Cryptosystème et signature numérique d'El Gamal

Nom - numéro du candidat

TIPE 2018

Sommaire

1 Introduction **2**

2 Le cryptosystème d'El Gamal **2**
2.1 Le Groupe $(\mathbb{Z}/p\mathbb{Z}^*, \times)$ . . . 2
2.2 Chiffrement d'El Gamal . . . 2
2.3 Signature d'El Gamal . . . 2
2.4 Problème du logarithme discret . . . 3
2.5 Génération des nombres premiers . . . 3
2.5.1 Méthode déterministe . . . 3
2.5.2 Méthode probabiliste : test de Miller-Rabin . . . 3
2.6 Génération des éléments primitifs du groupe $(\mathbb{Z}/p\mathbb{Z}^*, \times)$ . . . 4

3 Application : mise en place d'une messagerie sécurisée **6**
3.1 GitHub . . . 6
3.2 Architecture client-serveur . . . 6
3.3 Les opérations élémentaires . . . 8
3.3.1 Puissances modulaire . . . 8
3.3.2 Inverse modulaire . . . 9
3.4 L'algorithme de chiffrement symétrique : le DES . . . 11
3.4.1 Présentation du DES . . . 11
3.4.2 Principe de fonctionnement . . . 11
3.4.3 Générations des sous-clés . . . 11
3.4.4 Processus de chiffrement . . . 11

4 Analyse de la messagerie sécurisée **13**
4.1 Attaque du cryptosystème d'El Gamal : l'algorithme de Shanks . . . 13
4.2 Attaque du cryptosystème d'El Gamal : calcul de l'indice . . . 14
4.3 Comparaison avec le chiffrement RSA . . . 14
4.3.1 Fonctionnement du RSA . . . 14
4.3.2 Attaque du cryptosystème RSA : l'algorithme rho de Pollard . . . 15
4.4 Attaque sur la signature d'El Gamal . . . 15

5 Programmes (Python 3) **16**
5.1 Génération des nombre premiers : déterministe et probabiliste (Miller-Rabin) . . . 18
5.2 Client-serveur . . . 21
5.2.1 Serveur . . . 21
5.2.2 Client . . . 22
5.3 Opérations élémentaires . . . 24
5.4 Data Encryption Standard (DES) . . . 25
5.5 Attaques El Gamal : algorithme de Shanks et calcul de l'indice . . . 37
5.6 RSA . . . 38
5.7 Attaques RSA : Crible algébrique et algorithme rho de Pollard . . . 39

1 Introduction

2 Le cryptosystème d'El Gamal

2.1 Le Groupe $(\mathbb{Z}/p\mathbb{Z}^*, \times)$

Le groupe $(\mathbb{Z}/p\mathbb{Z}^*, \times)$, avec p **premier**, vérifie les propriétés d'un **groupe commutatif** :

- $\times$ est une loi de composition *interne*
- *Associativité* : $\forall\ (a,b,c)\ \in\ (\mathbb{Z}/p\mathbb{Z}^*)^3\ ,\ (a \times b) \times c\ [p] = a \times (b \times c)\ [p]$
- *Element neutre* : $\forall\ a\ \in\ \mathbb{Z}/p\mathbb{Z}^*\ ,\ a \times 1\ [p] = 1 \times a\ [p] = a\ [p]$
- Tout élément de $\mathbb{Z}/p\mathbb{Z}^*$ est *symétrisable* : $\forall\ a\ \in\ \mathbb{Z}/p\mathbb{Z}^*\ \ \exists\ b\ \in \mathbb{Z}/p\mathbb{Z}^*\ |\ a \times b \equiv 1\ [p]$ (p étant premier, d'après le petit théorème de Fermat, b existe)
- *Commutativité* : $\forall\ (a,b)\ \in\ (\mathbb{Z}/p\mathbb{Z}^*)^2\ ,\ a \times b\ [p] = b \times a\ [p]$

2.2 Chiffrement d'El Gamal

Le **chiffrement d'El Gamal** (1984) est une **variante du protocole Diffie-Hellman** qui repose sur le **problème du logarithme discret** (*cf.* §2.4).

