trop grave : en effet, si l'on commet une erreur en croyant que p et q sont premiers, le destinataire se rendra rapidement compte que les nombres ne sont pas premiers : soit la clef d n'est pas inversible, soit certains blocs du message décrypté sont incompréhensibles. Dans ce cas, on peut procéder à un changement de système RSA (recalcul de p et q).
- le calcul du couple (e, d) est extrèmement facile : il suffit d'appliquer l'algorithme d'Euclide étendu ;
- enfin chiffrement et déchiffrement sont réalisés par exponentiation modulaire. Nous avons vu que cette exponentiation pouvait être réalisée assez efficacement.

La sécurité fournie par RSA repose essentiellement sur la difficulté à factoriser de grands entiers. En effet, si un attaquant peut factoriser le nombre $n = pq$ de la clef publique, il peut alors déduire directement $\phi(n) = (p-1)(q-1)$ et donc calculer la clé privée à partir de la clé publique par l'algorithme d'Euclide étendu. Donc, si l'on dispose d'un algorithme rapide pour factoriser de grands entiers, casser RSA devient facile aussi.

Après 20 ans de recherche, aucun moyen plus efficace que la factorisation de n n'a été publié pour casser RSA. Cependant, la réciproque : "si factoriser de grands entiers est dur alors casser RSA est dur" n'a pas été prouvée.
On peut cependant remarquer que si l'on choisit une petite clef publique e (par exemple $e \leq \log n$) alors casser RSA permet de factoriser n en temps polynomial.

En effet, supposons que l'on ait cassé RSA ; on connaît donc la clef secrète d. Comme $ed - 1 \equiv 0 \mod (p-1)(q-1)$, on a :

$$\exists \lambda \in \mathbb{N} \ : \lambda.(pq - p - q + 1) = ed - 1$$

Or $pq = n$ et on peut suposser p et q différents de 2 et 3 ; ainsi p et q sont inférieurs à $\frac{n}{4}$, d'où $pq - p - q + 1 \geq \frac{n}{2}$. Finalement λ vérifie :

$$\lambda \leq \frac{2e.d}{n}.$$

De plus $d < n$ donc $\lambda < 2e$. Comme e est supposé petit, on peut donc essayer de déterminer p et q en essayant de manière exhaustive toutes les valeurs possibles pour $\lambda = 1 \ldots 2e - 1$. Plus précisément, posons

$$S_\lambda = n + 1 - \frac{ed+1}{\lambda}.$$

Si λ est la bonne valeur on a $S_\lambda = p + q$. Comme le produit $p.q = n$ est connu, on en déduit que p et q sont alors les deux racines entières de l'équation $X^2 - S_\lambda X + n = 0$.

D'où l'algorithme pour calculer la factorisation $p.q$ de n :

Pour $\lambda = 1, 2, ..., 2e$ **faire** $S_\lambda := n + 1 - \frac{ed+1}{\lambda}$

Si S_λ **est entier alors** calculer les 2 racines p et q de l'équation : $X^2 - S_\lambda + n = 0$.

Si p **et** q **sont entiers, alors** $n = p.q$ est une factorisation non triviale de n.

Donc, si e est petit, il est plus facile de factoriser n (un problème réputé difficile si p et q sont très grands) que de casser RSA. C'est sur ce point que réside la confiance portée à RSA.

En outre, un algorithme plus complexe permet d'arriver au même résultat, cette fois-ci sans condition sur la taille de e. L'idée est la même que celle de l'algorithme de Miller-Rabin, nous en donnons un aperçu ici :

1. Calculer s et t impair tel que $ed - 1 = 2^s t$.
2. Faire
   Choisir a premier avec n
   Tant qu'il n'existe pas de i tel que $a^{2^{i-1}t} \not\equiv \pm 1 \mod n$ et $a^{2^i t} \equiv 1 \mod n$.
3. Poser $u = a^{2^{i-1}t}$.
4. $\text{pgcd}(u-1, n)$ et $\text{pgcd}(u+1, n)$ sont des facteurs non triviaux de n.

Lemme 3. *L'algorithme est correct.*

Preuve. Si $a^{2^{i-1}t} \not\equiv \pm 1 \mod n$ et $a^{2^i t} \equiv 1 \mod n$, alors nous avons donc trouvé u tel que $u^2 \equiv 1 \mod n$ (donc $(u-1)(u+1)$ est un multiple de n) et tel que ni $u-1$, ni $u+1$ ne valent 0 ou n. Ces derniers contiennent donc des facteurs non triviaux de n. □

Nous avons donc un algorithme factorisant n, le tout est de savoir si il est rapide. En pratique, comme n est composé, un élément au hasard sur deux vérifie les conditions sur a. Ainsi le nombre d'itérations est en moyenne de deux et l'algorithme est immédiat ! Il reste donc à prouver :

Théorème 17. *Au moins un a inversible sur deux vérifie la condition de l'algorithme*

Preuve. La preuve est en deux parties : montrer qu'il existe au moins un a vérifiant ces conditions, puis montrer que dans ce cas il y en a au moins la moitié des inversibles. Nous commençons par la deuxième partie, plus facile.

Posons $m = 2^i t$ et supposons qu'il existe un a tel que $a^m \equiv 1$ et $a^{m/2} \not\equiv \pm 1$. Soit $\{b_1, \ldots, b_k\}$ l'ensemble des b_i inversibles tels que $b_i^{m/2} \equiv \pm 1 \mod n$ et $b_i^m \equiv 1 \mod n$. Alors, forcément $(b_i a)^{m/2} \not\equiv \pm 1$. Donc, si il existe un tel a, il en existe au moins $\frac{\phi(n)}{2}$, soit un inversible sur deux.

Il reste à prouver qu'il en existe au moins un. Cela n'est pas toujours le cas pour un composé quelconque, mais nous allons voir que c'est le cas pour un entier n produit de deux nombres premiers distincts. Nous avons que t est impair, donc $(-1)^t \equiv -1 \not\equiv 1 \mod n$. Donc il existe un a (au pire une certaine puissance de -1) vérifiant $u = a^{m/2} \not\equiv 1 \mod n$ et $a^m \equiv 1 \mod n$. Donc, par les restes chinois, $u^2 \equiv 1 \mod p$ et $u^2 \equiv 1 \mod q$. Ainsi, u doit être congru à 1 ou -1 modulo p et q. Il ne peut pas être congru à 1 modulo les deux nombres premiers sinon il le serait aussi modulo n. Donc, sans perte de généralité, nous supposons qu'il est congru à -1 modulo p. Si alors u est congru à 1 modulo q, nous avons trouvé notre élément.
Sinon, u est congru à -1 modulo n. Mais dans ce dernier cas, nous pouvons construire un autre élément qui peut convenir car alors $m/2$ ne divise ni $p-1$, ni $q-1$ (sinon u serait congru à 1 modulo p ou q).

En effet, nous avons $\frac{m}{2} = \frac{ed-1}{2^{s-i+1}} = \frac{k\phi(n)}{2^{s-i+1}}$, et comme $m/2$ ne divise pas $p-1$, nous pouvons poser $\frac{m}{2} = v\frac{p-1}{2^j}$ avec v impair et j entre 0 et s. Puis, nous prenons g une racine primitive modulo p et nous posons $\alpha \equiv g^{2^j} \mod p$ et $\beta \equiv a \mod q$. Alors $\alpha^{m/2} \equiv 1 \mod p$ par construction et $\beta^{m/2} \equiv a^{m/2} \equiv -1 \mod q$. Il ne reste plus qu'à reconstruire par restes chinois, γ correspondant aux restes respectifs α et β modulo p et q. Ce γ vérifie $\gamma^m \equiv 1 \mod n$ ainsi que $\gamma^{m/2} \not\equiv \pm 1 \mod n$.

Ainsi, dans tous les cas, il existe forcément au moins un élément vérifiant les conditions de l'algorithme, et donc au moins un élément sur deux vérifie bien cette précédente propriété et le cassage de RSA se fait bien avec une espérance de nombre de tirages de 2. □

4.4.6 DLP et Chiffrement d'ElGamal

Un autre problème mathématique est réputé difficile : il s'agit du calcul du logarithme discret (Discret Logarithm Problem : DLP) qui utilise la fonction puissance modulo un nombre premier p (Voir §1.2.7, page 28). Si g est un générateur de $\mathbb{Z}_p^*$, cette fonction est définie par :

$$\begin{array}{rccc} F_g : & \mathbb{Z}_p^* & \longrightarrow & \mathbb{Z}_p^* \\ & x & \longrightarrow & g^x \mod p \end{array}$$

Etant donné x, il est facile de calculer $y = g^x \mod p$ (toujours par un algorithme de type exponentiation rapide).

Par contre, on ne connaît pas à ce jour de méthode rapide pour calculer $x = \log_g y \mod p$, i.e. le logarithme discret de y. En fait, si l'on savait calculer rapidement x, on aurait aussi un algorithme rapide pour factoriser un entier quelconque en produit de nombres premiers.

Cette constatation est la base du protocole d'échange de clé de Diffie-Hellman (§4.6.1, page 110), et de l'algorithme de chiffrement ElGamal qui est décrit maintenant.

Description de ElGamal dans $\mathbb{Z}_p^*$

- Soit p un nombre premier tel que le problème du logarithme discret est difficile dans $\mathbb{Z}_p^*$
- Soit $g \in \mathbb{Z}_p^*$ un élément primitif.
- Soit s un nombre et $\beta = g^s$
- La clé publique est alors le triplet $K_e = (p, g, \beta)$.
- La clé secrète est le nombre $K_d = s$.

Chiffrement : Soit M le texte clair à chiffrer et $k \in \mathbb{Z}_{p-1}$ un nombre aléatoire secret.

$$E_{K_e}(M) = (y_1, y_2) \text{ avec } \begin{cases} y_1 = g^k \bmod p \\ y_2 = M.\beta^k \bmod p \end{cases}$$

Déchiffrement : Pour $y_1, y_2 \in \mathbb{Z}_p^*$, on définit :

$$D_{K_d}(y_1, y_2) = y_2.(y_1^s)^{-1}$$

En effet, $y_2.(y_1^s)^{-1} = M.\beta^k.(g^{k.s})^{-1} = M.g^{s.k}.(g^{k.s})^{-1} = M$

Généralisation de DLP

Nous avons décrit ici le problème du logarithme discret dans le groupe multiplicatif du corps fini $\mathbb{Z}_p$.

En fait, on peut décrire ce problème pour tout groupe. De nombreux travaux portent actuellement sur la définition de nouveaux groupes dans lesquel le calcul du logarithme discret est difficile. C'est la cas par exemple du groupe additif d'une courbe elliptique sur un corps fini $\mathbb{Z}_p$.

4.5 Système hybride de clef publique/privée

Malgré la relative simplicité des fonctions D et E dans le cadre de RSA ou de ElGamal, elles ne sont pas aussi faciles à calculer que leurs équivalents pour le DES : l'exponentiation modulaire est une opération plus coûteuse en temps que des permutations ou des recherches dans une table lorsque p est grand. Cela rend plus compliqué la conception d'un composant électronique destiné à chiffrer et déchiffrer rapidement en utilisant cette approche.

C'est pourquoi, d'un point de vue performances, un algorithme de type RSA est environ 10,000 fois plus lent qu'un DES. Pour des clés de 512 bits l'ordre de grandeur de capacité de chiffrement avec la technologie actuelle est de 75Kbits/s, les 1Mbits/s sont envisageables à court ou moyen terme.

La solution pour un chiffrement efficace réside dans l'utilisation conjointe de systèmes de chiffrement symétrique et asymétrique. Ainsi, le système PGP[8] chiffre un message avec le protocole suivant :

1. Le texte source est chiffré avec une clé K de type IDEA ou DES
2. La clé K est chiffrée selon le principe RSA avec la clé publique du destinataire
3. Envoi du message composé de la clé K chiffrée par RSA et d'un texte chiffré selon un algorithme IDEA ou DES

[8]PGP (Pretty Good Privacy) est l'outil le plus utilisé pour le chiffrement de messages ou l'authentification par courrier électronique.

Le récepteur effectuera alors les opérations de déchiffrement suivantes :

1. Déchiffrement de la clé grâce à la clé privée D en utilisant RSA
2. Déchiffrement du texte en utilisant DES ou IDEA avec la clé K

Pour l'authentification le principe est un peu différent :
- Génération d'un code qui identifie, par un nombre de taille fixe, le texte. On utilise pour cela en pratique une fonction de hachage (voir §4.7.1)
- Chiffrement de ce code en utilisant RSA avec la clé privée de l'émetteur du message

Cela permet au correspondant de vérifier la validité du message en utilisant la clé publique de l'expéditeur.

Comme application du système PGP, citons le programme SSH utilisé en réseau pour se loger sur une machine tierce, faire des transferts de fichiers et lancer des applications. Ce protocole est plus amplement détaillé au §4.9.

4.6 Protocoles cryptographiques

4.6.1 Protocole d'echange de clés de Diffie-Hellman

L'utilisation d'une fonction à sens unique comme la fonction puissance modulo un nombre premier (définie au §4.4.6) permet le partage de clef. Ce fût en fait la première technique qui a été proposée pour construire un système de clef publique.

Alice et Bob veulent partager une clef secrète K. On suppose que Alice et Bob ont convenu d'un entier premier p et d'un générateur de $\mathbb{Z}_p^*$, g. On peut alors définir le protocole suivant en 3 étapes pour construire K en secret :

1. Seule, Alice choisit un nombre $a \in \mathbb{Z}_p^*$ secret ; elle calcule alors $A = g^a \bmod p$ et envoie A à Bob.
   Symétriquement, Bob choisit seul un nombre $b \in \mathbb{Z}_p^*$ secret ; il calcule alors $B = g^b \bmod p$ et envoie B à Alice.
2. Alice calcule seule $K = B^a \bmod p$. On a $K = g^{a.b} \bmod p$.
   Symétriquement, Bob calcule de son côté $A^b \bmod p$, qui vaut aussi $g^{a.b} \bmod p$.

L'évolution du protocole est illustré dans la table suivante :

Ainsi, Alice et Bob possèdent tous les deux en secret la même clef, sans se l'être jamais directement communiqué.

Exercice 8. *On suppose que Mata Hari voit passer A et B. Expliquer pourquoi Mata Hari ne peut alors pas facilement en déduire K.*

Exercice 9. *Alice et Bob conviennent des paramètres suivants : $p = 541$ et $g = 2$. Alice génère le nombre secret $a = 292$. De son coté, Bob génère le nombre $b = 426$.*

Quelle est la clé secrète résultant du protocole d'échange de Diffie-Hellman ?

Alice		Bob
génère a		génère b
$A = g^a \bmod p$		$B = g^b \bmod p$
	$\xrightarrow{A}$	
	$\xleftarrow{B}$	
(dispose de $[a, A, B, p]$)		(dispose de $[b, A, B, p]$)
Clé secrète : $K = B^a \bmod p$		Clé secrète : $K = A^b \bmod p$

TAB. 4.4 – Protocole d'échange de clé de Diffie-Hellman

Exercice 10. *On suppose qu'un intrus coupe la ligne de communication entre Alice et Bob avant qu'il ne commence le partage de la clef (émission de A et B).*
Expliquer comment cet intrus peut alors lire toutes les communications entre Alice et Bob avec la clef secrète qu'ils pensent avoir construit.

4.6.2 Confidentialité et authentification

Pour assurer la confidentialité, les transformations d'un système à clé publique doivent satisfaire $D(E(M)) = M$.

Supposons que A veuille envoyer un message secret M à B. A doit alors avoir accès à E_B, la fonction de transformation publique de B. A va chiffrer M, $\tilde{M} = E_B(M)$ et envoyer le résultat $\tilde{M}$ à B.
À la réception B va utiliser sa fonction privée de transformation D_B pour déchiffrer. Si la transmission de A est espionnée l'intrus ne pourra déchiffrer $\tilde{M}$ car la fonction D_B est secrète donc la confidentialité est assurée.

Par contre, comme E_B est publique, B n'a aucun moyen de connaître l'identité de l'envoyeur. De même le message envoyé par A peut être altéré. Les propriétés d'authentification et d'intégrité ne sont donc pas assurées.

Pour qu'elles le soient les transformations doivent satisfaire la propriété $E(D(M)) = M$. En effet, supposons que A veuille envoyer un message authentifié M à B. Cela signifie que B doit pouvoir vérifier que le message a bien été envoyé par A et qu'il n'a pas été altéré en chemin. Pour cela A va utiliser sa transformation privée D_A, calculer $M' = D_A(M)$ et envoyer M' à B. B peut alors utiliser la transformation publique E_A pour calculer $E_A(M') = E_A(D_A(M)) = M$.

En supposant que M représente un texte "valide" (au sens du protocole utilisé), B est sûr que le message a été envoyé par A et n'a pas subi d'altérations pendant le transport. Cela vient de la nature uni-directionnelle de

E_A : si un attaquant pouvait, à partir d'un message M, trouver M' tel que $E_A(M') = M$ cela signifierait qu'il peut calculer l'inverse de la fonction E_A ce qui est une contradiction. Par contre dans ce cas le secret n'est pas assuré car tout le monde peut accéder à E_A et donc déchiffer le message.

Exercice 11. *En utilisant les fonctions E_A, D_A, E_B et D_B trouver un couple de fonctions de chiffrement et déchiffrement satisfaisant à la fois le secret et l'authentification.*

4.7 Signature Numérique et Authentification

4.7.1 Notion de Fonction de Hachage

Une fonction de hachage H est une application qui transforme une chaîne de taille quelconque en une chaine de taille fixe n.

$$H : \{0,1\}^* \longrightarrow \{0,1\}^n$$

Cette fonction est donc surjective et on parle alors de *collision* entre x et x' lorsque

$$\begin{cases} x \neq x' \\ H(x) = H(x') \end{cases}$$

Si y est tel que $y = H(x)$, alors x est dit être la *préimage* de y.

Les propriétés de base d'une fonction de hachage sont la compression d'une part et la facilité de calcul d'autre part ; elles peuvent également vérifier les propriétés additionnelles suivantes :

- résistance à la préimage : étant donné y, il est calculatoirement difficile de trouver un x tel que $y = H(x)$;
- résistance à la seconde préimage : étant donné x, il est calculatoirement difficile de trouver $x' \neq x$ tel que $H(x) = H(x')$;
- résistance à la collision : il est calculatoirement difficile de trouver x et x' tels que $H(x) = H(x')$;

Une fonction de hachage à sens unique[9] est une fonction de hachage qui vérifie les propriétés additionnelles de résistance à la préimage et à la seconde préimage ;

On définit également les Collision Resistant Hash Function qui vérifient les propriétés additionnelles de résistance à la seconde préimage et à la collision.

Les fonctions de hachage peuvent être utilisées pour :

- les codes de détection de manipulation (MDC[10]) qui gèrent l'intégrité d'un message ;

[9] One-Way Hash Function
[10] Manipulation Detection Code

- les codes d'authentification de messages (MAC[11]) qui gèrent à la fois l'intégrité et l'authentification de la source d'une donnée (voir paragraphe suivant).

Les fonctions de hachage cryptographiques les plus connues sont [25], [13] et [10]. Ces fonctions possèdent les propriétés suivantes :

- sans clé : ce sont des fonctions publiques ;
- compression ;
- résistance aux collisions ;
- "randomness-like" : non prédictible, indépendance entrée/sortie.

Le principe général de fonctionnement est le suivant :

- on définit d'abord une fonction de compression h (cf fig 4.11) qui peut être plus ou moins compliquée.

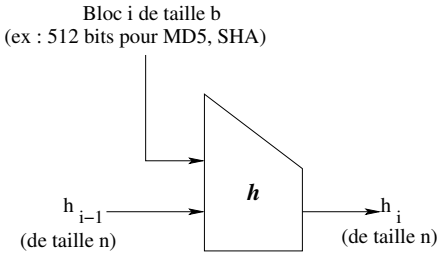

FIG. 4.11 – Fonction de compression d'une fonction de hachage

Si la taille de l'empreinte est n et celle des blocs b, cette fonction applique deux entrées (un bloc de b bit et la sortie précédente sur n bits) en une sortie de n bits.