Soit p un nombre **premier** et g un élément **générateur** de $(\mathbb{Z}/p\mathbb{Z}^*, \times)$.

Le destinataire B dispose :

— d'une **clé privée** s qui appartient à l'ensemble $\{1, \ldots, p-1\}$

— d'une **clé publique** égale à $g^s\ [p]$

Lorsque A veut **transmettre un message chiffré** à B, il doit :

1. Choisir un **aléa** k dans $1, \ldots, p-1\}$.
2. Calculer la **clé de session** $K = (g^s)^k\ [p]$.
3. Chiffrer son message M à l'aide d'un **algorithme de chiffrement symétrique** quelconque (DES par exemple) pour obtenir le cryptogramme C (dans la présentation originale de ce système, le message M était un élément de $(\mathbb{Z}/p\mathbb{Z}^*, \times)$ et le chiffrement consistait simplement en une multiplication par K modulo p).
4. Transmettre le couple (g^k, C). La quantité g^k étant l'**entête** du cryptogramme.

Pour **déchiffrer** le message, B doit :

1. Calculer la **clé de session** $K = (g^k)^s\ [p]$ à l'aide l'entête et de sa clé privé.
2. Déchiffrer le message C en utilisant le **même algorithme de (dé)chiffrement symétrique** que A à l'aide de K.

2.3 Signature d'El Gamal

Ce **mécanisme de signature** repose aussi sur la difficulté du problème du logarithme discret (*cf.* §2.4).

Paramètres :

— un nombre **premier** p et un **générateur** g du groupe multiplicatif de $(\mathbb{Z}/p\mathbb{Z}^*, \times)$

— la **clé privée** est un entier non nul x dans $\{1, \ldots, p-2\}$

— la **clé publique** est g^x $[p]$

Calcul d'une signature :

1. Choisir un entier k **aléatoire** dans $\{1, \ldots, p-1\}$.
2. Calculer $r = g^k$ $[p]$.
3. Résoudre l'équation dite **équation de signature d'inconnue** s : $\boxed{h = xr + ks}$ dans $\mathbb{Z}/(p-1)\mathbb{Z}^*$.

La **solution de l'équation de signature** est : $s = (h - rx)k^{-1}$ dans $(\mathbb{Z}/(p-1)\mathbb{Z}^*, \times)$.
La **signature du message** de condensé h est le couple (r, s).
La **vérification de la signature** consiste à **tester l'égalité** $g^h = (g^x)^r \times r^s$ dans $(\mathbb{Z}/p\mathbb{Z}^*, \times)$.

2.4 Problème du logarithme discret

Tout le cryptosystème d'El Gamal repose sur le fait qu'il est "difficile" de **trouver l'entier k** à partir de g^k $[p]$ dans $(\mathbb{Z}/p\mathbb{Z}^*, \times)$: c'est ce qu'on appelle le **problème du logarithme discret**.

2.5 Génération des nombres premiers

2.5.1 Méthode déterministe

1. Choix d'un nombre p aléatoire de taille n.
2. Test des entiers inférieurs à $\sqrt{p}$ comme diviseurs de p.
3. Si aucun entier divise p, alors p est premier.

Complexité : $O(\sqrt{10^n})$

2.5.2 Méthode probabiliste : test de Miller-Rabin

1. Choix d'un nombre p aléatoire de taille n.
2. Effectuer k tests de Miller-Rabin.
3. Si tous les tests sont des succès alors p est *pseudo-premier* et a une probabilité de ne pas être premier de $\frac{1}{4^k}$; sinon on choisit un nouveau p aléatoire et on recommence.

Détails théoriques du test :
Si p est premier alors $\mathbb{Z}/p\mathbb{Z}$ est un corps (résultat du cours qui s'obtient avec le théorème de Bezout).
On s'intéresse à l'équation :

$$x^2 = 1$$

dans $\mathbb{Z}/p\mathbb{Z}$. Comme c'est un corps, c'est en particulier un anneau intègre. En factorisant l'équation ci-dessus sous la forme :

$$(x-1)(x+1) = 0$$

on s'aperçoit que les seules solutions de cette équation, par intégrité, sont 1 et -1. **(1)**

Soit p premier.
On écrit p sous la forme :

$$p = 2^s \times d + 1$$

(s est le nombre maximal de fois que l'on peut mettre 2 en facteur dans $p-1$).
Soit a (appelé *témoin de Miller*) dans $\mathbb{Z}/p\mathbb{Z}$, on pose la suite $(b_i)_{0\leq i\leq s}$ avec $b_i = a^{2^i} \times d$
Par le petit théorème de Fermat, on sait que $b_s \equiv 1[p]$. On montre grâce à (1) que s'il existe k tel que b_k ne soit pas congru à 1 modulo p alors il existe un indice $i \in \{k, \ldots, s-1\}$ tel que $b_i \equiv -1[p]$. **(2)**

En effet, posons $i = \sup\{j \mid b_j$ ne soit pas congru à 1 modulo $p\}$. Un tel nombre existe car b_k n'est pas congru à 1 modulo p (l'ensemble que l'on considère est non vide), et pour tout $j \geq s$, $b_i \equiv 1[p]$ (l'ensemble est majoré par s). Ainsi, on a $i \in \{0, \ldots, s-1\}$. D'autre part $b_i^2 = b_{i+1} \equiv 1[p]$.
D'après (1), $b_i \equiv -1[p]$. La première possibilité est exclue par définition de i. Donc $b_i \equiv 1[p]$.

Le test de Miller-Rabbin consiste à calculer la suite des b_i pour voir si elle vérifie bien (2). On dit que p **passe le test** si les b_i satisfont effectivement (2). Si p ne **passe pas le test** alors on est sûr que p n'est **pas premier** sinon on ne peut rien dire. Il reste à itérer ce processus avec plusieurs témoins de Miller, l'heuristique du système étant ***« plus p passe de tests, plus il a de chance d'être premier »***.
L'efficacité de ce test vient du fait que le nombre de témoin « menteur », c'est-à-dire les témoins qui permettent à p de passer le test alors qu'il n'est effectivement pas premier, est inférieur à $1/4$ du nombre d'entiers inférieurs à p.