- Supposons que l'on veuille calculer l'empreinte d'un message M. On commence par compléter le message M jusqu'à ce que sa taille soit un multiple de b et que le dernier bloc contienne la taille du message M (dans MD5 comme dans SHA, on ajoute un 1 puis autant de 0 que nécessaire pour laisser les 64 derniers bits coder la taille réelle de M avant complétion).
  On a donc : $M = M_1 M_2 ... M_{k-1} M_k$ ou M_k contient $|M|$.
  On itère alors la fonction h comme décrit dans la figure 4.12 :
  IV (Initial Value) est une chaîne (de taille n) fixée mais publique. Il existe un théorème (le théorème de [Merkle,Damgård]) qui nous assure que si h est résistante aux collisions, alors il en est de même pour H.

4.7.2 Fiabilité des fonctions de hachage

La résistance à la collision des fonctions de hachage peut se mesurer : il faut déterminer la probabilité d'obtenir des collisions. Le paradoxe des

[11] Message Authentification Code

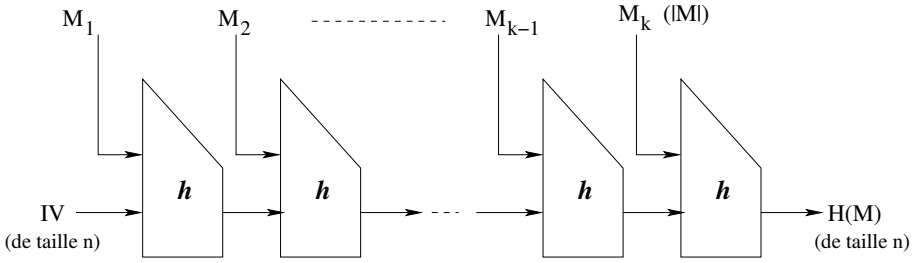

FIG. 4.12 – Itération de la fonction de compression pour le calcul d'empreinte

anniversaires va nous servir : dans un groupe de plus de 23 personnes il y a plus d'une chance sur deux qu'au moins deux d'entre elles aient le même anniversaire ! En effet, prenons une population de k personnes, sachant que le nombre de jours dans l'année est n, le nombre de combinaisons de k anniversaires différents est $A_n^k = \frac{n!}{n-k!}$. Donc la probabiblité que toutes les personnes aient des anniversaires différents est $\frac{A_n^k}{n^k}$. Donc la probabiblité pour que deux personnes au moins aient leur anniversaire le même jour est

$$1 - \frac{A_n^k}{n^k}.$$

Ainsi, en considérant 365 jours, cette probabilité est environ d'une chance sur 10 dans un groupe de 9 personnes, de plus d'une chance sur deux dans un groupe de 23 personnes et de 99.4% dans un groupe de 60 personnes. Ce paradoxe peut-être utilisé pour faire des collisions de fonctions de hachage, avec l'attaque par anniversaires de Yuval :
Entrées : x_1 légitime, x_2 frauduleux, une fonction de hachage h, sur m bits.
Sorties : $x_1' \approx x_1$ et $x_2' \approx x_2$ tels que $h(x_1') = h(x_2')$.

1. Générer $t = 2^{\frac{m}{2}} = \sqrt{2^m}$ modifications mineures de x_1, notées x_1'.
2. Pour tout t, calculer $h(x_1')$.
3. Générer des x_2', modifications mineures de x_2, jusqu'à collision avec un x_1'.

Théorème 18. *Dans un ensemble de $\lceil 1.18\sqrt{n} \rceil$ éléments choisis aléatoirement parmi n possibilités, la probabilité de collision est supérieure à 50%.*

Preuve. Nous avons vu que le nombre de collisions sur un espace de taille $n = 2^m$, avec k tirages est $1 - \frac{A_n^k}{n^k}$. Il faut estimer cette probabilité : $1 - \frac{A_n^k}{n^k} = 1 - (1 - \frac{1}{n})(1 - \frac{2}{n}) \ldots (1 - \frac{k-1}{n})$. Or, $1 - x < e^{-x}$, pour x positif, donc

$$1 - \frac{A_n^k}{n^k} > 1 - \prod_{i=1}^{k-1} e^{-\frac{i}{n}} = 1 - e^{-\frac{k(k-1)}{2n}}$$

Alors, pour que cette probabilité vaille α, il faut que $k(k-1) = 2nln(1-\alpha)$, soit, comme k est positif, $k = \frac{1}{2} + \sqrt{n}\sqrt{\frac{1}{4n} - 2ln(1-\alpha)}$. Ainsi, pour $\alpha = 0.5$, $k \approx 1.17741\sqrt{n}$ (on retrouve pour $n = 365$, $k \approx 22.4943$). □

Corollaire 4. *L'espérance du nombre de tirages de x'_2 dans l'attaque de Yuval est en $O(t) = O(\sqrt{2^m})$.*

Une application de l'attaque de Yuval est donc d'envoyer x'_1, puis de la répudier plus tard en soutenant que x'_2 avait en fait été envoyé !

Est-ce que cette attaque est réalisable ? Un calcul simple permet de s'en convaincre : pour une empreinte numérique sur 128 bits, il faudra donc effectuer de l'ordre de $O(2^{64})$ calculs, soit 2.10^9. Par ailleurs, en 2004, 2.5 GHz peuvent coûter 1 k€ et un jour compte $24 * 3600 \approx 10^5$ secondes. Donc 8000 k€ suffisent pour trouver une collision en 10 jours. Il faut compter en outre que ce coût, sans doute assez surévalué, est divisé par 2 environ tous les 18 mois. Enfin, si l'on utilise des fonctions de hachage sur 160 bits, ce coût est multiplié par $2^{16} = 64000$, ce qui reste encore aujourd'hui assez inatteignable.

4.7.3 Les MAC - Message Authentication Code

On peut utiliser les fonctions de hachage pour faire à la fois de l'authentification et du contrôle d'intégrité de message !

On utilise pour cela un MAC : c'est une fonction de hachage à sens unique paramétrée avec une clé secrète K (notée $H_K()$) qui respecte les propriétés de compression, de facilité de calcul (c'est une fonction de hachage) et enfin :

$\forall K$ inconnu et étant donné la suite $\begin{cases} (x_1, H_K(x_1)) \\ (x_2, H_K(x_2)) \\ \vdots \\ (x_i, H_K(x_i)) \end{cases}$, il est calculatoirement infaisable de calculer une paire $(x, H_K(x))$ avec $x \notin \{x_1, x_2, ..., x_i\}$.

Pour résumer, seul celui qui possède la clé K peut vérifier l'empreinte. Les MAC permettent donc de prouver l'authenticité sans fournir la confidentialité. Le principe de fonctionnement d'un MAC est exposé dans la figure 4.13.

Remarque : si le MAC fournit l'authentification et l'intégrité, ce n'est ni une signature, ni un certificat de non-répudiation !

En pratique, on cherche à rendre le cassage du MAC aussi difficile que le cassage de la fonction de hachage elle-même.

Nous avons vu précedemment (figure 4.12) le principe général de construction des fonctions de hachage les plus connues (MD5-like).

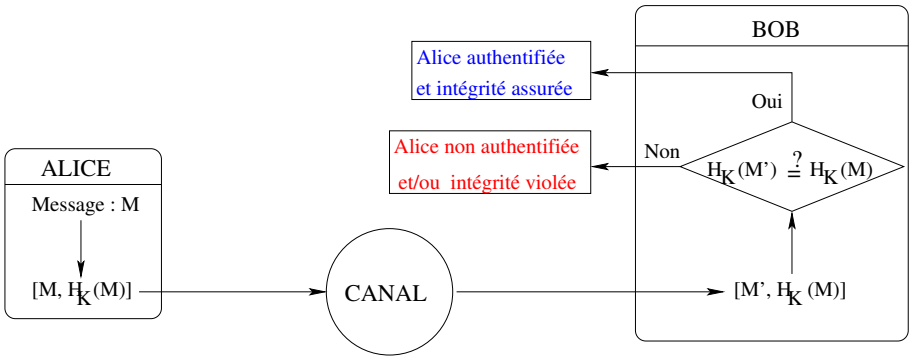

FIG. 4.13 – Principe d'utilisation d'un MAC

Mais comment utiliser de telles fonctions (qui sont sans clé) pour réaliser un MAC ?

On peut envisager en fait les constructions suivantes ($A \cdot B$ represente la concaténation des chaînes A et B) :

1. *méthodes "in-text" :*
   - $H(K \cdot M)$ (dite "prepend-only")
   - $H(M \cdot K)$ ("append-only")
   - $H(K_1 \cdot M \cdot K_2)$
2. via IV : on pose IV=K avant d'itérer la fonction de compression.
3. On peut enfin envisager une méthode hybride : $H_{K_1}(x \cdot K_2)$

Exercice 12. *Soit M un message clair auquel on associe un MAC issu d'une construction de type "prepend-only". Expliquez comment Oscar peut facilement générer un message $\tilde{M}$ ayant une empreinte MAC valide.*

4.7.4 Les signatures numériques

Les signatures manuscrites ont été longtemps utilisées pour prouver l'identité de leur auteur ou du moins l'accord du signataire avec le contenu du document. Avec des documents numériques, les objectifs d'une signature sont les suivants :

- Une signature est authentique. Elle convainc le destinataire que le signataire a délibérément signé le document.
- Une signature ne peut être falsifiée (imitée). Elle est la preuve que le signataire a délibérément signé le document.
- Une signature n'est pas réutilisable. Elle fait partie du document et une personne mal intentionnée ne peut pas déplacer la signature sur un autre document.

- Un document signé est inaltérable. Une fois le document signé, il ne peut plus être modifié.
- Une signature ne peut pas être reniée. La signature et le document sont des objets physiques et le signataire ne peut prétendre plus tard ne pas avoir signer le document.

Plusieurs solutions sont envisageables :

- signature à l'aide d'un cryptosystème à clef secrète.
- signature à l'aide d'une fonction de hachage et d'un cryptosystème à clé secrète ou publique ([28] p.40).

Pour la première solution, il faut soit un arbitre ([28] p.38-39), soit qu'une clef secrète soit partagée. Dans ce dernier cas, une idée est de modifier le mode de chiffrement CBC pour associer les blocs les uns après les autres comme sur la figure 4.14. Sur un pentium III cadencé à 500 MHz, une im-

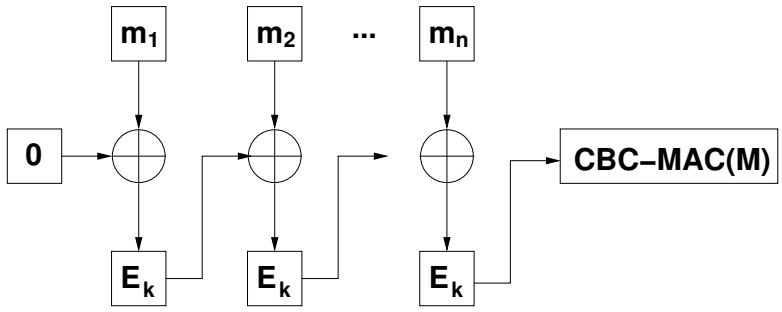

FIG. 4.14 – Signature CBC-MAC

plémentation logicielle du CBC-MAC-AES (c'est-à-dire un CBC-MAC où les chiffrements E_k sont réalisés par l'algorithme Rijndaël, standardisé sous le nom AES) peut produire des signatures de message à la vitesse de 234 Mbits/seconde. La même implémentation de AES seul tourne à une vitesse de 275 Mbits/seconde. Le coût de calcul d'une signature est donc ainsi très voisin de celui du cryptage.

La deuxième famille de signatures utilise des fonctions de hachage. La première variante est d'utiliser une clef secrète. L'idée la plus simple est de hacher le message avec une fonction h, mais d'ajouter une partie secrète au message, par exemple une clef ! C'est le principe de la signature [2] :

$$HMAC_K(M) = H[K||pad_1||H(K||pad_2||M)]$$

Ainsi, le MD5-MAC (HMAC où la fonction de hachage est MD5) est seulement de 5 à 20 % plus lent que MD5 seul. En effet, le deuxième calcul de h ne se fait que sur un petit mot (taille de la clef + pad_1 + taille d'un hash).

La deuxième variante est d'utiliser une fonction de hachage avec un système à clef publique. Le principe général est illustré dans la figure 4.15. Cette

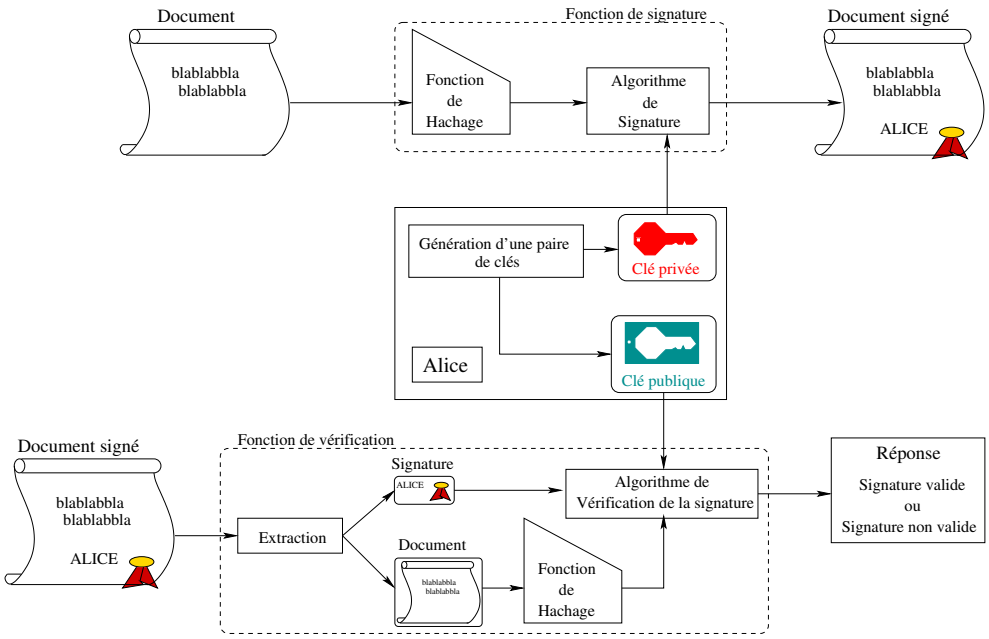

FIG. 4.15 – Principe de la signature à clef publique et de sa vérification

figure n'explicite poutant pas quels sont les fameux algorithmes de signature et de vérification.

En fait, on en dénombre principalement deux à l'heure actuelle qui font l'objet des paragraphes suivants.

Signature RSA

Il est en effet possible d'utiliser le système RSA pour définir une signature digitale. Il suffit pour cela d'appliquer RSA à l'envers ! (Ce principe a déjà été exposé dans la section 4.6.2).

Prenons (e, n) la clef publique d'Alice, et d son exposant secret. Alors Alice peut signer, par exemple de son nom m, en envoyant le message $s = m^d \mod n$. Tout récipiendaire peut alors vérifier que le message a bien été ecrit par Alice en appliquant sa clef publique : Si s est bien une signature d'Alice, alors $s^e \mod n$ doit correspondre à un message en clair, en l'occurence m. En outre, puisque seule Alice est sensée connaître d, seule Alice a pu composer la signature s.

Pour assurer également la confidentialité, il suffit d'appliquer le schéma suivant :

- Alice produit sa signature s, elle la concatène à son message x, pour donner $x \cdot s$, elle encode ensuite $x \cdot s$ avec la clef publique (f, b) de Bob, par $(x \cdot s)^f \mod b = y$.
- De son coté, Bob reçoit y, il le décode avec sa *clef secrète* g : $y^g \mod b = x \cdot s$. Il extrait s, et vérifie que c'est bien Alice qui lui a envoyé ce message avec la *clef publique* $s^e \mod n = m$.

Pour plus de détail, on se réferera à [28], page 491-493.

DSS - Digital Signature Standard

Ce schéma de signature est basé sur DLP et implémente le standard DSA[12]. Le système comprend des paramètres partagés q, p et g ainsi qu'une clef publique y et une clef privée x :

1. Choisir q premier de 160 bits.
2. Trouver $p = kq + 1$ premier de 512 à 1024 bits.
3. Tant que $a == 1$ choisir g et calculer $a \equiv g^{\frac{p-1}{q}}[p]$. *($g$ est alors d'ordre au plus q).*
4. Choisir x de 160 bits.
5. $y \equiv g^x[p]$.

Une fois les paramètres construits, la signature se fait de la façon suivante :

1. Alice choisit k aléatoire inférieur à q.
2. Alice calcule et envoie la signature de son message m à l'aide d'une fonction de hachage h (pour le DSS, il s'agit de SHA-1) :

$$r = (g^k \mod p) \mod q \quad et \quad s = (k^{-1}[h(m) + xr]) \mod q$$

3. Bob vérifie la signature si et seulement si $v == r$ pour :
   - $w \equiv s^{-1}[q]$;
   - $u_1 \equiv h(m)w[q]$;
   - $u_2 \equiv rw[q]$;
   - $v = ([g^{u_1} y^{u_2}] \mod p) \mod q$;

Théorème 19. *La vérification est correcte*

Preuve. s est construit de sorte que $k = (s^{-1}[h(m) + xr]) \mod q$. Or $g^{u_1} y^{u_2} \equiv g^{u_1 + xu_2}$ donc, d'après la donnée de u_1 et u_2, et comme $g^q \equiv 1[p]$ par construction, on a donc $g^{u_1} y^{u_2} \equiv g^{h(m)w + xrw}$. Or, comme $w \equiv s^{-1}$, la remarque de début de preuve nous donne $g^{u_1} y^{u_2} \equiv g^k$ ou encore $v \equiv r[q]$. □

Plus de détails dans [12], [32] page 190-193 et [28] page 509-520.

[12] Digital Signature Algorithm

Exercice 13. *Parallèlement au DSS, les russes avaient développé leur propre signature : GOST [28]. Les paramètres p, q, g, x et y sont les mêmes sauf que q est premier de 254 à 256 bits et p premier entre 509 et 512 bits (ou entre 1020 et 1024 bits). Seule la partie s de la signature change :*

1. [...]
2. *Alice calcule et envoie*

$$r = (g^k \mod p) \mod q \quad et \quad s = (xr + kh(M)) \mod q$$

3. *Bob vérifie la signature si et seulement si $u == r$ pour :*
   - $v = h(M)^{q-2} \mod q$.
   - $z_1 = sv \mod q$
   - $z_2 = ((q - r)v) \mod q$
   - $u = ((g^{z_1} y^{z_2}) \mod p) \mod q$

Justifier la vérification.

4.8 Architecture PKI

Une Architecture PKI[13], est un ensemble d'infrastructures permettant de réaliser effectivement des échanges sécurisés. En effet, une fois définis des algorithmes complexes utilisant par exemple des clefs publiques, le premier problème pratique qui se pose est "comment rattacher une clef publique à son propriétaire ?" L'idée des PKI est d'abord de ne pas distribuer des clefs mais plutôt des certificats numériques contenant ces clefs ainsi que des données d'identité (état civil, adresse, courriel pour une personne, ou encore nom de domaine, adresse IP pour un serveur ...) ; ensuite, les PKI sont des structures précises assurant en particulier la création et la gestion de ces certificatts.

4.8.1 Principe général

Pour s'authentifier, une entité cherche à prouver qu'elle possède une information secrète qui ne peut être connue que par elle seule.

Dans le cas d'une PKI, une entité (par exemple Alice) génère un couple clé publique/clé privée. Alice conserve sa clé privée (soit dans un répertoire accéssible par elle seule ou mieux sur une carte à puce ou une clé USB etc.).