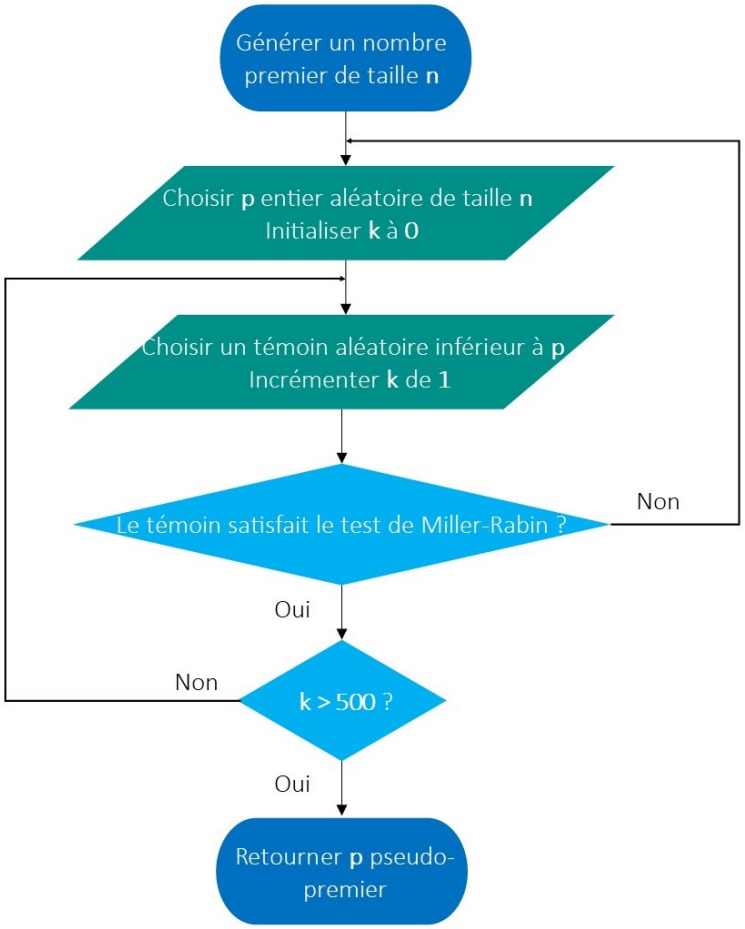

FIGURE 1 – Fonctionnement du test de primalité de Miller-Rabin

2.6 Génération des éléments primitifs du groupe $(\mathbb{Z}/p\mathbb{Z}^*, \times)$

Pour mettre en place le cryptosystème d'El Gamal, il faut avoir à disposition un **nombre p premier très grand** (au moins 100 chiffres) et ensuite **trouver un générateur** g du groupe

$(\mathbb{Z}/p\mathbb{Z}^*, \times)$. Nous avons choisi de **prendre le problème à l'envers** en cherchant d'abord l'élément générateur g puis un très grand nombre premier p.

Les **nombres de Sophie Germain** sont des nombres q tel que :

- q est premier
- $2q + 1$ est aussi premier

En partant de cette idée, nous avons **expérimenté une méthode** pour **trouver un couple de nombre premier** très grand, qui offre la possibilité de mettre en place le cryptosystème efficacement.

Tout d'abord il faut trouver un nombre q premier puis chercher un nombre p premier de la forme $p = kq + 1$, en faisant **varier** k. En testant cette méthode sur machine, nous avons conjecturé qu'il est toujours **possible de trouver** p qui convient avec des **valeurs de** k **de l'ordre de quelques centaines** (la valeur maximale atteinte par k était 5000).
Grâce à cette méthode, les **facteurs de** $p - 1$ **sont maintenant connus** (ce qui est presque impossible avec un nombre à 200 chiffres pris au hasard) et peu nombreux. En effet les f**acteurs de** $p - 1$ sont tous les **nombres qui se mettent sous la forme** z **ou** zq où z est un diviseur de k. D'après le théorème de Lagrange tous ces nombres sont susceptibles d'être **l'ordre d'un élément de** $(\mathbb{Z}/p\mathbb{Z}^*, \times)$.
Dès lors, si r **est l'ordre d'un élément** x et si r n'est pas un diviseur de k (ce que nous pouvons tester facilement) alors r **s'écrit sous la forme** zq et est donc très grand (de l'ordre de grandeur p à $log(k)$ près). x est ainsi le **générateur d'un sous-groupe** de $(\mathbb{Z}/p\mathbb{Z}^*, \times)$ et possède le **même nombre d'élément** que $(\mathbb{Z}/p\mathbb{Z}^*, \times)$ en ordre de grandeur.

En implémentant cette méthode, nous nous sommes aperçu qu'il est aisé de trouver un élément x **générateur d'un grand sous-groupe de** $(\mathbb{Z}/p\mathbb{Z}^*, \times)$.

3 Application : mise en place d'une messagerie sécurisée

3.1 GitHub

GitHub est un service d'hébergement et de gestion de projets en ligne que nous avons utilisé pour **gérer les codes sources** de nos programmes. Les programmes, codés en **Python 3** dans le cadre des TIPE, sont en *Open Source* à l'adresse `github.com/ludothai/TIPE-2018`.