Ensuite, il s'agit de rendre sa clé publique accessible à toute personne désireuse d'effectuer une transaction avec elle. Pour cela, Alice va utiliser un *tiers de confiance* (un CA : voir paragraphe suivant) pour créer un *certificat*. Un certificat est un document numérique qui contient en gros toutes les coordonnées d'Alice, ainsi que sa clé publique. Ce document est signé par le CA. Cette signature sert à certifier l'origine du certificat ainsi que son intégrité.

[13] Public Key Infrastructure

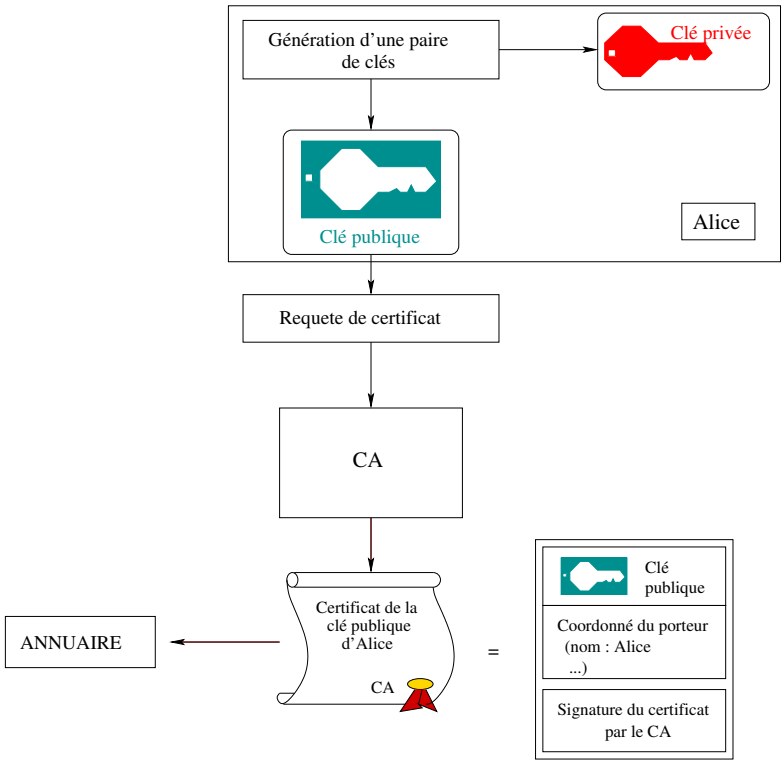

FIG. 4.16 – Principe de la création des certificats

La figure 4.16 illustre les étapes de la création/distribution d'un certificat. Il reste encore à savoir comment vérifier l'authenticité du tiers de confiance !En pratique, ce tiers possède également un cerificat signé par un autre tiers, etc. Le dernier de cette *chaîne de certificats* devra alors se signer lui-même. Nous parlerons de certificat auto-signé ou certificat racine. Pour augmenter la confiance, ces certificats racines pourront également faire l'objet de certifications croisées avec d'autres CA.

Une PKI est donc un ensemble de technologies, organisations, procédure et pratiques qui supporte l'implémentation et l'exploitation de certificats basés sur la cryptographie à clé publique. C'est donc un ensemble de systèmes fournissant des services relatifs aux certificats numériques.

Une PKI fournit donc les moyens d'utiliser ces certificats pour l'authentification entre entités. Il existe un standard pour ce mécanisme qui est décrit dans [14] et sera exposé dans la subsection 4.8.4.

4.8.2 Les éléments de l'infrastructure

Les fonctions d'une PKI

– émettre des certificats à des entités préalablement authentifiées ;

- révoquer des certificats, les maintenir ;
- établir, publier et respecter des pratiques de certification pour établir un espace de confiance ;
- rendre les certificats publics par le biais de services d'annuaires ;
- éventuellement, gérer les clés et fournir des services d'archivage. Dans ce cas les services à rendre aux utilisateurs sont de trois types :

1. *Gestion des clefs* (Key management). Une architecture peut assurer la gestion de la création, la distribution, le stockage, l'utilisation, le recouvrement, lm'archivage et la destruction des clefs (couples publics/privés, où clef secrète de chiffrement symétrique). Pour cela un certain nombres de règles de base, les *axiomes de gestion de clefs*, doivent être impérativement respectées :
   [Axiome 1] *Les clefs secrètes doivent l'être et le rester.* En pratique, personne, sauf les possésseurs légitimes, ne doit y avoir accès ; la sécurité n'étant pas basée sur l'ignorance des cryptosystèmes mais sur celle des clefs, il est plus facile d changer un clef compromise qu'un algorithme.
   [Axiome 2] *Les clefs secrètes doivent exister seulement* :
   - En clair, à l'intérieur d'un module résistant (TRSM, Tamper Resistant Security Module).
   - Chiffrées ou au moins en morceaux à l'extérieur.

   [Axiome 3] *Limiter le déploiement des clefs* : les clefs doivent se trouver en un nombre minimal d'endroits pour limiter les expositions et les risques.
   [Axiome 4] *Séparer les clefs* : les clefs doivent être créées et utilisées pour un seul objectif, le raffinemenbt dépendant de la politique de sécurité.
   [Axiome 5] *Synchroniser les clefs* : il faut vérifier que toute clef a été employée sans risque pour la sécurité des autres clefs.
   [Axiome 6] *Journal d'évênements* : Tous les évênements de gestion de toutes les clefs sont transcrits dans un journal qui est lui-même géré de manière sûre. Un évênement renferme, une date, un type (création, accès à un local, ...), le personnel/système concerné et le résultat de l'action (succès, autorisé, non-autorisé, ...).

2. *Mise en commun des clefs* (Key agreement). Cela est réalisé par exemple par lme protocole de Diffie et Hellman de la subsection 4.6.1.
3. *Transport des clefs* (Key transport). Cela est réalisé par exemple par RSA et illustré sur la figure 4.17

Les acteurs d'une PKI On distingue différentes entités au sein d'une PKI :

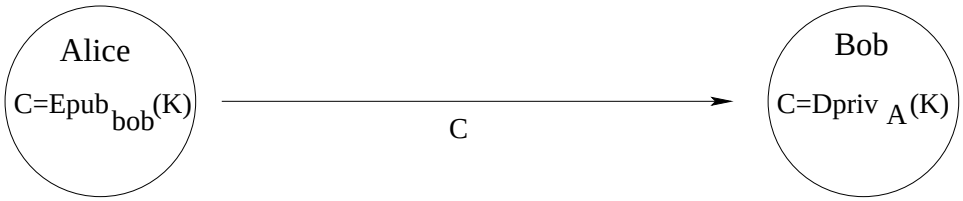

FIG. 4.17 – Transport de clefs utilisant la Cryptographie à Clé Publique

- Le *détenteur d'un certificat* : c'est une entité qui possède une clé privée et est le sujet d'un certificat numérique contenant la clé publique correspondante. Il peut s'agir d'une personne physique (certificat client), d'un serveur Web (certificat serveur), d'un equipement réseau (certificat VPN) etc.
- L'*utilisateur d'un certificat* : celui ci récupère le certificat et utilise la clé publique qu'il contient dans sa transaction avec le détenteur du certificat.
- L' *Autorité de Certification* (CA[14]) : c'est un ensemble de ressources (logicielles et/ou matérielles) et de personnels défini par son nom et sa clé publique qui :
  - génère des certificats ;
  - émet et maintient les informations sur les Listes de Révocation des Certificats (CRL[15]) ;
  - publie les certificats non encore expirés ;
  - maintient les archives concernant les certificats expirés et révoqués.

  C'est également l'entité juridique et morale d'une PKI.
- L' *Autorité d'enregistrement* (RA[16]) : c'est l'intermédiaire entre le détenteur de la clé et le CA. Il vérifie les requêtes des utilisateurs et les transmet au CA (le niveau de vérification dépend de la politique de sécurité mise en oeuvre). Chaque CA a une liste de RA accrédités. Un RA est connu d'un CA par son nom et sa clé publique. Le CA vérifie les informations fournies par un RA par le biais de sa signature.
- [L' *émetteur de CRL*] : l' émission de listes de révocation peut être déléguée hors du CA à une entité spécialisée.
- le *Dépôt* ou *Annuaire* (Repository) qui se charge quand à lui de :
  - distribuer les certificats et les CRL.
  - accepter les certificats et les CRL d'autres CA et les rend disponibles aux utilisateurs.

  Il est connu par son adresse et son protocole d'accès.
- L'*Archive* se charge du stockage sur le long terme des informations

[14]Certification Authority
[15]Certification Revocation List
[16]Registration Authority

pour le compte d'un CA. Cela permet de régler les litiges en sachant quel certificat était valable à telle époque.

De cette organisation il ressort que l'élément central d'une PKI est l'autorité de Certification. Une question qui vient donc est : "Pourquoi faut-il des autorités d'enregistrement ?". Deux types de facteurs interviennent :

1. Des éléments techniques.
   - Un certain nombre d'algorithmes, efficaces, sont nécessaires à la création, la gestion des clefs. Un RA peut fournir ces implémentations, logicielles ou matérielles aux utilisateurs qui n'en disposent pas.
   - Les utilisateurs ne sont pas forcément capables de publier leurs certificats.
   - Le RA peut émettre une révocation *signée* même si un utilisateur a perdu toutes ses clefs (le RA est lui sensé ne pas perdre les siennes !).

   Le RA est donc un intermédiaire sûr.
2. Une organisation simplifiée.
   - Il est moins cher d'équiper un RA que tous les utilisateurs.
   - Le nombre de CA nécessaires est réduit par le regroupement de fonctions simples.
   - La proximité avec les utilisateurs est augmentée.
   - Souvent des structures *pré-existantes* peuvent faire office de RA immédiatement.

Cependant, séparer le CA et le RA est en général moins sûr que de centraliser le tout dans un CA : en effet, un utilisateur et un RA peuvent se mettre d'accord, *mais* le RA transmet ensuite au CA. Le CA, ayant la décision finale, peut donc falsifier les informations. La sécurité reste donc dans la confiance en le CA.

4.8.3 Les certificats

Emission d'un certificat La figure 4.18 résume les différentes étapes liées à l'émission d'un certificat.

Une fois qu'un certificat a été émis et déposé dans un annuaire, les autres utilisateurs peuvent venir interroger l'annuaire pour récupérer des certificats ou consulter les CRL.

Il existe en fait plusieurs normes pour les PKI, la plupart en cours d'évolution. Des exemples d'infrastructures à clé publiques faisant actuellement l'objet d'une normalisation à l'IETF sont PKIX (Public Key Infrastructure X.509), SPKI (Simple Public Key Infrastructure) et DNSSEC (Domain Name System Security).

Dans la suite, le principal format utilisé pour les certicats est présenté : la norme X.509 [22].

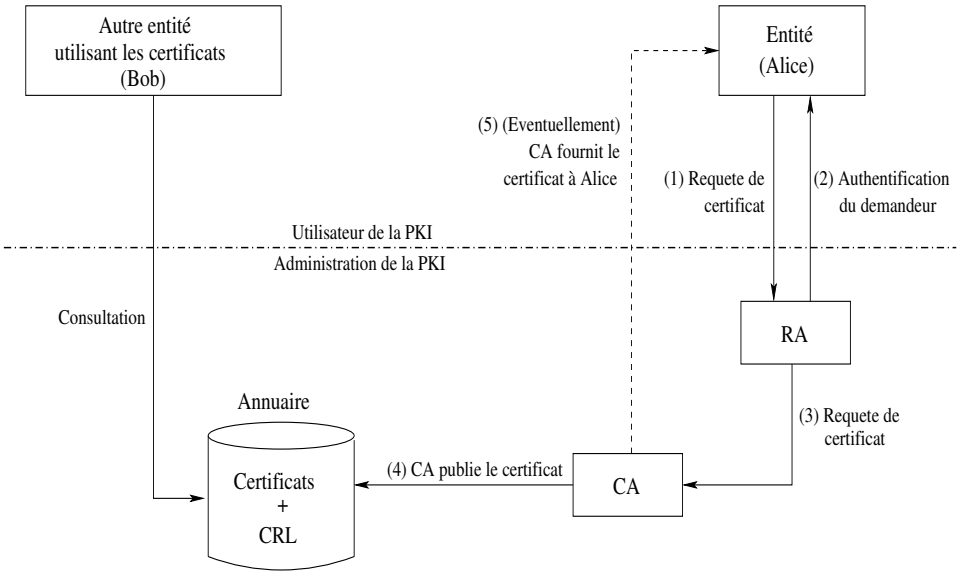

FIG. 4.18 – Emission d'un certificat

Format d'un certificat : la norme X.509 La figure 4.19 donne l'allure d'un certificat au format X.509. Cette figure illustre par des exemples la valeur des principaux champs qui composent ce certificat.

On notera sur les exemples de "Distinguished Name" (DN) (champs "Issuer name" et "Subject Name") la liste des sous-champs qui composent ces DN. Ils permettent de localiser très précisément les entités associées.

Le tableau 4.5 donne la signification des différents champs d'un certificat X.509.

Version :	Indique à quelle version de X.509 correspond ce certificat.
Serial number :	Numéro de série du certificat(propre à chaque CA).
Signature Algo ID :	Identifiant du type de signature utilisée.
Issuer Name :	Distinguished Name(DN) du CA qui emet le certificat
Validity period :	Période de validité.
Subject Name :	Distinguished Name (DN) du détenteur de la clé publique
Subject pub. key info :	Informations sur la clef publique de ce certificat.
Issuer Unique ID :	Identifiant unique de l'emetteur de ce certificat
Subject Unique ID :	Identifiant unique du détenteur de la clé publique
Extensions :	Extensions génériques optionnelles.
Signature :	Signature numérique du CA sur les champs précédents

TAB. 4.5 – Signification des champs d'un certificat X.509

Cette norme a connu plusieurs versions successives motivées par des retours d'expériences (v1 : 1988, v2 : 1993, v3 : 1996) . Elle précise également

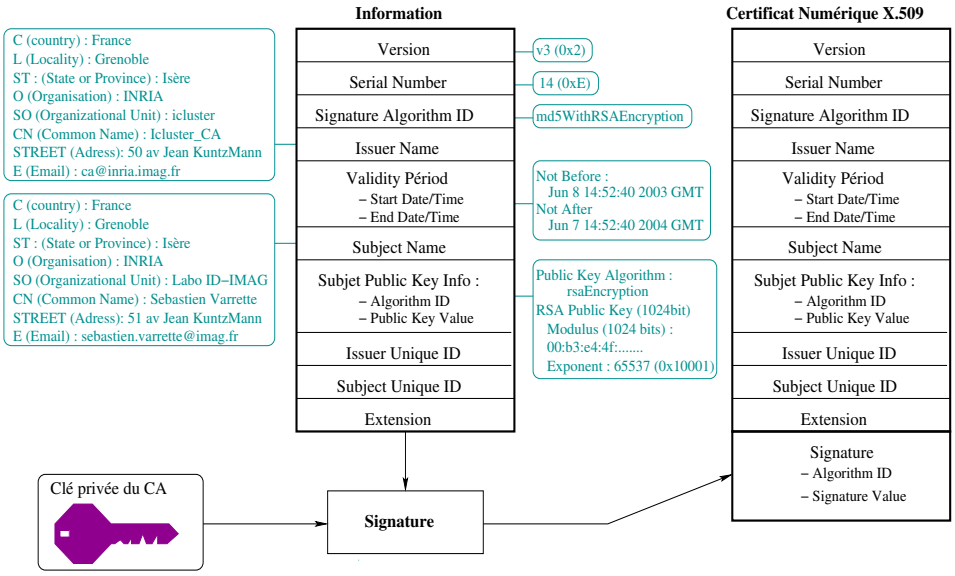

FIG. 4.19 – Génération et contenu d'un certificat X.509

le format des Listes de Révocations de certificats. Celui-ci est explicité dans la figure 4.20.

Les certificats X-509 sont écrits dans un langage de spécifications, ASN.1 (Abstract Syntax Notation). Les caractéristiques principales de ce langage sont les suivants :

- les commentaires commencent par "–" jusuq'à la fin de la ligne.
- les mots clefs ASN.1 sont en MAJUSCULES.
- les noms de champs commencent par une minuscule.
- les noms de types de données commencent par une majuscule.
- le symbole " :=" représente la définition du type d'une donnée.
- les blocs de données sont encadrés par des accolades.

Parmi les types principaux ASN.1 utilisés pour les certificats X-509, on peut trouver :

- "OID" : (object identifier) des nombres séparés par des points. Le département de la défense des USA est par exemple désigné par {1.3.6}.
- "AlgorithIdentifier" : un nom descriptif d'algorithme ; la table 4.6 présente les principaux.
- "Directory String" : information en texte.
- "Distinguished Names", "General names", "Time" : par exemple les dates sont définies par "Time :=CHOICE { UTCTime, GeneralizedTime }", où "UTCTime" est valide entre 1950 et 2049 et le "GeneralizedTime" après 2050.

L'extrait suivant présente une partie de la spécification ASN.1 des certificats X-509. Au vu de la figure 4.19, cette spécifaction est quasiment limpide :

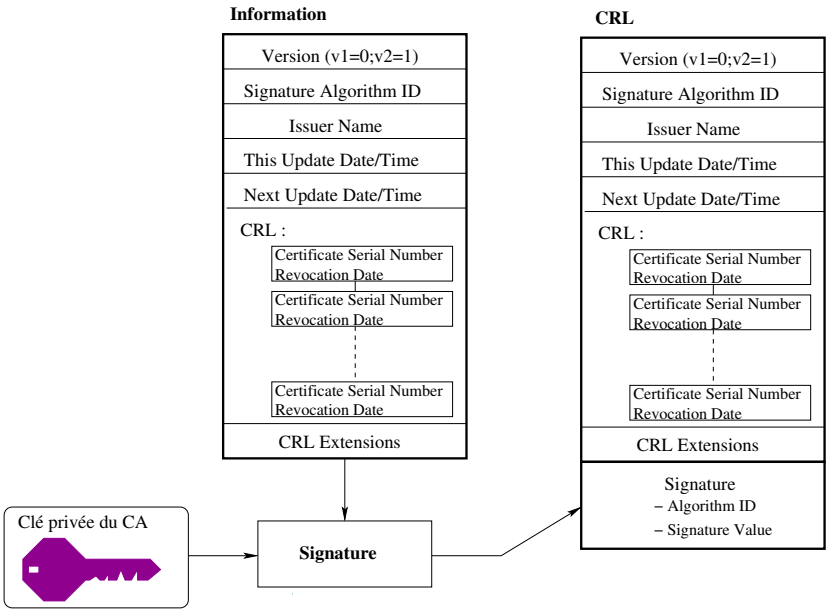

FIG. 4.20 – Génération et contenu d'une CRL

```
Certificate  ::=  SEQUENCE  {
     tbsCertificate       TBSCertificate,
     signatureAlgorithm   AlgorithmIdentifier,
     signatureValue       BIT STRING  }

TBSCertificate  ::=  SEQUENCE  {
     version         [0]  EXPLICIT Version DEFAULT v1,
     serialNumber         CertificateSerialNumber,
     signature            AlgorithmIdentifier,
     issuer               Name,
     validity             Validity,
     subject              Name,
     subjectPublicKeyInfo SubjectPublicKeyInfo,
```

Alg.	Hash.	OID	Identificateur
3DES-CBC		1.2.840.113549.3.7	DES-EDE3-CBC
RSA		1.2.840.113549.1.1.1	RSAEncryption
RSA	MD5	1.2.840.113549.1.1.4	md5withRSAEncryption
RSA	SHA-1	1.2.840.113549.1.1.5	sha1withRSAEncryption
DSA		1.2.840.10040.4.1	id-dsa
DSA	SHA-1	1.2.840.10040.4.3	id-dsawithSha1
DSA	SHA-1.320	1.3.14.3.2	id-dsawithSha1.320
ECDSA	SHA-1	1.2.840.10045.1	ecdsawithSha1

TAB. 4.6 – Quelques identificateurs d'algorithmes et numéros associés

```
        issuerUniqueID  [1]  IMPLICIT UniqueIdentifier OPTIONAL,
                             -- If present, version MUST be v2 or v3
        subjectUniqueID [2]  IMPLICIT UniqueIdentifier OPTIONAL,
                             -- If present, version MUST be v2 or v3
        extensions      [3]  EXPLICIT Extensions OPTIONAL
                             -- If present, version MUST be v3
        }
-----------------------------------------------------------------------
   Version  ::=  INTEGER  {  v1(0), v2(1), v3(2)  }
-----------------------------------------------------------------------
   CertificateSerialNumber  ::=  INTEGER
-----------------------------------------------------------------------
   AlgorithmIdentifier ::= SEQUENCE{
         algorithm       OBJECT IDENTIFIER,
         parameters      ANY DEFINED BY algorithm OPTIONAL}
-----------------------------------------------------------------------
   Validity ::= SEQUENCE {
        notBefore      Time,
        notAfter       Time }

   Time ::= CHOICE {
        utcTime        UTCTime,
        generalTime    GeneralizedTime }
-----------------------------------------------------------------------
   UniqueIdentifier  ::=  BIT STRING
-----------------------------------------------------------------------
   SubjectPublicKeyInfo  ::=  SEQUENCE  {
        algorithm            AlgorithmIdentifier,
        subjectPublicKey     BIT STRING  }
-----------------------------------------------------------------------
   Extensions  ::=  SEQUENCE SIZE (1..MAX) OF Extension

   Extension  ::=  SEQUENCE  {
        extnID      OBJECT IDENTIFIER,
        critical    BOOLEAN DEFAULT FALSE,
        extnValue   OCTET STRING  }
-----------------------------------------------------------------------
   Name ::= CHOICE {
        rdnSequence RDNSequence }

   RDNSequence ::= SEQUENCE OF RelativeDistinguishedName

   RelativeDistinguishedName ::= SET SIZE (1..MAX) OF AttributeTypeAndValue

   AttributeTypeAndValue ::= SEQUENCE {
     type     AttributeType,
     value    AttributeValue }

   AttributeType ::= OBJECT IDENTIFIER

   AttributeValue ::= ANY DEFINED BY AttributeType
```

4.8.4 L'administration

Le modèle PKIX Le groupe de travail PKIX a été établi à l'automne 1995 dans le but de développer les standards Internet nécessaires à une infrastructure reposant sur les certificats X-509. La première norme produite, le RFC-2459, décrit la version 3 des certificats X-509 et la version 2 des CRL. Le RFC-2587 sur le stockage des CRL, le RFC-3039 sur la politique de certification et le RFC-2527 sur le cadre pratique de certifications ont suivi. Enfin, le protocole de gestion des certificats, (Certificate Management Protocol : RFC-2510), le statut des certificats en-ligne, (Online Certificate Status Protocol : RFC-2560), le format de demande de gestion de certificat (Certificate Management Request Format : RFC 2511), la datation des certificats (Time-Stamp Protocol : RFC-3161), les messages de gestion de certificats, (Certificate Management Messages : RFC-2797) et l'utilisation de FTP et HTTP pour le transport des opérationsd PKI (RFC-2585) sont les autres standards mis au point par le groupe.

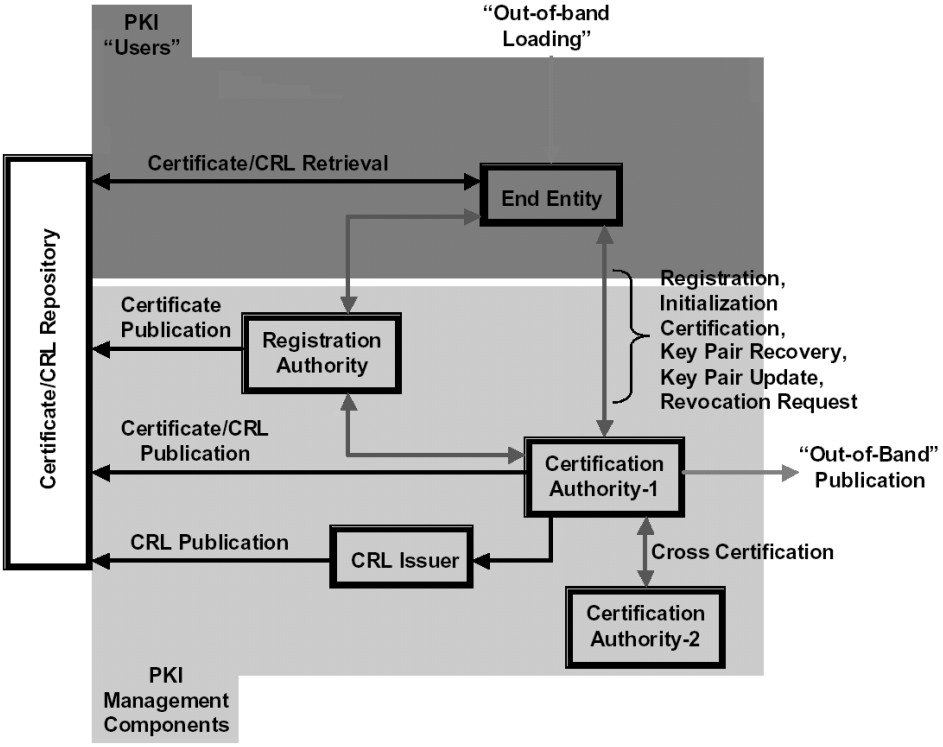

FIG. 4.21 – Le modèle PKI pour les certificats X-509

Au total, l'ensemble des fonctions d'administration d'une infrastructure à clef publique a été défini, la figure 4.21 récapitulant les structures retenues par le groupe. Nous en décrivons ci-après les principales règles décrites en

détail dans le RC-2510 :
[Axiome 1] Il y a une norme (ISO 9594-8) à respecter.
[Axiome 2] Il doit être possible de mettre à jour régulièrement une paire de clefs sans modifier les autres paires.
[Axiome 3] L'utilisation de la confidentialité dans les fonctions d'administration doit être minimale pour faciliter la normalisation.
[Axiome 4] L'utilisation de différents algorithmes de cryptographie doit être garantie (RSA, DSA, MD5, SHA-1, etc.) ; chaque CA, RA ou utilisateur doit pouvoir, en principe, utiliser l'algorithme le plus adapté à ses besoins.
[Axiome 5] La possibilité de générer des clefs doit pouvoir être laisséee aux utilisateurs.
[Axiome 6] La possibilité de publier des certificats doit pouvoir être laissée aux utilisateurs, aux RA et aux CA.
[Axiome 7] Les CRL doivent être accessibles directement aux utilisateurs. Les "Denial-of-Services" qui en résulteraient éventuellement ne doivent pas être facilités.
[Axiome 8] Les mécanismes de transport les plus variés (courriel, http, ftp, TCP/IP) doivent être utilisables.
[Axiome 9] L'autorité finale de création de certificat reste le CA suivant sa propre politique de sécurité, en particulier il peut ne pas répondre exactement aux requêtes.
[Axiome 10] La possibilité de migrer d'un CA non compromis à un autre de manière transparente pour les utilisateurs doit être prévue (i.e. les clefs restent valides).
[Axiome 11] Un CA peut assurer les fonctions de RA.
[Axiome 12] Un utilisateur doit être prêt à prouver qu'il détient bien la clef privée correspondant à un certificat donné.

Les fonctions d'administration Dans ce cadre, les fonctions d'administration au sein d'une PKI sont les suivantes :

1. Établissement du CA.
   - Production de la CRL initiale.
   - Publication "out-of"band" (i.e. autrement qu'électroniquement) de la clef publique ou d'un hash de cette clef vérifiable sur le dépot.
   - Production d'un certificat signé avec cette propre clef publique (ASN.1 : la signature est vérifiable avec le champ "SubjectPublicKeyInfo", le "Subject" et l'"Issuer" sont identiques, leurs clefs sont également identiques).
   - Migration d'un ancien CA vers un nouveau (cf. subsection 4.8.4).
2. Initialisation des entités finales.
   - Importer une clef publique d'un CA.
   - Demander les informations sur les options supportées par la PKI (les

algorithmes autorisés, les chaines de certification entre le CA racine et le CA certifiant, etc.).

3. Certification : création de nouveaux certificats.
   - Enregistrement initial : création des clefs par le CA, le RA ou l'entité finale (les clefs ne doivent pas être prévisibles, tous les nombres au hasard doivent utiliser des générateurs sûrs). Deux schémas d'enregistrement sont obligatoirement possibles dans une PKI : un schéma centralisé le plus simple possible (initiation faite par le CA, pas d'authentification de l'origine des messages, génération des clefs par le CA, pas de confirmation) et un schéma authentifié présenté figure 4.22 (initiation par l'entité finale, authentification *requise* de l'origine des messages, génération des clefs par l'entité finale, message de confirmation *requis* sans quoi le certification doit être révoqué).

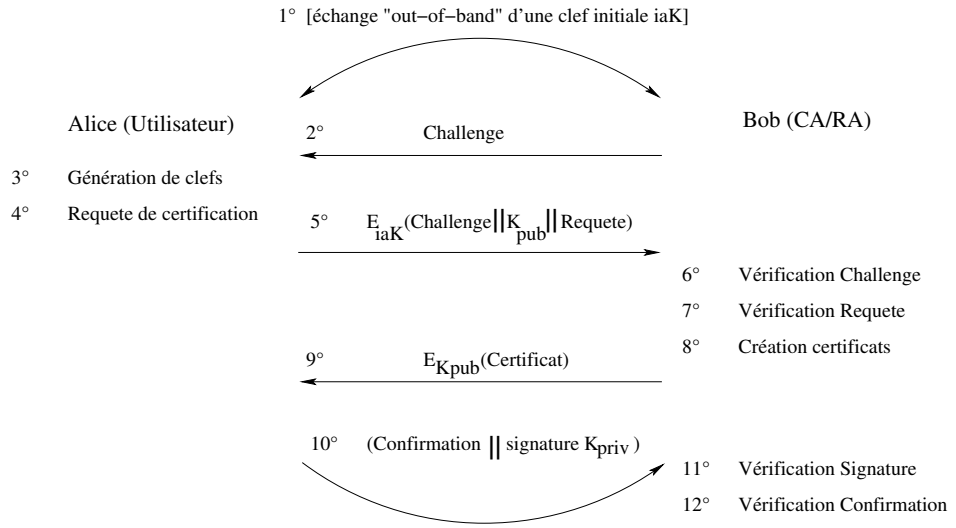

FIG. 4.22 – Enregistrement authentifié

   - Mise à jour des clefs : chaque certificat doit être changé régulièrement. La clef suivante est échangée à l'aide de la précédente.
   - Mise à jour des certificats : à la date d'expiration, le certificat peut être prolongé et une nouvelle date d'expiration remplace l'ancienne.
   - Mise à jour des clefs d'un CA (cf. subsection 4.8.4).
   - Certifications croisées : un certificat croisé est un certificat contenant la clef publique d'un CA et signé par un autre CA. Il peut être inter-domaines (si "subject" et "issuer" sont dans des domaines administratifs différents) ou intra-domaine ; mutuel ou non.
4. Publication : des certificats et des CRL.
5. Recouvrement : les clefs d'une entité finale *peuvent* être sauvegardées préventivement par un système de sauvegarde associé à un CA/RA ;

en cas de nécessité de recouvrement, un nouvel enregistrement peut s'avérer nécessaire.

6. Révocation : une personne/serveur/etc. autorisée informe un CA d'une situtation qui requiert la création de nouvelles entrées de CRL ou de nouvelles CRL.
7. Environnement de Sécurité personnel (PSE) : les opérations locales des utilisateurs (déplacement, changement de mot de passe, etc.) ne sont pas du ressort de la PKI mais peuvent utiliser le même type de protocoles (authentification ,enregistrement, etc.).

Authentification d'entités à partir de certificats Nous venons de voir de façon générale les différents éléments d'une PKI. Nous disposons donc d'un mécanisme pour nous fournir des certificats qu'il s'agit maintenant d'utiliser. L'authentification d'entités à partir de certificats a fait l'objet en particulier d'une norme présentée dans [14] et est présentée ci-dessous.

Le tableau 4.7 résume les notations utilisées dans ce protocole tandis que la figure 4.23 en donne les différentes étapes.

A	Nom qui identifie Alice
B	Nom qui identifie Bob
$Sign_x(M)$	La signature du message M avec la clé privée de X
R_x	Un challenge aléatoire produit par X
$Cert_X$	Le certificat de X
$X \cdot Y$	La concaténation de X et de Y
$TokenID$	Identificateur de Token. Il précise les informations identifiant le token, le type de protocole... qui faciliteront le traitement du Token.
$TokenXY$	Un token envoyé de X vers Y
$TokenXY_i$	Le i-ème token envoyé de X vers Y
$[Z]$	Précise que le champs Z est optionnel

TAB. 4.7 – Notations utilisées dans FIPS 196

Quelques remarques préliminaires :

- Tout d'abord, le contenu des tokens de la forme $TokenXY$ et $TokenXY_i$ sera explicité dans le suite.
- Ensuite, seule une version simplifiée du protocole est présentée ici. Celui ci prévoit par exemple la présence de champs optionnels contenant des informations que chaque partie souhaite communiquer à l'autre au cours de l'authentification. Pour plus de détails, voir [14].

Mais passons maintenant à la description du protocole (il s'agit du protocole d'authentification mutuelle ; l'authentification unilatérale ne contient pas le dernier message) :

1. Alice envoit à Bob une demande d'authentification

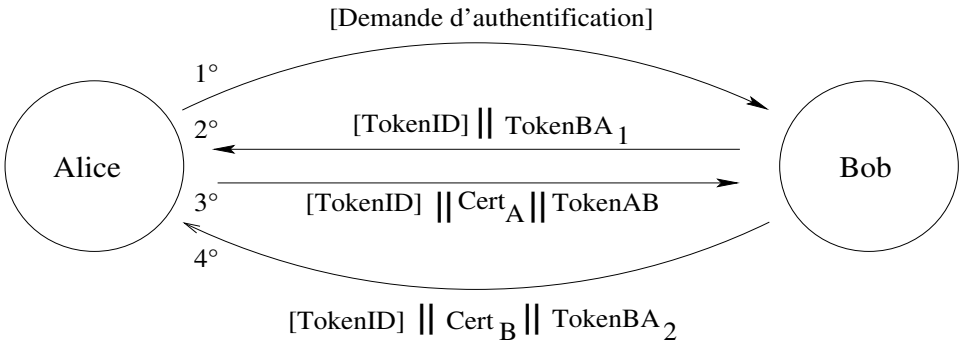

FIG. 4.23 – Standard d'Authentification d'Entité utilisant la Cryptographie à Clé Publique

2. Bob génère un nombre aléatoire R_b qu'il retient et créer ainsi le token de challenge $TokenBA_1$ par :

$$TokenBA_1 = R_b$$

Bob l'envoie à Alice avec éventuellement un token d'identification.

3. Alice génère un nombre aléatoire R_a qu'elle retient. ensuite, elle génère un token d'authentification $TokenAB$ de la forme :

$$TokenAB = R_a \cdot R_b \cdot B \cdot Sign_a(R_a \cdot R_b \cdot B)$$

Alice l'envoie à Bob avec son certificat (qui contient donc sa clé publique qui permettra de vérifier la signature) et éventuellement un identificateur de token (en fait, même l'envoi du certificat est optionnel)

4. Bob commence par vérifier le certificat d'Alice. Ensuite, il peut vérifier les informations contenues dans $TokenAB$ à l'aide de la clé publique d'Alice qui est contenue dans le certificat. Après vérification de la signature d'Alice, celle ci est authentifiée auprès de Bob. Il lui reste à s'authentifier en générant lui aussi un token d'authentification $TokenBA_1$ par :

$$TokenBA_2 = R_b \cdot R_a \cdot A \cdot Sign_b(R_b \cdot R_a \cdot A)$$

Bob l'envoie à Alice avec son certificat et éventuellement avec un identificateur de token (même remarque que précedemment)

5. Alice procède de la même manière pour authentifier Bob. A la fin, il y a donc bien authentification mutuelle.

Une fois les deux parties authentifiées, elles peuvent convenir d'une clé secrète en utilisant les protocoles d'échange de clé classiques (exemple : Diffie-Hellman 4.6.1) etc. ou [28] chap 22).

Processus de migration d'un ancien CA vers un nouveau Différents problèmes interviennet lorsqu'il devient nécessaire de migrer un CA. Tout d'abord, il faut que les anciennes clefs restent valides au moins un certain temps, afin que la migration soit relativement transparente pour les utilisateurs. De ce fait, plusieurs certificats et clefs du même objet peuvent co-exister. La migration est donc résolu par les actions suivantes de l'opérateur du CA :

1. Générer une nouvelle paire de clefs.
2. Créer un nouveau certificat "OLDwNEW" contenant l'ancienne clef publique et signé par la nouvelle clef privée. La validité de ce certificat doit se terminer avec l'expiration des anciennes clefs.
3. Créer un nouveau certificat "NEWwOLD" contenant la nouvelle clef publique et signé par l'ancienne clef privée. La validité de ce certificat se termine quand *tous* les utilisateurs de ce CA ont récupéré la nouvelle clef publique et au plus tard à l'expiration de l'ancienne clef publique.
4. Créer un nouveau certificat auto-signé contenant la nouvelle clef publique et signé par la nouvelle clef privée.
5. Publier ces trois nouveaux certificats dans le dépot.
6. Exporter la nouvelle clef publique "out-of-band".

Á la suite de ces manipulations des dépots peuvent contenir les anciennes et les nouvelles clefs ou seulement les anciennes si ils n'ont pas encore été mis à jour. De la même manière, un utilisateur peut posséder les anciennes ou les nouvelles clefs du CA. Les conflits sont alors résolus de la manière résumée dans le tableau 4.8.

4.8.5 Politique de sécurité et contre-mesures

La réponse à la question :"Qu'est-ce qu'un système sûr ?" dépend fortement du contexte. Il est donc nécessaire de concevoir des systèmes adaptés aux menaces réelles. Ainsi, une *politique de sécurité* est un modèle de confiance fondé sur l'analyse des menaces.
Dans ce cadre, la politique de sécurité est la stratégie quand les contre-mesures sont la tactique.

Politique de sécurité Une bonne politique doit définir les buts, la stratégie globale et répondre aux menaces. En particulier, il est fondamental de décrire :

- Qui est responsable de quoi (mise en œuvre, exécution, audit, tests, etc.).
- Quelle est la politique de sécurité de base du réseau d'ordinateurs.
- Pourquoi chacun doit faire ce qu'il fait.

	Certificat signé avec NEW_{priv}	Certificat signé avec OLD_{priv}
Dépot contient OLD et NEW PSE contient NEW_{pub}	Vérification DIRECTE	Récupérer "OLDwNEW" Vérifier "OLDwNEW" avec NEW_{pub} Vérifier le certificat avec OLD_{pub}
PSE contient OLD_{pub}	Récupérer "NEWwOLD" Vérifier "NEWwOLD" avec OLD_{pub} Vérifier le certificat avec NEW_{pub}	Vérification DIRECTE
Dépot ne contient que OLD PSE contient NEW_{pub}	Vérification DIRECTE (sans le dépot)	ÉCHEC à cause du CA
PSE contient OLD_{pub}	ÉCHEC à cause du CA	Vérification DIRECTE

TAB. 4.8 – Résolution des conflits liés à la migration d'un CA

La politique de sécurité consite donc à déterminer quelles contre-mesures employer sachant qu'en pratique, un système est sûr :

- *si le temps passer à le décrypter dépasse le temps de validité de l'information qu'il renferme.*
- *si les moyens à mettre en œuvre pour le décrypter dépassent la valeur de l'information qu'il contient.*

Modélisation de la menace et contre-mesures Pour modéliser efficacemnt la menace contre un système, il faut penser à toutes les menaces possibles jusqu'à n'en plus trouver. B. Schneier a donc proposé une démarche méthodique pour la description des menaces et des contre-mesures qui en découlent : les arbres d'attaque [27]. L'idée est très simple, une structure en arbre est nécessaire, le nœud racine étant le but à atteindre, par exemple "lire un message chiffré". Les fils de chaque nœud étant alors les moyens d'y arriver. Il faut ensuite donner une valeur à chaque case et remonter les coût en selectionnant à chaque fois le minimum. Le coût obtenu pour le nœud racine étant la valeur du but.