3.2 Architecture client-serveur

Architecture Client-Serveur est l'architecture utilisée pour la majorité des communications sur internet. Le module « socket » de Python est utilisé pour ouvrir des connexions utilisant le **protocole TCP/IP**. Le serveur est donc repéré par son adresse IP et le port qu'il dédie à la messagerie. Une programmation dite « en parallèle » est mise en place à l'aide du module `Thread` de Python pour permettre la réception et l'envoi simultané de messages.

Procédure de Handshake :
A chaque **connexion d'un client** sur le serveur, il doit renseigner **un pseudo** qui permettra aux autres utilisateurs de l'identifier, et **une clé publique**. Le serveur **stocke ces informations et les rend accessibles** aux autres utilisateurs.
Le serveur est donc bien un **canal public** puisque tous les utilisateurs peuvent accéder à tous les messages, même ceux qui ne leur sont pas adressés. D'où l'importance du cryptage.

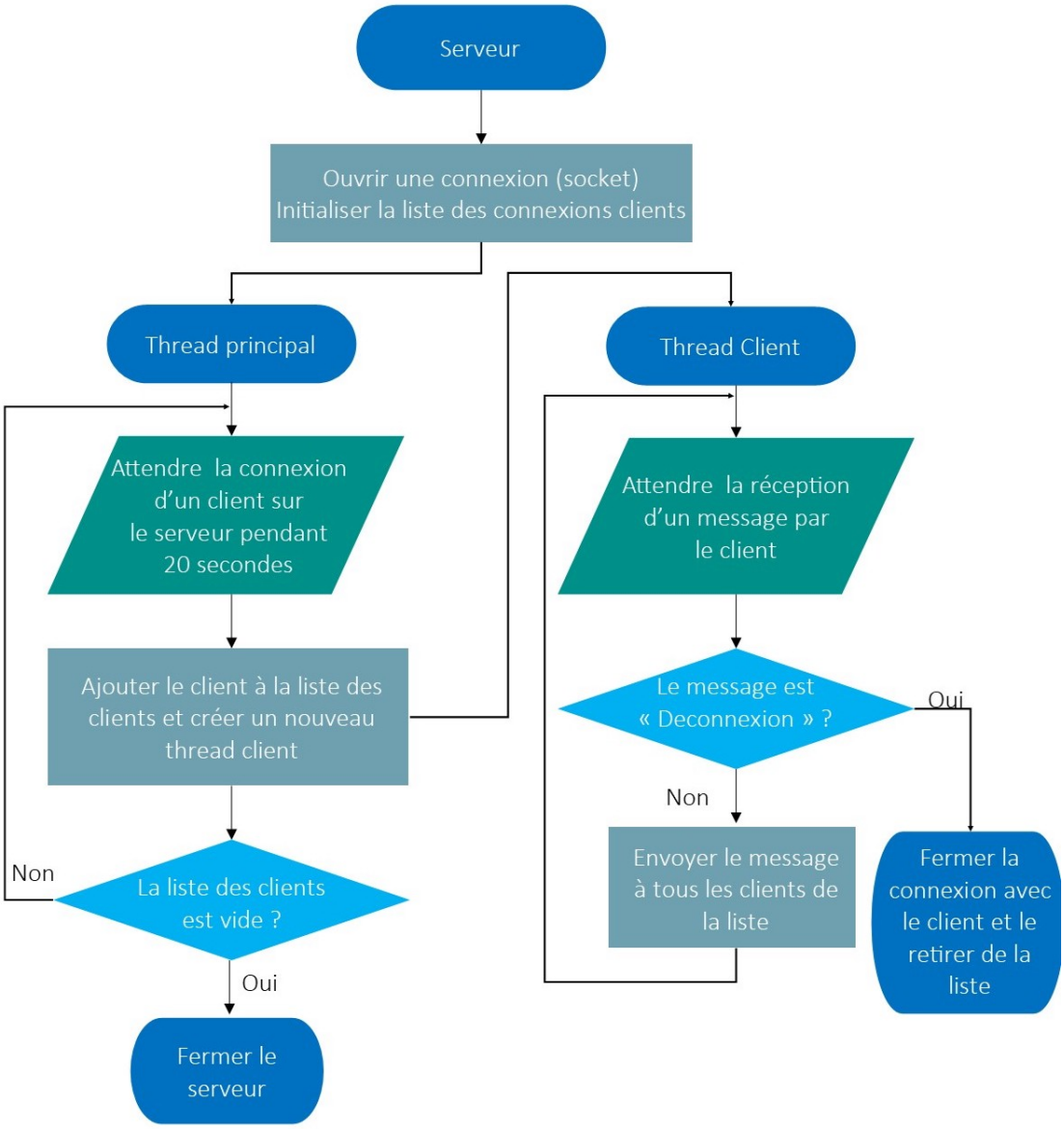

FIGURE 2 – Schéma de fonctionnement du serveur

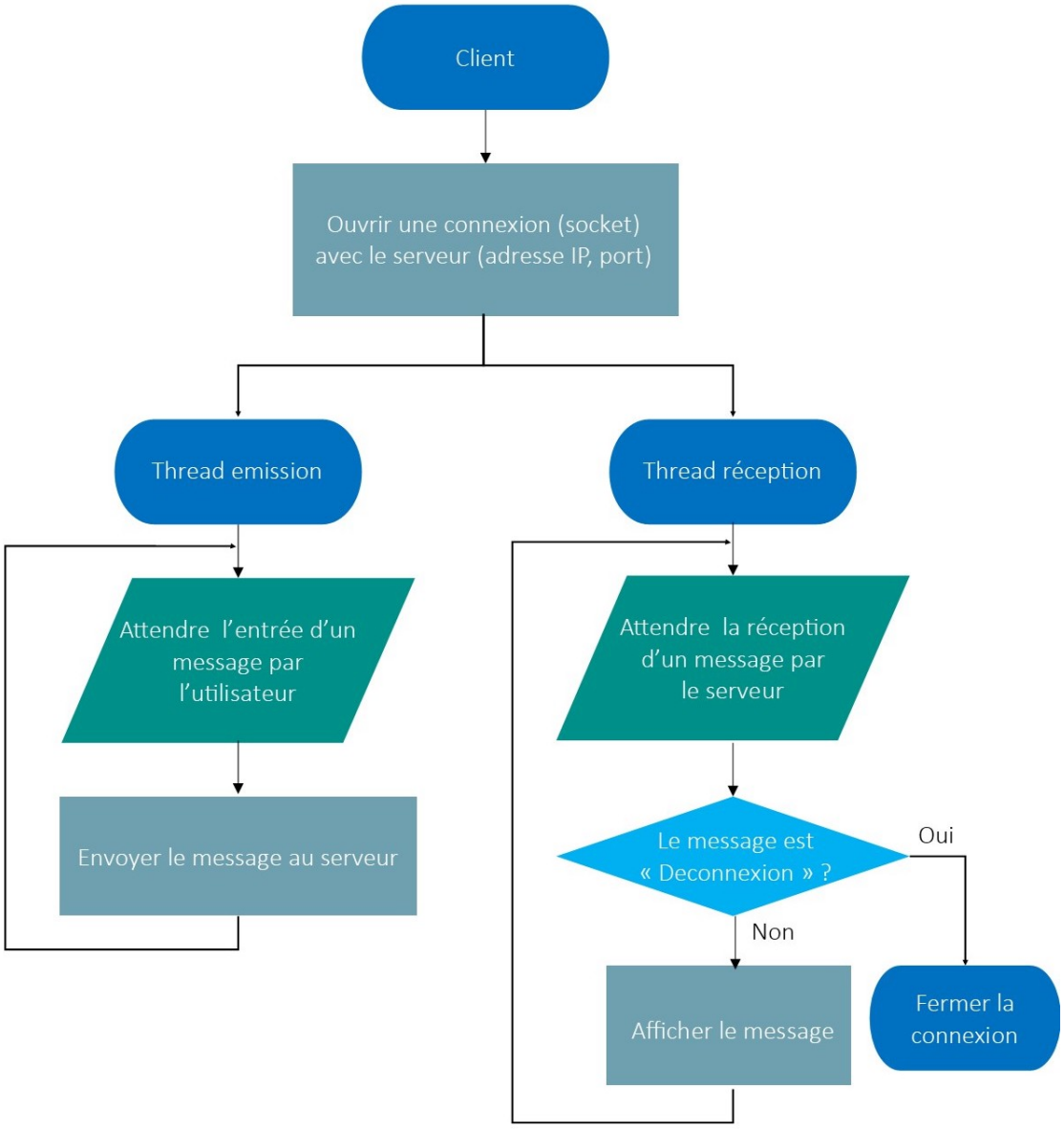

FIGURE 3 – Schéma de fonctionnement du client

3.3 Les opérations élémentaires

3.3.1 Puissances modulaire

Méthode : Exponentiation rapide en base 2

Variables :

- **res** : résultat
- **a** : base à la puissance 2^i avec i l'indice de la boucle en cours

Etapes :

1. Conversion de l'exposant en binaire.
2. On parcourt l'exposant en binaire ($i$ème coefficient) : si le coefficient vaut 1 on multiplie le résultat par a et on en extrait le reste modulo n. Puis met a au carré.
3. On retourne le résultat.