Ensuite, il faut un arbre pour chaque but différent. Chaque arbre doit circuler entre différentes personnes afin d'ajouter de plus en plus d'attaques.

Ce procédé doit être répété régulièrement afin de trouver les points faibles du système.

4.8.6 Défauts des PKI

Les PKI ne sont pas la panacée, quelques défauts peuvent être isolés :

1. Plusieurs personnes peuvent avoir un même "Distinguished Name". Ce risque est réduit par l'accumulation d'informations (pays, localité, région, adresse, courriel, IP, DNS, etc.). Cependant, une question demeure, qui certifie ces informations complémentaires ?
2. Que faut-il exactement comprendre lorsqu'un CA prétend être digne de confiance ? Ce problème est réduit par les certifications croisées, mais il faut que *tous* les enregistrements soient bien initiés "out-of-band".
3. La politique de sécurité de l'ensemble des CA est donc celle du plus faible !
4. Enfin, il manque une autorité de Datation.

En conclusion, il est nécessaire de bien définir la politique de sécurité et de constamment faire évoluer l'arbre d'attaques d'une PKI. Enfin, au final, les défauts des PKI sont quasiment tous résolus si une PKI est restreinte à une seule entité administrative (e.g. une seule entreprise) et restent raisonnables lorsque plusieurs de ces entités interagissent.

4.9 Un utilitaire de sécurisation de canal : SSH

4.9.1 Description

SSH est un protocole sécurisé de connexion à distance et autres services réseaux utilisant les service d'un réseau non-sur dans un modèle client-serveur.

Il se compose de trois parties :
- une couche transport, qui permet l'authentification du serveur, la confidentialité et l'intégrité,
- une couche d'authentification de l'utilisateur,
- un protocole de connexion capable de multiplexer le tunnel crypté pour permettre le passage de plusieurs canaux logiques.

De plus, ce protocole garantit :
- la confidentialité des données par un chiffrement fort ;
- l'intégrité des communications ;
- l'authentification des entités ;
- le contrôle de privilèges ;
- la mise en place d'un canal sécurisé pour la suite des communications ;

Comme PGP, il utilise un protocole à clé publique pour échanger des clés secrète utilisés par les algorithmes de chiffrements plus rapides utilisés pour la connexion elle-même.

Il existe deux versions de ce protocole. Seul le protocle SSH-1 est décrit ici (les différences avec le protocole SSH-2 seront vues au §4.9.8), à partir de [18] et de [33].

4.9.2 Authentification du serveur

Chaque serveur possède sa propre clé publique qui l'identifie. Le problème est de diffuser cette clé aux clients. Il y a grosso-modo deux stratégies :

- le client maintient une base de donnée de clé publiques des serveurs auquel il se connecte,
- Il existe un tier de confiance capable de diffuser de facon sûr, les clés.

En pratique, pour l'instant, l'algorithme consiste à ce que le serveur envoie sa clé publique, et lors de la première connexion, le client la stocke dans sa base (actuellement dans le fichier `~/.ssh/known_hosts`). Le client compare cette clé avec les clés recus ultérieurements. Néanmoins, cela autorise quand même une attaque *man-in-the-middle*, même si elle devient plus complexe. Pour ajouter un peu de sécurité, une signature de la clé est calculée et peut être facilement échanger par d'autres moyens (téléphone).

4.9.3 Etablissement d'une connexion sécurisée

Elle se déroule en plusieurs phases et permet au serveur de fournir de façon sécurisée une clé de session K qui sera utilisée par un algorithme de chiffrement à clé secrète choisi :

1. *le client contacte le serveur*
   Il s'agit simplement pour le client d'émettre une demande de connexion sur le port TCP du serveur (le port 22).
2. *le client et le serveur s'échangent les versions du protocole SSH qu'ils supportent*
   Ces protocoles sont représentés par une chaine ASCII
   (ex : `SSH-1.5-1.2.27`).
3. *le serveur et le client passent à une communication par paquets formatés.*
   Le format des paquets est donné dans la figure 4.24.
   Voici la signification des différents champs de ce paquet :
   - **Length** : il s'agit de la taille du paquet (sans compter les champs "Length" et "Padding") codée sous forme d'un entier de 32 bits.
   - **Padding** : il s'agit de données aléatoires dont la longueur varie de 1 à 8 octets ; plus précisément, la longueur est donnée par la formule

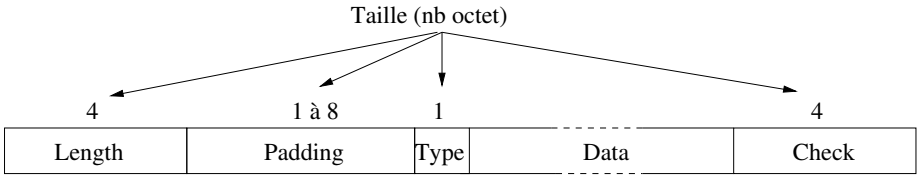

FIG. 4.24 – Format d'un paquet SSH

8 - (length mod 8). Ce champ permet de rendre les attaques à texte clair choisi difficilement exploitables.
- **Type** : spécifie le type de paquet.
- **Data** : les données transmises.
- **Check** : il s'agit d'un CRC (Cyclic Redundancy Check) sur 32 bits (avec le polynôme classique 0xedb88320) sur les champs "Padding", "Type" et "Data".

Ce sont les champs "Padding", "Type", "Data" et "Check" qui seront chiffrés dès qu'une clé de session aura été établie.

4. *Le serveur s'identifie au client et fournit les paramètres de session*
Le serveur envoie les informations suivantes au client (toujours en "clair") :
   - une "host key" (publique) H (qui est LA clef publique du serveur) ;
   - la "server key" (publique) S (une clef RSA regénérée toutes les heures) ;
   - une séquence C de 8 octets aléatoires (octets de contrôle) : le client devra inclure ces octets dans la prochaine réponse sous peine de rejet (mesure de protection contre les attaques de type "IP-spoofing") ;
   - la liste des méthodes de chiffrement, de compression et d'authentification supportées.

5. *Le client envoie au serveur une clé secrète de session*
Le client choisit une méthode M de chiffrement supportée par les deux parties (parmi IDEA en mode CFB, DES en mode CBC, 3DES en mode CBC et RC4), puis génère aléatoirement une clé de session K et envoie $[M, C, E_H(E_S(K))]$. Ainsi, seul le serveur sera capable de déchiffrer K, car seul le serveur connait la clef privée associée à H. Le deuxième cryptage par S assure que le message n'a pas été rejoué puisque S n'est valide que pendant une heure.

   Le client et le serveur calculent également chacun de leur coté un identificateur de session (Id_S) qui identifie de manière unique la session et qui sera utilisé plus tard.

6. *Chaque entité met en place le chiffrement et complète l'authentification du serveur*
Après avoir envoyé sa clé de session, le client devra attendre un message

de confirmation du serveur (chiffré à l'aide de la clé de session K), qui achèvera d'authentifier le serveur auprès du client (seul le serveur attendu est capable de déchiffrer la clé de session).

7. *La connexion sécurisée est établie* : le client et le serveur disposent d'une clé secrète de session qui leur permet de chiffrer et de déchiffrer leurs messages.

4.9.4 Authentification du client

Une fois qu'une connexion sécurisée a été mise en place, le client peut s'authentifier au serveur. Cette authentification peut se décliner sous différentes formes :

- *Mot de passe.*
- *Clé publique* : l'initialisation de cette méthode d'authentification se fait en deux phases :

  1. Le client génère à l'aide de la commande `ssh-keygen` les fichiers `~/.ssh/identity` (qui contiendra la clé secrète RSA d et qui est protégée par une pass-phrase) et `~/.ssh/identity.pub` (qui contiendra la clé publique RSA (e, n)).
     Ce dernier fichier est au format suivant :
     `Taille_cle Exposant Modulo Nom_client`
  2. le contenu de `identity.pub` est ajouté sur le serveur au fichier `~/.ssh/authorized_keys`

  Une fois cette étape effectuée, l'authentification du client se déroule de la manière suivante :

  1. le client envoie une requête pour une authentification par clé publique. le modulo (n) est passé en paramètre et sert d'identificateur.
  2. le serveur peut rejeter la requête s'il ne permet pas l'authentification de cette clé (pas d'entrée associée dans le fichier `authorized_keys` etc...).
     Sinon, il génère aléatoirement un challenge C de 256 bits et envoie au client $C^e \bmod n$ (chiffrement RSA avec la clé publique du client).
  3. A l'aide de sa clé privée D, le client déchiffre le challenge C, le concatène avec l'identificateur de session Id_S, hashe le tout avec MD5 et envoie le résultat du hashage au serveur comme réponse au challenge.
  4. Le serveur effectue la même opération de son coté et vérifie la concordance du résultat avec ce qu'il reçoit. Cette dernière vérification achève l'authentification du client.

– *Utilisation du nom de l'hote* : cette solution (utilisant des fichiers `.rhost`) n'est évidemment pas satisfaisante en terme de sécurité mais reste possible.

4.9.5 Sécurité des algorithmes

Les algorithmes de cryptage, intégrité et compression sont négociés dans les deux sens. Les algorithmes utilisés sont bien connu et bien établis. Néanmoins cela autorise le changement d'algorithme si une faille apparaissait.

4.9.6 Intégrité des données

Comme on l'a vu, le contrôle d'intégrité se fait par un simple CRC32 ce qui n'est pas efficace contre les corruptions effectives (cf l'attaque par insertion de Futoransky et Kargieman [16]).

Cette faiblesse est en partie à l'origine du changement de version. Dans SSH-2, l'intégrité de chaque paquet est vérifié en ajoutant des bits de signature (MAC : message authentication code ; voir §4.7.3). Ils sont calculés à partir du numéro de séquence du paquet (qui ne fait pas parti du paquet) et de son contenu non encrypté. Les algorithmes peuvent être SHA1 ou MD5.

4.9.7 Compression des données

Les données peuvent être compressées par exemple à l'aide de l'utilitaire gzip ce qui permet de limiter la bande passante utilisée pour la transmission des paquets.

4.9.8 Différences majeures entre SSH-1 et SSH-2

SSH-2 fournit :

– un plus grand choix dans la négociation d'algorithmes entre le client et le serveur (choix du hashage etc...) ;
– la gestion des certificats X.509 pour les clés publiques ;
– une plus grande flexibilité dans l'authentification (incluant une authentification partielle) ;
– un meilleur test d'intégrité par chiffrement comme on l'a vu
– le remplacement périodique de la clé de session tous les 1GHz ou toutes les heures par défaut.
– les clés secrètes sont désormais de 128 bits, les clés publiques de 1024 bits..
– plusieurs méthodes pour l'échange de clés, en particulier Diffie-Hellman (voir §4.6.1) :
  Le client choisit d'abord x, calcule et envoie $e \equiv g^x[p]$ pour p premier et g d'ordre q. Le serveur choisit y, calcule un $f \equiv g^y[p]$ et l'envoie

avec sa clef publique (K_S), le tout hashé et signé par sa clef privée :

$$(K_S||f||s)$$

C vérifie alors que la clé publique de l'hôte est la même que dans sa base de clefs et vérifie la signature. Le client et le serveur peuvent alors calculer la clef de session Diffie-Hellman $K \equiv e^y \equiv f^x$.
Pour SSH2, les opérations se font modulo $p = 2^{1024} - 2^{960} - 1 + 2^{64} * [floor(2^{894}pi) + 129093]$, le générateur est 2, d'ordre $\frac{p-1}{2}$. p a été prouvé premier et sa valeur décimale est :
`179769313486231590770839156793787453197860296048756011706444423684197180216158519368947833795864925541502180565485980503646440548199239100050792877003355816639229553136239076508735759914822574862575007425302077447712589550957937778424442426617334727629299387668709205606050270810842907692932019128194467627007`

4.9.9 Multiplexage de canaux

SSH est capable de faire passer dans une seule connexion cryptée, plusieurs connexions (eg. TCP). Il permet par exemple de faire passer un flux X11, ou de se connecter à un serveur qui n'utilise pas un protocole crypté (eg. telnet).

Chapitre 5

Détection et correction d'erreurs.

Dans le chapitre 2, on supposait toutes les communications « sans bruit », c'est-à-dire qu'un message émis était systématiquement reçu tel quel. Ce n'est pas le cas dans la réalité, et le taux d'erreur que peuvent introduire certains canaux justifie l'enrichissement des modèles de la théorie des codes de nouveaux outils.

Commençons par introduire les techniques de base dans le cas d'un code binaire (la théorie sera énoncée dans le cas général). Lorsqu'une source émet une séquence de bits $S = s_1, \ldots, s_k$, la séquence $S' = s'_1, \ldots, s'_k$ reçue par le destinataire peut différer : si $s'_i \neq s_i$, on dit qu'il y a eu erreur sur le bit d'indice i.

Le taux d'erreur (nombre d'erreurs par rapport au nombre de bits émis) dépend de la nature de la ligne de transmission (locale/internationale, nombre de répéteurs, support câble/satellite, ...) ; il varie habituellement de 10^{-4} à 10^{-7}.

Les méthodes pour se protéger contre ces erreurs en les détectant voire en les corrigeant de manière automatique, consistent à ajouter des bits d'information supplémentaires, appelés *bits de contrôle*, ou *bits de redondance*. Le code transforme donc la séquence à transmettre $S = s_1, \ldots, s_k$ en une séquence $\phi(S) = s'_1, \ldots, s'_k, \ldots, s'_{k+r}$ qui comporte r bits de redondance par rapport au message initial.

Pour permettre le décodage, la fonction ϕ doit bien sûr être injective ; ainsi, le calcul après réception de $\phi^{-1}(s'_1, \ldots, s'_{k+r}) = S$ permet de reconstruire le message source.

Comme ϕ est injective, $E_\phi = \mathrm{Im}(\phi)$ est un sous-ensemble $E_\phi \subset \{0,1\}^{k+r}$ qui contient seulement 2^k mots de code (chacun codé sur $k+r$ bits). Ainsi, la réception d'un mot qui n'est pas dans E_ϕ (i.e. un mot de $\overline{E_\phi} = \{0,1\}^{k+r} \setminus E_\phi$) indique une erreur. La détection d'erreur repose alors sur un test de non-appartenance à E_ϕ.

Dans un souci d'efficacité, le code ϕ doit être choisi pour que ce test :

- permette de détecter le plus d'erreurs possibles ;
- soit le moins coûteux possible.

Suite à une détection d'erreurs, le décodeur peut éventuellement procéder à une correction. On distingue deux classes de corrections :

- correction directe : le signal erroné reçu contient suffisamment d'information pour permettre de retrouver le mot émis ;
- correction par retransmission (ou ARQ) : une demande de retransmission du message source est effectuée lorsque l'erreur ne peut pas être corrigée.

Le but de ce chapitre est d'introduire les principaux codes détecteurs/correcteurs, les codes *cycliques* (on dit aussi polynômiaux), et leurs fondements mathématiques. Les théories mathématiques sur lesquelles s'appuient la manipulation des codes correcteurs (arithmétique des corps finis, polynômes et classes cyclotomiques), sont présentés au chapitre 1 pour mettre en évidence dans le texte les résultats directement utilisés pour construire des codes. On pourra les consulter au préalable, ou en renvoi lors de leur utilisation.

Un exemple simple de détection : bit de parité longitudinale. Si on code le mot $m = (s_1, \ldots, s_k)$ par $\phi(m) = (s_1, \ldots, s_k, s_{k+1})$ où $s_{k+1} = (\sum_{i=1}^{k} s_i)\ mod\ 2$, il est évident que cette dernière égalité devient fausse lorsqu'un nombre impair de bits dans $\phi(m)$ changent de valeurs. Ainsi, l'ajout d'un bit de parité longitudinale permet de détecter une erreur portant sur un nombre impair de bits (figure 5.1).

Mot de code sur 7 bits	Mot avec bit de parité
0101001	01010011
0001001	00010010
0000000	00000000

FIG. 5.1 – Détection d'erreur par bit de parité.

Un exemple simple de correction directe : contrôle de parité longitudinale et transversale. On calcule cette fois un bit de parité pour chaque ligne, et chaque colonne (le mot est fragmenté en plusieurs lignes). Ce code permet de corriger une erreur (on peut localiser le bit où s'est produite l'erreur) et de détecter trois erreurs (figure 5.2).

Mot de code sur 21 bits	Mot avec bits de parité
0101001	0101001 **1**
0001001	0001001 **0**
0000000	0000000 **0**
	01000000

FIG. 5.2 – Correction d'erreur par bits de parité.

5.1 Formalisation du problème et définitions

5.1.1 Code systématique par blocs

Soit $m \in V^+$ un message à transmettre (on suppose donc le message source préalablement encodé sous forme d'une séquence de chiffres d'un code sur V). La plupart des codes correcteurs découpent d'abord m en blocs de taille identique (on parle de *codes par blocs*), et rajoutent à la fin de chaque bloc les chiffres de redondance associés (c'est le principe du *code systématique*).

Définition 15. *Un* codage systématique par blocs de k chiffres avec r bits de redondance *consiste à :*

1. *partitionner le message* $m = m_0 \dots m_{lk-1} \in V^{lk}$ *à transmettre en blocs de* k *chiffres consécutifs, i.e. :*

$$m = M_0 ... M_{l-1} \text{ avec } M_i = m_{ik} \dots m_{(i+1)k-1}$$

2. *Coder chaque bloc* M_i *en ajoutant* r *chiffres de redondance* $R_i \in V^r$*, i.e. :*

$$R_i = \phi(M_i)$$

où $\phi : V^k \to V^r$ *est une fonction.*

Le message m est alors codé par la séquence : $[M_0 R_0 M_1 R_1 \dots M_{l-1} R_{l-1}]$. On définit les quantités suivantes :

- $n = k + r$ est appelé *longueur* du code ;
- $R = \frac{k}{n}$ est appelé *rendement* du code ;
- le code est dit *code* (n, k).

5.1.2 Code correcteur et distance de Hamming

Un code (n,k) est dit *t-détecteur* (resp. *t-correcteur*) si il permet de détecter (resp. corriger) toute erreur portant sur t chiffres ou moins sur un bloc de n chiffres.

Exemples :

- L'ajout d'un bit pour contrôler la parité des 7 bits qui le précèdent est un code systématique par bloc. C'est un code (8,7). Il est 1-détecteur et 0-correcteur avec un rendement 87,5%.
- Le contrôle de parité longitudinale et transversale sur 21 bits (avec ajout de 11 bits de contrôle) est un code (32,21). Il est 1-correcteur avec un rendement 65,625%.

Le nombre d'erreurs lors de la transmission d'un mot de code est le nombre de chiffres différents entre le mot émis et le mot reçu.

Dans toute la suite on considère un code de longueur n ; les mots de code sont donc des éléments de V^n, l'espace vectoriel de dimension n sur V. Soient m_1 et m_2 deux éléments de V^n : $m_1 \oplus m_2$ (resp. $m1 \otimes m2$) désigne l'élément de V^n obtenu par addition (resp. multiplication) composante par composante de m_1 et m_2.

Définition 16. *Soit $x = (x_1, \ldots, x_n) \in V^n$.*
On appelle poids de Hamming *de x, notée $w(x)$, le nombre de composantes non nulles de x i.e.*

$$w(x) = Card\{i \in \{1, \ldots, n\} / x_i \neq 0\}\,.$$

Remarque 2. *Cette fonction satisfait l'inégalité triangulaire : $w(x \oplus y) \leq w(x) + w(y)$.*

Définition 17. *Soient $x = (x_1, \ldots, x_n)$ et $y = (y_1, \ldots, y_n)$ deux mots de V^n.*
On appelle distance de Hamming *entre x et y, notée $d(x, y)$, le nombre de composantes pour lesquelles x et y diffèrent, i.e.*

$$d(x, y) = w(x - y)\,.$$

L'application d ainsi définie est une distance sur V^n ; en effet pour tous x, y et z dans V^n on a :

- $d(x, y) \in \mathbb{R}^+$;
- $d(x, y) = 0 \Leftrightarrow x = y$;
- $d(x, y) = d(y, x)$;
- $d(x, y) \leq d(x, z) + d(y, z)$.

Cette distance de Hamming permet de caractériser le nombre d'erreurs que peut corriger un code C de longueur n. En effet, soit $m \in C$ un mot de n chiffres émis et soit m' le mot reçu, supposé différent de m (i.e. $d(m, m') > 0$) :

- pour que C puisse détecter une erreur, il est nécessaire que $m' \notin C$ (sinon, le mot reçu est un mot de C donc considéré comme correct).
- pour pouvoir faire une correction (i.e. retrouver m à partir de m'), il faut que m soit l'unique mot de C le plus proche de m' ; i.e.

$$\forall x \in C : x \neq m \Rightarrow d(x, m') > d(m, m')\,.$$

C'est ce que traduit la propriété :

Propriété 8. *Soit C un code de longueur n. C est t-correcteur si*

$$\forall x \in V^n,\ \ Card\{c \in C / d(x,c) \leq t\} \leq 1\,.$$

Remarque 3. *La correction d'une erreur peut aussi être décrite en considérant les boules $B_t(c)$ de centre $c \in C$ et de rayon t :*

$$B_t(c) = \{x \in V^n / d(c,x) \leq t\}\,.$$

Lors de la réception de m', on peut corriger m' en m si on a :

$$B_{d(m,m')}(m') \cap C = \{m\}\,.$$

La capacité de correction d'un code C est donc liée à la distance minimale entre deux eléments de C.

Définition 18. *La* distance minimale *du code C, notée $\delta(C)$, est définie par :*

$$\delta(C) = \min_{(c_1,c_2)\in C^2; c_1 \neq c_2} d(c_1,c_2)\ . \tag{5.1}$$

La propriété 8 et la remarque 3 sont à la base du théorème suivant qui caractérise un code t-correcteur.

Théorème 20. *Soit C un code de longueur n. Les propriétés suivantes sont équivalentes, et toutes impliquent par conséquent que C est t-correcteur :*

(i) $\forall x \in V^n :\ Card\{c \in C / d(x,c) \leq t\} \leq 1$;
(ii) $\forall c_1, c_2 \in C : c_1 \neq c_2 \Longrightarrow B_t(c_1) \cap B_t(c_2) = \emptyset$;
(iii) $\forall c_1, c_2 \in C : c_1 \neq c_2 \Longrightarrow d(c_1,c_2) > 2t$;
(iv) $\delta(C) \geq 2t+1$.

Preuve.

- $(i) \Longrightarrow (ii)$: Supposons qu'il existe $x \in B_t(c_1) \cap B_t(c_2)$; ainsi c_1 et c_2 sont à une distance $\leq t$ de x. D'après (i), cela n'est possible que pour au plus un mot de code ; donc $c_1 = c_2$.
- $(ii) \Longrightarrow (iii)$: par l'absurde. Si $d(c_1,c_2) \leq 2t$; soient $i_1, \ldots, i_d$ avec $d \leq 2t$ les indices où les chiffres de c_1 et c_2 diffèrent. Soit x le mot de V^n dont les chiffres sont les mêmes que ceux de c_1 sauf les chiffres en position $i_1, ..., i_{d/2}$ qui sont égaux à ceux de c_2. Alors $d(x,c_1) = d/2 \leq t$ et donc $x \in B_t(c_1)$. De même, $d(x,c_2) = d - (d/2) \leq t$ et donc $x \in B_t(c_2)$. D'où $x \in B_t(c_1) \cap B_t(c_2)$ ce qui contredit (ii).
- $(iii) \Longrightarrow (iv)$: immédiat.
- $(iv) \Longrightarrow (i)$: par la contraposée. Supposons qu'il existe $x \in V^n$, tel que Card$\{c \in C / d(x,c) \leq t\} \geq 2$. Il existe alors $c_1 \neq c_2$ tels que $d(x,c_1) \leq d(x,c_2) \leq t$. Et comme d, distance de Hamming, vérfie l'inégalité triangulaire : $d(c_1,c_2) \leq d(c_1,x) + d(x,c_2) \leq 2t$. Donc $\delta(C) \leq 2t$.

□

Exercice 14. *Montrer que si C est t-correcteur alors C est k-détecteur avec $k \geq 2t$. La réciproque est-elle vraie ?*

Exercice 15. *Soit C un code (n,k) sur un vocabulaire V. Écrire un programme qui calcule le taux de détection et de correction du code ; donner le coût de ce programme en fonction de k et n. Application : Soit le code binaire $C = \{0000000000, 0000011111, 1111100000, 1111111111\}$. Que valent n et k pour ce code ? Calculer son rendement, son taux de détection et de correction.*

Exercice 16. *Le contrôle de parité transversale et longitudinale sur 4 bits conduit au code (9,4) suivant :*
Le mot de code associé à la source $b_0b_1b_2b_3$ est :

$$[b_0, b_1, b_2, b_3, (b_0 + b_1), (b_2 + b_3), (b_0 + b_2), (b_1 + b_3), (b_0 + b_1 + b_2 + b_3)] .$$

On obtient ainsi C= { 000000000, 000101011, 001001101, 001100110, ... }.

1. *Montrer que C est exactement 1-correcteur.*
2. *Donner une configuration comportant 2 erreurs non corrigibles.*
3. *Montrer que C est 3-détecteur mais pas 4-détecteur.*

5.1.3 Code parfait

Un code t-correcteur permet de corriger toute erreur e de poids $w(e) \leq t$; mais il peut subsister des erreurs détectées non corrigées (voir exercice précédent). On dit qu'un code est parfait lorsque toute erreur détectée est corrigée. Ce paragraphe étudie quelques propriétés des codes parfaits.

Définition 19. *Un code t-parfait est un code t-correcteur dans lequel toute erreur détectée est corrigée.*

Si C est t-parfait, lorsqu'on reçoit un mot $m' \notin C$, alors il existe un unique mot m de C tel que $d(m, m') \leq t$. De plus, comme C est t-correcteur, les boules de rayon t et de centre des mots de C sont deux à deux disjointes ; d'où le théorème suivant.

Théorème 21. *Le code C (n,k) sur V est t-parfait si les boules ayant pour centre les mots de C et pour rayon t forment une partition de V^n, i.e.*

$$\biguplus_{c \in C} B_t(c) = V^n .$$

Évidemment, une telle partition n'est possible que pour certaines valeurs de n et k, en fonction du cardinal de V. Le thérorème suivant donne des conditions nécessaires pour l'existence de codes t-correcteurs et t-parfaits.

Théorème 22. *Soit $C(n,k)$ un code t-correcteur sur V. Alors*

$$1 + C_n^1(|V|-1) + C_n^2(|V|-1)^2 + \cdots + C_n^t(|V|-1)^t \leq |V|^{n-k}. \qquad (5.2)$$

Si de plus il y a égalité, alors $C(n,k)$ est t-parfait.

Preuve. La preuve repose sur le calcul du cardinal de $\biguplus_{c\in C} B_t(c)$, les boules étant 2 à 2 disjointes dans le cas d'un code t-correcteur. En effet, le cardinal d'une boule de rayon t de V^n est :

$$|B_t(x)| = 1 + C_n^1(|V|-1) + C_n^2(|V|-1)^2 + \cdots + C_n^t(|V|-1)^t\,.$$

Comme un code $C(n,k)$ possède exactement $|V|^k$ éléments de V^n associées à des boules deux à deux disjointes dont l'union est incluse dans V^n, on a :

$$|V|^k\Big(1 + C_n^1(|V|-1) + C_n^2(|V|-1)^2 + \cdots + C_n^t(|V|-1)^t\Big) \leq |V|^n,$$

d'où la première inégalité. S'il y a égalité, alors le cardinal de l'union est égal à $|V|^n$; de plus, comme le code est t-correcteur, les boules sont disjointes. Donc leur union est égale à V^n et le code est parfait. □

Exercice 17. *Montrer que tout code 1-correcteur sur des mots de $k = 4$ bits ($V = \{0,1\}$) requiert au moins 3 bits de redondance. Montrer que si il existe un code 1-correcteur avec 3 bits de redondance, alors il est parfait.*

5.1.4 Cas particulier où $V = \{0,1\}$ - Codes de Hamming

Dans le cas d'un vocabulaire binaire, on suppose que $V = \mathbb{F}_2 = \mathbb{Z}/2\mathbb{Z}$.
Soit $x \in F_2^n$; on a $w(x) = \text{Card}\{i \in \{1,\ldots,n\}/x_i \neq 0\} = \sum_{i=1}^n x_i$.
Dans F_2, $a + b = a - b$; on a alors les propriétés suivantes :

Propriété 9. *Soient x et y deux éléments quelconques de F_2^n ;*
- $w(x) = d(x,0)$;
- $d(x,y) = w(x+y)$;
- $w(x+y) = w(x) + w(y) - 2w(x.y)$.

Exercice 18. *Soit un code binaire $(k+r,k)$ 1-correcteur qui consiste à ajouter r bits de redondance pour k bits de données.*

a. Montrer que

$$k \leq 2^r - r - 1. \qquad (5.3)$$

b. En déduire une borne sur le rendement maximal d'un code 1-correcteur dont le nombre de bits de contrôle est 3, puis 4, puis 5, puis 6.

c. Existe-t-il un code 1-parfait de longueur $n = 2^m$ où $m \in \mathbb{N}$?

Codes de Hamming

Connaissant k, l'inégalité 5.3 (qui sécrit $2^r \geq n+1$) permet de déterminer le nombre minimal de bits de contrôle à ajouter pour obtenir un code 1-correcteur. Les codes de Hamming (1950) permettent alors d'atteindre cette limite théorique : pour n bits, le nombre de bits de contrôle est $\lfloor \log_2 n \rfloor + 1$ (soit, si $n+1$ est une puissance de 2, $\log_2(n+1)$). Ce sont donc des codes parfaits pour $n = 2^r - 1$) ; autrement dit, ce sont des codes $(n, n - \lfloor \log_2 n \rfloor - 1)$.
Description. Un mot $c = c_1 \dots c_n \in \{0,1\}^n$ de code de Hamming est tel que : les bits c_i dont l'indice i est une puissance de 2 (i.e 2^l avec $l = 0, 1, \dots$), sont des bits de contrôle et les autres sont des bits de données. Le bit de contrôle d'indice $i = 2^l$ est le ou-exclusif (contrôle de parité) de tous les bits de données c_j dont l'indice j écrit en base 2 a le $(l+1)^{\text{ième}}$ bit (à partir de la gauche) à 1.

Par exemple, dans un code $(7,4)$ de Hamming, au message source $[s_0, s_1, s_2, s_3]$ correspond le mot de code $[c_1, c_2, c_3, c_4, c_5, c_6, c_7]$ donné par :

$$\begin{cases} c_3 = s_1 \\ c_5 = s_2 \\ c_6 = s_3 \\ c_7 = s_4 \\ c_1 = c_3 \oplus c_5 \oplus c_7 \\ c_2 = c_3 \oplus c_6 \oplus c_7 \\ c_4 = c_5 \oplus c_6 \oplus c_7 \end{cases}$$

Ainsi le message 1000 est codé 1110000 (dans le code $(7,4)$ de Hamming).

Pour assurer la correction dans un code de Hamming, le contrôle de parité est fait de la façon suivante. Tous les bits de contrôle d'indice $i = 2^l$ sont vérifiés ; une erreur est détectée si l'un de ces bits est erroné (parité erronée). Soit alors e la somme des indices des bits de contrôle i qui sont erronés. Si il y a une seule erreur, elle provient alors du bit e.

Théorème 23. *Le code de Hamming $(n, n - \lfloor log_2 n \rfloor - 1)$ est un code 1-correcteur qui requiert un nombre de bits de contrôle minimal parmi tous les codes (n,k) qui sont 1-correcteur.*
En particulier, le code de Hamming $(2^m - 1, 2^m - 1 - m)$ est un code 1-parfait.

La preuve découle directement des propriétés ci-dessus.

En outre, nous avons immédiatement la propriété suivante :

Propriété 10. *La distance d'un code de Hamming est 3.*

Preuve. Le code est 1-parfait donc $\delta(C) \geq 2t + 1 = 3$. Il suffit donc d'exhiber deux mots distants de 3 : pour n'importe quel mot, en changeant uniquement le premier bit de donnée (c_3), seuls les deux premiers bits de contrôle doivent être modifiés. □

Correction d'erreur dans le code (7,4). En reprenant l'exemple précédent, supposons que l'on recoive le mot 1111101 : le message source est alors 1101, mais le contrôle de parité indique que les bits 2 et 4 sont erronés ; la correction d'une erreur unique est alors réalisée en modifiant le bit d'indice 4+2=6. Le mot corrigé est alors 1111 (qui est codé en 1111111).

5.1.5 Codes cycliques : CRC

Pour qu'un code correcteur soit intéressant en pratique, il faut qu'il soit efficace à calculer d'une part (i.e. implémentable sur un circuit simple) et que d'autre part il permette facilement de calculer le mot de code le plus proche du mot reçu lors de la détection d'erreurs.

De plus, il est important de pouvoir construire des codes ayant un rendement maximal : pour un nombre de chiffres de redondance donné, le taux de correction doit être maximal.

En pratique, les codes les plus utilisés qui satisfont à ces critères sont les codes de redondance cyclique, appelés *CRC*. Ces codes sont des codes linéaires (i.e. les bits de contrôle sont des combinaisons linéaires des bits d'information) qui sont de plus stables par décalage des chiffres. Un tel code est caractérisé par un *polynôme générateur*; on parle parfois de *code polynomial*.

5.2 Construction de codes cycliques

5.2.1 Codes linéaires.

Nous avons vu qu'un code correcteur (n, k) est caractérisé par une application $\phi : V^k \to V^n$. L'analyse de ϕ dans un contexte quelconque est difficile ; aussi, on se restreint au cas où ϕ est linéaire. Cette restriction offre des avantages en termes de temps de calcul : le codage et décodage seront effectués en temps linéaire. Pour pouvoir étudier ce cas, il faut que V^k et V^n soient des espaces vectoriels, donc que V soit un corps (avec les restrictions déjà mentionnées, voir les rappels mathématiques au chapitre 1).

Tous les résultats d'algèbre linéaire sont valides. Par exemple, si ϕ est une application linéaire de V^n, alors : $\dim(\mathrm{Im}(\phi)) + \dim(\mathrm{Ker}(\phi)) = n$.

Définition 20. *Soit C un code (n, k) sur V. On dit que C est un code linéaire de longueur n et de dimension k lorsque C est un sous-espace vectoriel de V^n de dimension k.*

Puisque deux espaces vectoriels de même dimension sur un même corps sont isomorphes, un code C (n, k) sur V est un code linéaire si et seulement si il existe une application linéaire $\phi : V^k \longrightarrow V^n$ telle que $\phi(V^k) = C$.

Pour une base de C fixée, la donnée de la matrice G comportant k lignes et n colonnes et à coefficients dans V et dont les k lignes forment la base de

C détermine l'application linéaire ϕ. Une telle matrice G est appelée *matrice génératrice* du code C.

Remarque 4. -) Parce que les mots de code sont souvent représentés sous forme de vecteurs lignes : on a choisi de représenter ϕ par une matrice rectangulaire dans laquelle les vecteurs générateurs de $Im(\phi)$ sont les lignes. **-)** Pour un même code linéaire C, il y'a autant de matrices génératrices qu'il y'a de choix possibles d'une base de C. Mieux, si G est une matrice génératrice de C, alors pour toute matrice carrée inversible A d'ordre k et à éléments dans V, la matrice $G' = A \cdot G$ est une matrice génératrice de C.

Preuve.

$$\mathrm{Im}(G') = \{x^t.A \cdot G / x \in V^k\} = \{y^t.G / y \in \mathrm{Im}((A)^t)\} = \{y^t.G / y \in V^k\} = C\ .$$

□

Exemple : Considérons le code $(4,3)$ de parité sur $V = \{0,1\}$. Le mot de code associé au mot d'information $X = [x_0, x_1, x_2]$ est alors $B = [b_0, b_1, b_2, b_3]$ défini par :

$$\begin{cases} b_0 = x_0 \\ b_1 = x_1 \\ b_2 = x_2 \\ b_3 = x_0 + x_1 + x_2 \bmod 2 \end{cases}$$

Ce code est un code linéaire sur $V = Z/2Z$ de matrice génératrice :

$$G = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}.$$

On a alors : $B = X.G$.

Théorème 24. *Soit C un code (n,k) linéaire et soit $d = \min_{x \in C; x \neq 0} w(x)$. Alors*

$$\delta(C) = d,$$

i.e. C est $(d-1)$-détecteur et $\lfloor \frac{d-1}{2} \rfloor$-correcteur. Le code C est alors dit code (n,k,d).

Preuve. Comme ϕ est linéaire, $C = Im(\phi)$ est un sous-espace vectoriel de V^n. Donc $0 \in C$ (0 est le vecteur dont les n composantes sont nulles). D'où : $\forall c \in C, \delta(C) \leq d(c,0) = w(c)$, i.e. $\delta(C) \leq \min_{c \in C} w(c)$.
Réciproquement, soient c_1 et c_2 deux éléments de C tels que $\delta(C) = d(c_1, c_2) = w(c_1 - c_2)$. Comme C est un sous-espace vectoriel, $c = c_1 - c_2$ appartient à C ; $\delta(C) \geq \min_{c \in C} w(c)$.
Finalement $\delta(C) = \min_{c \in C} w(c)$. □

Théorème 25. Borne de Singleton. *La distance minimale d d'un code (n, k) linéaire sur V est majorée par :*

$$d \leq n - k + 1 \,.$$

Preuve. Soit C un code linéaire défini par l'application linéaire ϕ. Comme ϕ est injective, rang$(\phi) = k$. D'où dim(C) = dim(Im(ϕ))=k.
Considérons le sous-espace E de V^n formé par les vecteurs dont les $k-1$ dernières composantes sont nulles : dim$(E) = n - k + 1$.
Ainsi, E et C sont deux sous-espaces vectoriels de V^n et dim(C) + dim(E) $= n+1 > n$. Il existe donc un élément non nul a dans $C \cap E$. Comme $a \in E$, $w(a) \leq n - k + 1$; comme $a \in C$, $\delta(C) \leq w(a) \leq n - k + 1$. cqfd. □

Exemple : Le code de Hamming $(7, 4)$ est un code linéaire qui admet comme matrice génératrice :

$$G = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} . \tag{5.4}$$

Sa distance est 3 ; c'est donc un code linéaire $(7, 4, 3)$.

Plus généralement, le code de Hamming pour n bits est un code $(n, n - 1 - \lfloor log_2 n \rfloor, 3)$ (nous avons vu qu'un tel code est 1-correcteur).

5.2.2 Codage et décodage des codes linéaires.

Le décodage et la correction d'erreur, si le code n'a pas de structure particulière, peut être très long, voire impossible pour des très grands code. Il consisterait, pour chaque mot reçu, à le comparer avec tous les mots de C et à l'identifier au plus proche. En pratique, et grâce à la structure linéaire des codes, le codage et le décodage s'effectuent en temps raisonnable avec des opérations simples.

Définition 21. *On dit que deux codes linéaires C et C', avec les mêmes paramètres n et k sont équivalents lorsqu'il existe une permutation σ des coordonnées telle que*

$$C' = \{\sigma([c_1, \ldots, c_n]) \ : \ [c_1, \ldots, c_n] \in C\} \,.$$

Pour une telle permutation σ, on a

$$G_\sigma = G \cdot P_\sigma \tag{5.5}$$

où G et G_σ sont les matrices génératrices de C et C_σ respectivement et P_σ est la matrice de permutation correspondant à σ.

Pour l'étude du codage et du décodage du code linéaire C, nous pouvons nous limiter au cas où la matrice génératrice G, s'exprime sous la forme :

$$G = \left[\begin{array}{c|c} & \\ L & R \\ & \end{array}\right] \tag{5.6}$$

où L est une matrice carrée $k \times k$ inversible, et R une matrice $k \times r$.
En effet, si G n'est pas de la forme (5.6), on permute les colonnes de G jusqu'à obtenir L inversible ; ceci est possible puisque $\mathrm{rg}(G) = k$. On obtient alors une matrice génératice G_σ d'un code linéaire C_σ équivalent à C. Après le codage et la correction à partir de C_σ, on retrouve le message initial en reappliquant la matrice de permutation : en effet, $P_\sigma \cdot P_\sigma = I_d$.

Nous étudirons donc seulement le cas où G, s'exprime sous la forme (5.6).

On construit alors $G' = L^{-1}G$, soit :

$$G' = \left[\begin{array}{c|c} & \\ I_k & T = L^{-1}R \\ & \end{array}\right]$$

et G' est également génératrice de C, voir la remarque 4.

La matrice G' est appelée matrice génératrice *normalisée (ou canonique)* du code C. Tout code linéaire possède une matrice génératrice normalisée. Un code linéaire est donc systématique.

Cette matrice normalisée est de plus unique. En effet, Soit $[I_k|T_1]$ et $[I_k|T_2]$ deux matrices génératrices de C. Pour tout $x \in V^k$, $[x^t, x^tT_1]$ et $[x^t, x^tT_2]$ sont donc dans C. Comme C est un sous-espace vectoriel, leur différence $[0, x^t(T_1 - T_2)]$ est aussi dans C. Or, le seul elément de C dont les k premières composantes sont nulles est 0. D'où $\forall x \in V^k$: $x^tT_1 = x^tT_2$, i.e. $T_1 = T_2$.

Codage On déduit de ce qui précède une méthode de codage de C : tout mot source $u \in V^k$ (vecteur ligne) est codé par $\phi(u) = uG' = [u, u.T]$. Ainsi, on écrit u suivi de uT qui contient les chiffres de redondance.

Décodage La méthode de détection d'erreur et de décodage utilise la propriété suivante :

Propriété 11. *Soit H la matrice $r \times n$ définie par :* $H = \left[\begin{array}{c|c} & \\ T^t & -I_r \\ & \end{array}\right]$.

Alors $x = [x_1, \ldots x_n] \in C$ si et seulement si $H.\begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = O$ *(i.e. le vecteur nul de V^r).*

Preuve. Soit $x \in C$; alors $\exists u \in V^k : x = u[I_k|T]$.
D'où $xH^t = u.[I_k|T].\begin{bmatrix} T \\ -I_r \end{bmatrix}$.
Or, $[I_k|T].\begin{bmatrix} T \\ -I_r \end{bmatrix} = [T - T] = [0]$; i.e. pour tout x, $Hx = 0$.

Réciproquement : si $Hx = 0$ alors $[x_1, \ldots, x_n].\begin{bmatrix} T \\ -I_r \end{bmatrix} = [0, \ldots, 0]_r$.

La composante j $(1 \leq j \leq r)$ donne alors : $[x_1, \ldots, x_k].\begin{bmatrix} T_{1,j} \\ \vdots \\ T_{k,j} \end{bmatrix} - x_{k+j} = 0$

Soit $x_{k+j} = [x_1, \ldots, x_k].\begin{bmatrix} T_{1,j} \\ \vdots \\ T_{k,j} \end{bmatrix}$. D'où $[x_{k+1}, \ldots, x_{k+r}] = [x_1, \ldots, x_k].T$.
Finalement on a : $[x_1, \ldots, x_n] = [x_1, \ldots, x_k].[I_k|T] = [x_1, \ldots, x_k].G'$; ainsi $x \in C$. □

On appelle H la *matrice de contrôle* de C. Pour détecter une erreur si on reçoit un mot y, il suffit donc de calculer Hy, qu'on appelle le *syndrôme d'erreur*. On détecte une erreur si et seulement si $Hy \neq 0$. On cherche alors à calculer le mot émis x à partir de y. Pour cela, on calcule le vecteur d'erreur $e = y - x$. Ce vecteur e est l'unique elément de V^n de poids de Hamming $w_H(e)$ minimal tel que $He = Hy$. En effet, si C est t-correcteur, $\exists! x \in C\big(d_H(x, y) \leq t\big)$. Comme $d_H(x, y) = w_H(x - y) = w_H(e)$, on en déduit : $\exists! e \in V^n\big(w_H(e) \leq t\big)$. De plus $He = Hy - Hx = Hy$ car $Hx = 0$.

On construit alors une table de $|V|^r$ entrées, dans laquelle chaque entrée i correspond à un elément unique z_i de $\mathrm{Im}(H)$; dans l'entrée i, on stocke le vecteur $e_i \in V^n$ de poids minimal et tel que $He_i = z_i$.
La correction est alors triviale : Quand on reçoit y, on calcule $H.y$ et l'entrée i de la table tel que $z_i = Hy$. On trouve alors e_i dans la table et on renvoie $x = y + z_i$.

Exemple : Considérons le code de Hamming (7,4) dont la matrice génératrice G est donnée par la relation (5.4) (voir à la page 153). Dans ce cas, on a :

$$L = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \quad ; \quad R = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{et} \quad T = L^{-1}R = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{pmatrix} .$$

La matrice génératrice canonique G' et la matrice de contrôle H sont données

par :

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix} \quad \text{et} \quad \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} .$$

Si l'on reçoit le mot $y = 1111101$, alors le calcul $Hy = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix}$ montre qu'il y a erreur. Pour corriger l'erreur, il suffit (la table à $2^3 = 8$ entrées n'est pas necessaire dans cet exemple simple) de constater que le vecteur $e = 0000010$ est de poids (de Hamming) minimal et tel que $He = Hy$. Donc la correction de y est $x = y + e = 1111111$.

En s'intéressant aux codes linéaires, on a donc nettement amélioré les performances du codage et décodage avec correction d'erreur. Il existe une classe de codes linéaires, appelée classe des codes cycliques, qui améliore encore la facilité du calcul, et la possibilité de l'implanter simplement sur des circuits électroniques. De plus, grâce aux codes cycliques, on verra aussi des méthodes simples pour construire des codes en garantissant un taux de correction.

5.2.3 Codes cycliques

Un code cyclique est un code linéaire qui est stable pour l'opération de décalage de chiffre.

Définition 22. *On appelle* opération de décalage *l'application linéaire σ de V^n définie par*

$$\sigma([u_0, \ldots, u_{n-1}]) = [u_{n-1}, u_0, \ldots, u_{n-2}] .$$

L'opération σ est linéaire, et on peut exhiber sa matrice Σ :

$$\Sigma = \begin{bmatrix} 0 & 0 & \cdots & \cdots & 1 \\ 1 & 0 & \cdots & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & 1 & 0 \end{bmatrix}$$

Définition 23. *On dit qu'un code linéaire C de V^n est* cyclique *lorsque*

$$\sigma(C) = C .$$

Exemple : Le code de parité $(n, n-1)$ est un code cyclique. En effet, si $c = (c_0, \ldots, c_{n-1})$ est un mot de code, alors $c_{n-1} = \sum_{i=0}^{n-2} c_i \bmod 2$. Mais on a alors aussi $c_{n-2} = c_{n-1} + \sum_{i=0}^{n-3} c_i \bmod 2$; ainsi, $\sigma(c) = (c_{n-1}, c_0, \ldots, c_{n-2})$ est aussi un mot de code. Le code de parité est donc cyclique.

5.2.4 Construction d'un code cyclique : polynôme générateur.

Tout élément $U = [u_0, \ldots, u_{n-1}]$ de V^n peut être représenté par le polynôme de degré n de $V[X]$ (voir la section 1.3.8 du chapitre 1 page 47)

$$P_U = \sum_{i=0}^{n-1} u_i X^i.$$

Comme $X^n - 1$ est un polynôme de degré n, $V^n \equiv V[X]/(X^n - 1)$. Dans $V[X]/(X^n - 1)$, on a alors

$$P_{\sigma(U)} = X.P_U(X) - u_{n-1}.(X^n - 1) = X.P_U \bmod (X^n - 1)$$

Autrement dit, l'opération de décalage σ d'un vecteur revient à multiplier son polynôme associé par X dans $V[X]/(X^n - 1)$.

Cette propriété est à la base du théorème suivant qui donne une caractérisation algébrique d'un code cyclique.

Théorème 26. *Tout code cyclique $C = (n, k)$ admet une matrice génératrice G_C de la forme :*

$$G_C = \begin{bmatrix} m \\ \sigma(m) \\ \ldots \\ \sigma^{k-1}(m) \end{bmatrix}$$

avec $m = [a_0, a_1, \ldots, a_{n-k} = 1, 0, \ldots, 0]$ tel que

$$g(X) = \sum_{i=0}^{n-k} a_i X^i$$

est un diviseur unitaire de $(X^n - 1)$ de degré $r = n - k$.
Le polynôme g est appelé polynôme générateur *du code cyclique.*
Réciproquement, tout diviseur g de $(X^n - 1)$ est polynôme générateur d'un code cyclique.

Cette propriété permet la construction directe de code correcteur par la donnée d'un polynôme diviseur de $X^n - 1$. Un tel polynôme peut être calculé à partir des classes cyclotomiques relatives à q modulo n (voir la section 1.3.8 de la page 47).

Exemple : Construction d'un code cyclique (7,4).
Considérons la décomposition de $X^7 - 1$ dans $\mathbb{F}_2[X]$. Les classes cyclotomiques et les polynômes irréductibles associés sont alors :

- pour $i = 0$: $\Sigma_0 = \{0\}$. D'où $g_{\Sigma_0} = X - \alpha^0 = X - 1$.
- pour $i = 1$: $\Sigma_1 = \{1, 2, 4\}$ (car $8 = 1 \bmod 7$). D'où $g_{\Sigma_1} = (X - \alpha^1)(X - \alpha^2)(X - \alpha^4)$.

- pour $i = 3$: $\Sigma_3 = \{3, 6, 5\}$ (car $2.5 = 10 = 3 \bmod 7$). D'où $g_{\Sigma_3} = (X - \alpha^3)(X - \alpha^5)(X - \alpha^6)$.

En fait, on montre que $\mathbb{F}_2[X]$ n'admet que deux ploynômes irréductibles de degré 3 : $(1 + X + X^3)$ et $(1 + X^2 + X^3)$. Ces deux polynômes sont donc g_{Σ_1} et g_{Σ_3}. Le polynôme $g = (1 + X^2 + X^3)$ est donc le polynôme générateur d'un code cyclique. Comme g est de degré 3 et que $n = 7$, ce code est un code (7,4). Sa matrice est :

$$G_C = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix}$$

Plus généralement, la factorisation de $X^7 - 1$ permet d'exhiber 6 diviseurs non triviaux de $X^7 - 1$ qui définissent chacun un code cyclique :

- $g_1 = X - 1$: code cyclique (7,6)
- $g_2 = X^3 + X + 1$: code cyclique (7,4)
- $g_3 = X^3 + X^2 + 1$: code cyclique (7,4)
- $g_4 = (X - 1)(X^3 + X + 1)$: code cyclique (7,3)
- $g_5 = (X - 1)(X^3 + X^2 + 1)$: code cyclique (7,3)
- $g_6 = (X^3 + X + 1)(X^3 + X^2 + 1)$: code cyclique (7,1)

5.2.5 Codage et décodage d'un code cyclique.

Le lien entre code cyclique et polynôme générateur est important ; il est à la base des algorithmes efficaces de codage et de décodage qui évitent la multiplication par la matrice génératrice G_C (de coût $O(nk)$) en la remplaçant par une multiplication par le polynôme g, de coût $O(n \log n)$. L'intérêt pratique est que les circuits associés au codage et décodage sont relativement simples à réaliser.

Soit C un code cyclique de polynôme générateur g ; soit G une matrice génératrice de C construite à partir de g comme dans le théorème 26.

Opération de codage.

Soit $a = [a_0, \ldots, a_{k-1}] \in V^k$ un mot source et $P_a = \sum_{i=0}^{k-1} a_i X^i$ son polynôme associé. Le mot de code associé à a est $\phi(a) = aG$ de polynôme associé P_{aG}. De part l'écriture de G à partir des coefficients de $g(X)$ (théorème 26), on a :

$$\begin{aligned} P_{aG} &= \sum_{i=0}^{k-1} a_i (X^i g(X) \bmod X^n - 1) \\ &= [g(X)(\sum_{i=0}^{k-1} a_i X^i)] \bmod X^n - 1 \\ &= [g(X).P_a(X)] \bmod X^n - 1 \ . \end{aligned}$$

Le codage correspond donc à un produit de polynômes, les degrés des monômes étant pris modulo n : en effet, calculer $X^n - 1$ revient à considérer que $X^n = 1 = X^0$.
Exemple : Soit le code cyclique (7,4) précédent associé au polynôme $g = (1 + X^2 + X^3)$. Soit $a = [a_0, a_1, a_2, a_3]$ un mot source ; le code de ce mot est alors égal à $s.G$; il est obtenu par multiplication à droite par la matrice G. Le même mot est obtenu en calculant les coefficients du polynôme :

$$(a_0 + a_1X + a_2X^2 + a_3X^3)(1 + X^2 + X^3) \bmod X^7 - 1\,.$$

Ainsi, pour $a = [1001]$: $P_a.g \bmod X^7 - 1 = (1 + X^3)(1 + X^2 + X^3) = 1 + X^2 + 2.X^3 + X^5 + X^6 = 1 + X^2 + X^5 + X^6$. Donc $\phi(a) = [1010011]$.

Détection d'erreur et opération de décodage.

Le codage précédent montre que tout mot de code reçu est un multiple de $g(X)$. Soit $m \in V^n$ un message reçu et soit P_m son polynôme associé. Lors du décodage, on calcule d'abord $P_e = P_m \bmod g$;

- si $P_e = 0$: le polynôme reçu est bien un multiple de g : il correspond donc à un mot de code. On retrouve le message émis en calculant le quotient P_m/g.
- sinon, $P_e \neq 0$: le mot reçu n'est pas un mot de code et il y a donc eu des erreurs lors de la transmission.

La détection d'erreur est donc réalisée via le calcul de $P_e = P_m$ modulo g ; P_e est le *syndrôme d'erreur*.

Dans le cas où P_e est non nul, on peut procéder à une correction. Une manière brutale est de tabuler tous les mots de V^n en associant à chacun le mot de code le plus proche, qui correspond au message corrigé. La correction se fait alors par lecture de la table.

Le lien entre code cyclique et polynômes fait qu'il existe des algorithmes moins coûteux en place mémoire pour corriger un message erroné. La méthode de Meggitt et ses variantes en sont des exemples.

5.2.6 Classes cyclotomiques et distance minimale.

Le taux de correction d'un code cyclique est difficile à calculer. Cependant, ce paragraphe présente un théorème qui permet de garantir une minoration de la distance minimale d'un code, et par suite (cf théorème 24) une minoration du taux de détection. Ce théorème est de plus constructif : il permet la construction de codes cycliques ayant un taux de correction garanti.

On suppose que n est premier avec q. Soit alors α une racine primitive de $X^n - 1$ dans $\mathbb{F}_q[X]$ et soit C un code cyclique de polynôme générateur g. Comme g est un diviseur de $X^n - 1$, on a $g = \prod_{i \in \Sigma}(X - \alpha^i)$ où $\Sigma \subset \{0, 1, \ldots, n-1\}$ est la réunion de classes cyclotomiques relatives à q

modulo n. Le théorème suivant montre que l'analyse de Σ permet d'avoir une minoration de la distance minimale $\delta(C)$ du code C.

Théorème 27. *On suppose que n est premier avec q.*
Si il existe un entier a tel que $\{a+1, a+2, \ldots, a+s\} \subset \Sigma$ alors

$$\delta(C) \geq s+1.$$

Le code C est donc au moins s-détecteur et $[s/2]$-correcteur.

Exemple : Considérons $n = 7$ et $V = \mathbb{F}_2$. Un code cyclique est alors associé à un diviseur de X^7-1. Nous avons vu que les classes cyclotomiques relatives à 2 modulo 7 sont :

- $\Sigma_0 = \{0\}$; $g_0 = X - 1$.
- $\Sigma_1 = \{1, 2, 4\}$; $g_1 = (1 + X + X^3)$.
- $\Sigma_3 = \{3, 6, 5\}$; $g_2 = (1 + X^2 + X^3)$.

On a : $\{1, 2\} \subset \Sigma_1$. Ici, $s = 2$: le code (7,4) associé à g_1 est donc au moins 1-correcteur.
De la même façon, on a $\{5, 6\} \subset \Sigma_3$; le code (7,4) associé à g_2 est donc aussi au moins 1-correcteur (c'est un code de Hamming).
Considérons maintenant le code C associé au polynôme $g = g_1.g_2$; comme g est de degré 6, C est un code (7,1) qui comporte 6 bits de redondance. Il est caractérisé par la classe $\Sigma = \Sigma_1 \cap \Sigma_3 = \{1, 2, 3, 4, 5, 6\}$; ici, $s = 6$. Ce code est donc au moins 3-correcteur.

5.2.7 Codes BCH.

En application du théorème 27 (voir page 160), Bose, Chaudhuri et Hocquenghem ont proposé une méthode de construction de codes cycliques ayant un taux de correction arbitraire. Ces codes sont appelés codes BCH. Soit a un entier arbitraire. Pour avoir un taux de correction de t au moins, il suffit en effet d'après le théorème 27 que le polynôme g ait parmi ses racines $\{\alpha^{a+1}, \alpha^{a+2}, \ldots, \alpha^{a+2t}\}$. Autrement dit, il suffit que g soit associé à la partie $\Sigma \subset \{1, 2, \ldots, n-1\}$ plus petite union possible de classes cyclotomiques contenant $\{a+1, a+2, \ldots, a+2t\}$.

Un tel code est appelé *code BCH*. Sa distance minimale est supérieure à $2t$ et donc son taux de correction est garanti supérieur à t. De part leurs propriétés algébriques, il existe des algorithmes spécifiques de codage et décodage des code BCH, plus performants que ceux pour les codes cycliques quelconques.

Dans la plupart des cas pratiques, on choisit n premier avec q ; $X^n - 1$ se factorise alors à partir des classes cyclotomiques, toutes les racines étant de multiplicité 1. En particulier, les codes BCH les plus utilisés correspondent à $n = q^t - 1$; un tel code est dit *primitif*.

Les paragraphes suivants présentent différents codes de type BCH qui sont utilisés dans la pratique.

5.3 Codes cycliques usuels

Nous présentons dans cette section rapidement quelques codes couramment utilisés dans des contextes dans lesquels la communication numérique est cruciale.

Pour les applications où le taux d'erreurs est faible, un codage de Hamming peut être utilisé. Le Minitel par exemple utilise un code de Hamming (128,120) qui est 1-correcteur : le message est tronçonné en blocs de 15 octets, i.e. 120 bits ; Un 16ème octet contient 8 bits de contrôle qui permet de localiser un bit d'erreur parmi les 128. En outre un 17ème octet, dit de validation et ne contenant que des 0, est utilisé pour détecter des perturbations importantes. Le code final est donc un code binaire 1-correcteur de paramètres (136,120).

Cependant, les développements des codes cycliques et notamment la mise au point de procédure de codage et de décodage particulièrement efficaces ont motivé l'intégration de ces codes correcteurs dans de nombreuses applications où le taux d'erreurs est important et donc la capacité de correction critique. Les paragraphes suivants montrent quelques uns de ces codes utilisés pour la lecture de disques compacts et pour la transmission d'images par satellites.

5.3.1 Codes de Reed-Solomon

Les codes de Reed-Solomon sont des codes BCH primitifs sur $V = \mathbb{F}_{2^m}$ (i.e. $q = 2^m$) et de longueur $n = 2^m - 1$ (i.e. $n = q - 1$; n est donc premier avec q).

L'intérêt de ces codes est double. D'une part, ils sont optimaux dans le sens où ils requièrent un nombre de chiffres de redondance minimal pour une capacité de correction fixée. D'autre part, ils sont particulièrement faciles à coder et décoder.

En effet, le polynôme $X^{2^m-1} - 1$ se factorise très simplement sur $\mathbb{F}_{2^m}$: ses racines sont tous les élements non nuls de $\mathbb{F}_{2^m}$:

$$X^{2^m-1} - 1 = \prod_{\lambda \in \mathbb{F}_{2^m} - \{0\}} (X - \lambda) .$$

Le groupe multiplicatif $\mathbb{F}_{2^m} - \{0\}$ étant cyclique, soit α un élément primitif. On a alors :

$$X^{2^m-1} - 1 = \prod_{i=1}^{2^m-1} (X - \alpha^i) .$$

La construction d'un code de Reed-Solomon avec r chiffres de redondance repose sur le choix d'un polynôme générateur de degré r, dont les racines sont consécutives :

$$g(X) = \prod_{i=s}^{s+r-1} (X - \alpha^i) .$$

Le code de Reed-Solomon ainsi obtenu est donc un code $(n = 2^m - 1, k = n - r)$ avec r arbitraire.

Le taux de correction de ce code est optimal. En effet, en tant que code BCH, la distance minimale δ du code de Reed-Solomon est au moins $\delta \geq r+1$. Or, la borne de Singleton (théorème 25) montre que $\delta \leq n - k + 1 = r + 1$; ainsi, la distance est $\delta = r + 1$ et atteint la borne de Singleton.

Les codes de Reed-Solomon sont très utilisés en pratique [24]. Ainsi, le satellite d'exploration de Jupiter *Galileo* utilise le code de Reed-Solomon (255,223) de distance 33. Ce code est 16-correcteur sur le corps $\mathbb{F}_{2^8}$. Si α est une racine d'un polynôme utilisé pour engendrer $\mathbb{F}_{2^8}$, le polynôme générateur du code de Reed-Solomon (255,223,33) est :

$$g = \prod_{j=12}^{43} (X - \alpha^{11j})$$

qui est de degré $r = 32$.

5.3.2 Codes C.I.R.C.

Les codes de Reed-Solomon sont à la base des codes C.I.R.C. (*Cross Interleaved Reed-Solomon Code*, i.e. code de Reed-Solomon à entrelacement croisé). Les codes C.I.R.C. sont des codes "raccourcis" des codes de Reed-Solomon : ils ne contiennent que les mots du code de Reed-Solomon commençant par un nombre fixé de 0, en supprimant ces 0 de tête. Ils ont donc la même distance que le code de Reed-Solomon dont ils sont issus.

Par exemple, le code C.I.R.C (32,28,5) est un code raccourci du code de Reed-Solomon(255,251,5) est utilisé pour les diques compacts. Ce code permet de détecter 4 erreurs et d'en corriger 2, ce qui s'avère assez efficace pour les disques compacts qui sont soumis à de multiples perturbations (rayures, poussières, défauts de surface etc).

5.3.3 Quelques autres codes cycliques

De nombreux autres codes cycliques ou leurs variantes sont utilisés en pratique.

Les codes de Golay sont des codes cycliques dont la longueur n est un nombre premier, de plus premier avec $q = \text{card}(V)$. Dans ce cas, $(X^n - 1)$ possède une factorisation simple en deux facteurs non triviaux de degré chacun $(n-1)/2$. Ainsi, le code de Golay avec $n = 11$ et $q = 3$ est un code ternaire associé à un polynôme génerateur de degré $r = 5$. Ce code a donc pour paramètre $(11, 6)$; il est 2-correcteur et parfait.

Les codes de Reed-Muller sont des codes (n, k) cycliques *étendus* : les mots de code sont ceux d'un code cyclique $(n-1, k)$ avec un chiffre supplémentaire qui est la somme des $n - 1$ chiffres du mot du code cyclique. Ces

codes de Reed-Muller sont notamment utilisés pour la transimssion d'images par satellite.

Chapitre 6

Solutions des exercices

Exercice 1, page 74.

1. “0123321045677654” et “0123456701234567”.
2. “0123012345670123” et “0123456777777777”.
3. Les fréquences sont égales à 1/8, l'entropie est donc maximale et 3 bits sont nécessaires pour chaque caractère. Il faut donc $3*16/8 = 6$ octets.
4. Pour la première chaine on obtient l'entropie : $H = 4(3/16*\log_2(16/3))+ 4(1/16 * \log_2(16)) = 1.81127 + 1 = 2.81127$. Par Huffman le code est donc : 00,01,100,101,1100,1101,1110,1111, d'où un codage sur seulement $2*3*2+2*3*3+4*1*4 = 46 = 5.75$ octets. La deuxième chaîne s'y prête encore mieux : $H = 7(1/16 * \log_2(16)) + (9/16 * \log_2(16/9)) = 1.75 + .4669171867 = 2.216917187$. Huffman donne alors : 00,01,100,101,1100,1101,111,1, d'où seulement $2+2+3+3+4+4+3+1*9 = 30 = 3.75$ octets !
5. Huffman donne : 000,001,010,011,1 pour respectivement 12, 34, 56, 67 et 77. Ainsi, $4*3+4*1 = 16 = 2$ octets sont nécessaires, mais avec une table de taille 256 octets au lieu de 8 triplets soit 3 octets.
6. un “Move-to-Front” suivi d'un code statistique est donc un “code statistique localement adaptatif”.
7. Pour $k = 1$ on obtient “0123220345676644” et “0123456770123457” d'entropies respectives $H = 2.858$ et $H = 2.953$. Pour $k = 2$, cela donne : “0123112345675552” et “0123456777012347” d'où $H = 2.781$ et $H = 2.875$.

Exercice 2, page 80.
La compression de "BLEBLBLA" par LZW donne donc le résultat suivant :

chaîne		Affichage		Dictionnaire		
B	$\rightsquigarrow$	42		BL	$\leftrightarrow$	80
L	$\rightsquigarrow$	4C		LE	$\leftrightarrow$	81
E	$\rightsquigarrow$	45		EB	$\leftrightarrow$	82
BL	$\rightsquigarrow$	80		BLB	$\leftrightarrow$	83
BLB	$\rightsquigarrow$	83		BLBL	$\leftrightarrow$	84
A	$\rightsquigarrow$	41				

Si l'on décompresse cette sortie sans traiter le cas particulier, on obitent à cause du décalage de mise en place du dictionnaire la sortie suivante :

Code		Affichage		Dictionnaire		
42	$\rightsquigarrow$	B				
4C	$\rightsquigarrow$	L		BL	$\leftrightarrow$	80
45	$\rightsquigarrow$	E		LE	$\leftrightarrow$	81
80	$\rightsquigarrow$	BL		EB	$\leftrightarrow$	82
83	$\rightsquigarrow$	? ? ?				

On voit donc bien ici, que le traitement du cas particulier permet de mettre à jour le dictionnaire un itération plus tôt. Ainsi, toute chaîne répétée deux fois de suite peut être traitée correctmeent à la décompression.

Exercice 3, page 93.
LA CURIOSITE EST UN VILAIN DEFAUT

Exercice 4, page 94.

1. En supposant qu'un seul message en clair et son équivalent crypté aient été interceptés, la clef est facilement obtenue par $K = \tilde{M} \oplus M$.
2. En outre, Si $C_1 = M_1 \oplus K$ et $C_2 = M_2 \oplus K$ alors $C_1 \oplus C_2 = M_1 \oplus M_2$ dans lequel la clef n'apparait plus. Une analyse pourait alors déterminer des morceaux de M_1 ou M_2.

Exercice 7, page 105.

- Prenons $p = 47$ et $q = 59$ (Ces valeurs sont faibles et ne correspondent évidemment pas à des clé réelles.
- on calcule $n = p * q = 47 * 59 = 2773$
- on choisit e, premier par rapport à $\phi(n)$. Prenons par exemple $e = 17$.
- On calcule alors, par l'algorithme d'Euclide étendu[1], d tel que $d.e = 1 \bmod (p-1).(q-1)$, soit $d = 157$.

On a alors un couple clef publique (17 , 2773) clef privée (157 , 2773).
Pour chiffrer B c'est la valeur $01000010 = 66$ (voir table 2.1 à la page 50), on calculera (66^{17}) (mod 2773) c'est-à-dire 872. Pour retrouver le message d'origine on calculera (1553^{157}) (mod 872) qui donne bien 66.

[1] sous Maple par exemple, cet algorithme correspond à la commande **igcdex**

Exercice 9, page 110.

- Alice calcule $A = 2^{292} \bmod 541 = 69$ qu'elle transmet à Bob.
- Bob calcule $B = 2^{426} \bmod 541 = 171$ qu'il transmet à Alice.
- La clé secrète est alors $(171)^{292} \bmod 541 = 368 = (69)^{426} \bmod 541$

Exercice 11, page 112.
Pour garantir le secret et l'authentification il faut qu'un domaine commun existe entre A et B, ou plus précisément entre E_A, D_A, E_B et D_B.

A envoie alors $\tilde{M} = E_B(D_A(M))$ à B ; B calcule alors $D_B(E_A(\tilde{M})) = E_A(D_A(M)) = M$.

Un espion ne peut pas décoder $\tilde{M}$ car il ne dispose pas de D_B et le secret est donc préservé. Si un espion envoie un message M' à la place de $\tilde{M}$, M' ne pourra pas être décodé en un message valide M : en effet, pour cela il faudrait connaître D_A. Cela assure l'authenticité.

Exercice 12, page 116.
Oscar peut ajouter de nouveaux blocs à la fin du message et calculer ainsi une empreinte MAC valide.

Exercice 13, page 120.
$a^{z_1} y^{z_2} \equiv a^{sv} a^{xz_2} \equiv a^{xrv+k} a^{xz_2} \equiv a^{xrv+k} a^{xqv-xrv} \equiv a^k [p]$.

Exercice 14, page 147.
Si on reçoit un mot m' avec k erreurs par rapport au mot m émis ; l'erreur sera détectée ssi $m' \notin C$. Or, comme C est t-correcteur, on a $\forall x \in C$: $d(m, x) \geq 2t + 1$. Donc il n'existe pas de mot de C à une distance $\leq 2t$. Ainsi, si $d(m', m) = k \leq 2t$ alors $m' \notin C$ et on détecte l'erreur. Donc C est k-détecteur avec $k \geq 2t$.
Réciproque : elle est vraie : En effet, si C est k-détecteur et que $k \geq 2t$, alors on a $\delta(C) > k$, donc $\delta(C) \geq 2t + 1$ et par conséquent C est au moins t-correcteur.

Exercice 15, page 148.
Il suffit de calculer la distance minimale entre deux mots de code : il y en a $|V|^k$, qui sont codés avec $n = k + r$ chiffres. On suppose les mots de C stockés dans un tableau $C[1..2^k]$. Chaque mot $C[i]$ est un tableau $[1..k + r]$ de chiffres de V.

Application : $k = log_2(|C|) = 2$; donc $r = 10 - 2 = 8$. Le code C est un code $(10, 2)$. On a $\delta = 5$: C est donc 4-détecteur et 2-correcteur.

Exercice 16, page 148.

1. Il suffit de calculer $\delta(C)$. Soit $x \neq y$ deux mots de C.
   Alors $\exists i \big(0 \leq i \leq 3$ et $x_i \neq y_i\big)$. Soit e le nombre de ces indices i où les chiffres de x et y diffèrent :

Algorithme 12 Distance d'un code

```
delta := +infini ;
Pour i = 1 à 2^k Faire
  Pour j = i + 1 à 2^k Faire
    d_ij = 0 ; { calcul de la distance d_ij entre C[i] et C[j] }
    Pour l = 1 à k + r Faire
      Si (C[i][l] ≠ C[j][l]) Alors
        d_ij = d_ij + 1 ;
      Fin Si
    Fin Pour{ Mise à de jour de δ = Min (d_ij) }
    Si ( d_ij < δ ) Alors
      δ = d_ij
    Fin Si
  Fin Pour
Fin Pour
TauxDeDetection = δ − 1 ;
TauxDeCorrection = (δ−1)/2 ;
```

- si $e = 1$: alors, le bit en position i apparaît dans trois autres chiffres de x et y, à savoir les bits : $(b_i + b_k, b_i + b_l, b_0 + b_1 + b_2 + b_3)$. Donc $d(x, y) \geq 4$.
- si $e = 2$: x et y ont exactement 2 bits en position i et j $(0 \leq i < j \leq 3)$ différents. Alors, comme chaque bit b_i apparaît dans deux chiffres de contrôle de la forme $b_i + b_l$ et $b_i + b_m$ et idem pour j, x et y différent au moins pour deux de ces bits. Ainsi $d(x, y) \geq 4$.
- Si $e = 3$: x et y ont 3 bits de différents. Donc x et y diffèrent aussi pour leur dernier bit $(b_0 + b_1 + b_2 + b_3)$. Ainsi $d(x, y) \geq 4$.
- Si $e = 4$: alors $d(x, y) \geq 4$.

Ainsi $\delta(C) \geq 4$. Les deux mots $a = 000000000$ et $b = 110000110$ sont dans C et ont une distance de 4 : finalement $\delta = 4$.

2. En reprenant les deux mots précédents : soit $c = 100000010$. $d(a, c) = d(b, c) = 2$ et aucun mot de code n'est à une distance 1 de c. Donc l'erreur n'est pas corrigible.
3. $\delta = 4 \Longrightarrow C$ est 3-détecteur $(\delta - 1 = 3)$.
   a est obtenu à partir b en changeant 4 bits : donc C n'est pas 4-détecteur.

Exercice 17, page 149.
Considérons un code avec r bits de redondance, i.e. $(n, 4)$ avec $n = 4+r$. Pour avoir un code 1-correcteur il est nécessaire que $1 + n \leq 2^{n-4}$, i.e. $2^r - r \geq 5$. Le nombre minimal r assurant cette condition est $r = 3$.

Exercice 18, page 149.

a. L'inégalité du théorème 22 s'écrit : $1 + n \leq 2^{n-k}$, soit $1 + k + r \leq 2^r$.

b. Le rendement du code est $\frac{k}{r+k}$; donc maximiser le rendement à r fixé revient à maximiser k. L'inégalité du théorème 22 permet alors d'avoir une borne sur le plus grand k possible et donc une borne pour le rendement R.
Avec $r = 3$, on a nécessairement $n \leq 7$ et donc $k \leq 4$. D'où $R \leq \frac{4}{7}$.
Avec $r = 4$: $k \leq 11$ et $R \leq \frac{11}{15}$.
Avec $r = 5$: $k \leq 26$ et $R \leq \frac{26}{31}$.
Avec $r = 6$: $k \leq 57$ et $R \leq \frac{57}{63}$.

c. La longueur d'un code binaire 1-parfait vérifie d'après le théorème 22 (calcul analogue au a.) : $n = 2^r - 1$. Il n'existe donc pas de code 1-parfait de longueur 2^m.

Index

Bibliographie

[1] Eric Bach and Jeffrey Shallit. *Algorithmic Number Theory — Efficient Algorithms*, volume I. MIT Press, Cambridge, USA, 1996. ISBN : 0-262-02405-5.

[2] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Hmac : Keyed-hashing for message authentication (rfc 2104), February 1997. `http://www.ietf.org/rfc/rfc2104.txt`

[3] Michael Ben-Or. Probabilistic algorithms in finite fields. In *22th Annual Symposium on Foundations of Computer Science*, pages 394–398, Los Alamitos, California, USA, October 1981. IEEE Computer Society Press.

[4] David M. Burton. *Elementary number theory.* International series in Pure and Applied Mathematics. McGraw-Hill, 4^{th} edition, 1998.

[5] David G. Cantor and Hans Zassenhaus. A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, 36(154) :587–592, April 1981.

[6] Don Coppersmith. Fast evaluation of logarithms in fields of characterstic two. *IEEE Transactions on Information Theory*, IT-30 :587–594, 1984.

[7] J. Daemen and V. Rijmen. "The Block Cipher Rijndael". In J.-J. Quisquater and B. Schneier, editors, *Proceedings of the Third Smart Card Research and Advanced Applications Conference*, pages 288–296. Springer-Verlag, 2000.

[8] J. Daemen and V. Rijmen. "Rijndael, the advanced encryption standard". *Dr. Dobb's Journal*, 26(3) :137–139, March 2001.

[9] Michel Demazure. *Cours d'algèbre. Primalité, Divisibilité, Codes*, volume XIII of *Nouvelle biliothèque Mathématique.* Cassini, Paris, 1997.

[10] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. The hash function ripemd-160, 1997. `http://www.esat.kuleuven.ac.be/~bosselae/ripemd160.html`

[11] Peter D. T. A. Elliott and Leo Murata. On the average of the least primitive root modulo p. *Journal of The london Mathematical Society*, 56(2) :435–454, 1997.

[12] Federal Information Processing Standards. Publication 186 : Digital signature standard (dss), May 1994. `http://www.itl.nist.gov/fipspubs/fip186.htm`

[13] Federal Information Processing Standards. Publication 180-1 : Secure hash standard, April 1997. `http://www.itl.nist.gov/fipspubs/fip180-1.htm`

[14] Federal Information Processing Standards. Publication 196 : Entity authentication using public key cryptography, February 1997. `http://www.itl.nist.gov/fipspubs/fip196.htm`

[15] Federal Information Processing Standards. Publication 197 : Advanced encryption standard, October 2000. `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf`

[16] Ariel Futoransky and Emiliano Kargieman. Security advisory, ssh insertion attack, June 1998. `www.cotse.com/exploits/netapps/encryption/ssh-insertion-attack.txt`

[17] Torbjörn Granlund. *The GNU multiple precision arithmetic library*, 2002. Version 4.1, `www.swox.com/gmp/manual`.

[18] Daniel J.Barrett and Richard E.Silverman. Ssh, the definitive shell, 2001.

[19] Donald E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, MA, USA, troisième edition, 1997.

[20] Neal Koblitz. *A course in number theory and cryptography*, volume 114 of *Graduate texts in mathematics*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK /, etc., 1987.

[21] Leo Murata. On the magnitude of the least prime primitive root. *Journal of Number Theory*, 37(1) :47–66, 1991.

[22] M. Myers, C. Adams, D. Solo, and D. Kemp. Internet x.509 certificate request message format (rfc 2511), March 1999. `http://www.ietf.org/rfc/rfc2511.txt`

[23] Tomás Oliveira e Silva. Least primitive root of prime numbers, January 2000. `http://www.ieeta.pt/~tos/p-roots.html`

[24] Alain Poli and Llorenç Huguet. *Codes Correcteurs : Théorie et applications*. Logique, Mathématiques, Informatique. Masson, 1989.

[25] Ronald Rivest. The md5 message-digest algorithm (rfc 1321), April 1992. `http://www.faqs.org/ftp/rfc/rfc1321.txt`

[26] J. Barkley Rosser and Lowell Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6 :64–94, 1962.

[27] Bruce Schneier. *Secrets et mensonges : sécurité numérique dans un monde en réseau.* Vuibert informatique, 2000.

[28] Bruce Schneier. *Cryptographie appliquée, Algorithmes, protocoles et codes source en C.* Vuibert, Paris 2eme édition, 2001.

[29] Victor Shoup. Searching for primitive roots in finite fields. *Mathematics of Computation*, 58(197) :369–380, January 1992.

[30] Victor Shoup. Fast construction of irreducible polynomials over finite fields. *Journal of Symbolic Computation*, 17(5) :371–391, May 1994.

[31] Victor Shoup. NTL 5.3 : A library for doing number theory, 2002. `www.shoup.net/ntl`.

[32] Douglas Stinson. *Cryptographie : théorie et pratique.* International Thomson Publishing France, Paris, 1996.

[33] T. Ylonen. The ssh (secure shell) remote login protocol, 1996. `http://kcgl1.eng.ohio-state.edu/~jonesd/ssh/DOC/ssh-rfc-nov95.txt`