Complexité : $O(log(a))$

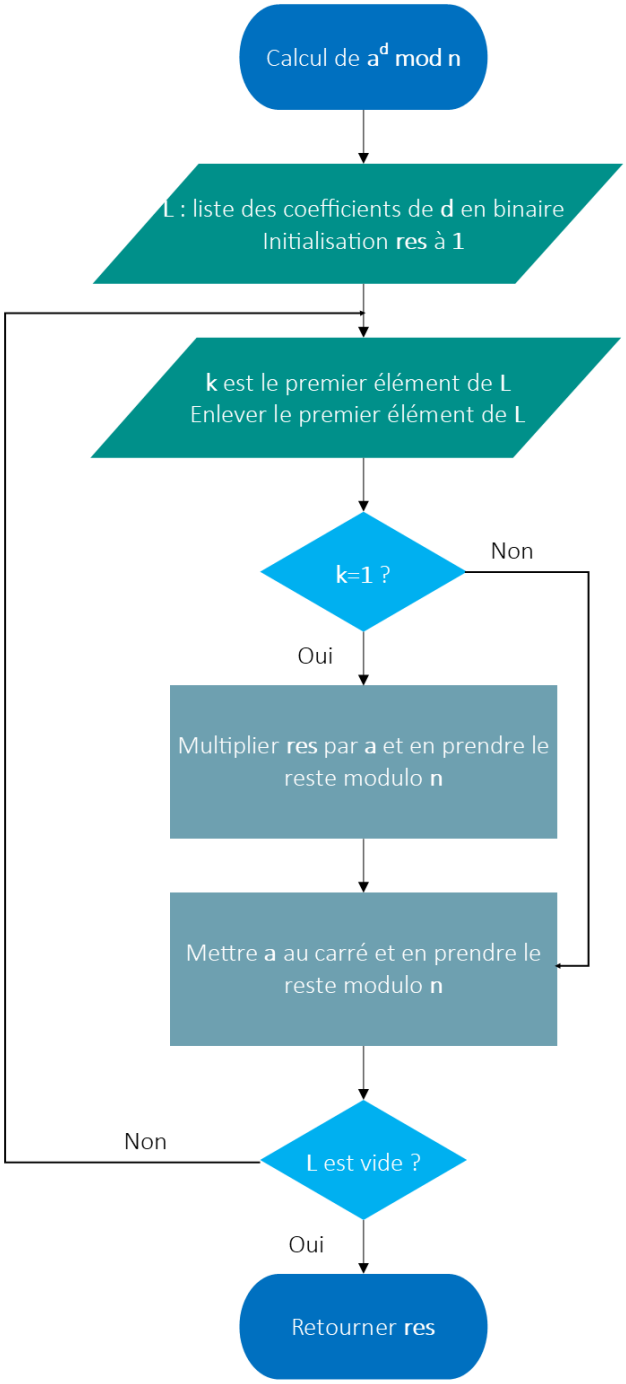

FIGURE 4 – Schéma du calcul d'une puissance modulaire

3.3.2 Inverse modulaire

Méthode : Algorithme d'Euclide étendu (recherche des coefficients de l'équation de Bézout)
Le but est de calculer r_i, u_i, v_i tel que $r_i = u_i a + v_i p$

Etapes :

Tant que $r_i \neq 0$

Faire

$q = E\left(\frac{r_i - 1}{r_i}\right)$

$r_{i+1} = r_{i-1} - qr_i$

$u_{i+1} = u_{i-1} - qu_i$

$v_{i+1} = v_{i-1} - qv_i$

Retourner u_i

En effet, $1 = u_i a + v_i p$ donc $u_i a \equiv 1[p]$

Complexité : $O((log(p))^2)$

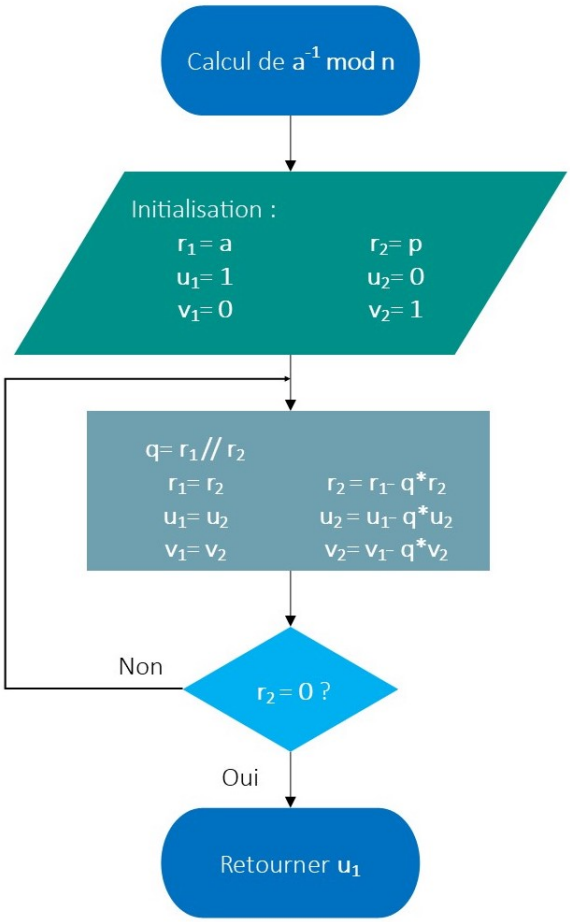

FIGURE 5 – Schéma du calcul d'une inversion modulaire

3.4 L'algorithme de chiffrement symétrique : le DES

3.4.1 Présentation du DES

Le "DES" qui signifie *Data Encryption Standard* est un algorithme de chiffrement qui permet de **chiffrer un message** à partir de la clé de chiffrement qui lui ait fourni. On dit que c'est un algorithme de chiffrement **symétrique** car les étapes pour chiffrer et déchiffrer les messages sont les mêmes.
L'algorithme DES est un algorithme de **chiffrement par bloc** puisqu'il transforme des blocs de 64 bits en d'autres blocs de 64 bits en manipulant des clés individuelles de 64 bits ramenées à 56 bits (un bit de chaque octet est utilisé pour le contrôle de parité).

3.4.2 Principe de fonctionnement

Une fois qu'un message (binaire) est découpé et complété en bloc de 64 bits, on applique le DES (fonction f sur la Figure) à chaque bloc avant de les rassembler en un seul bloc constituant le message crypté.

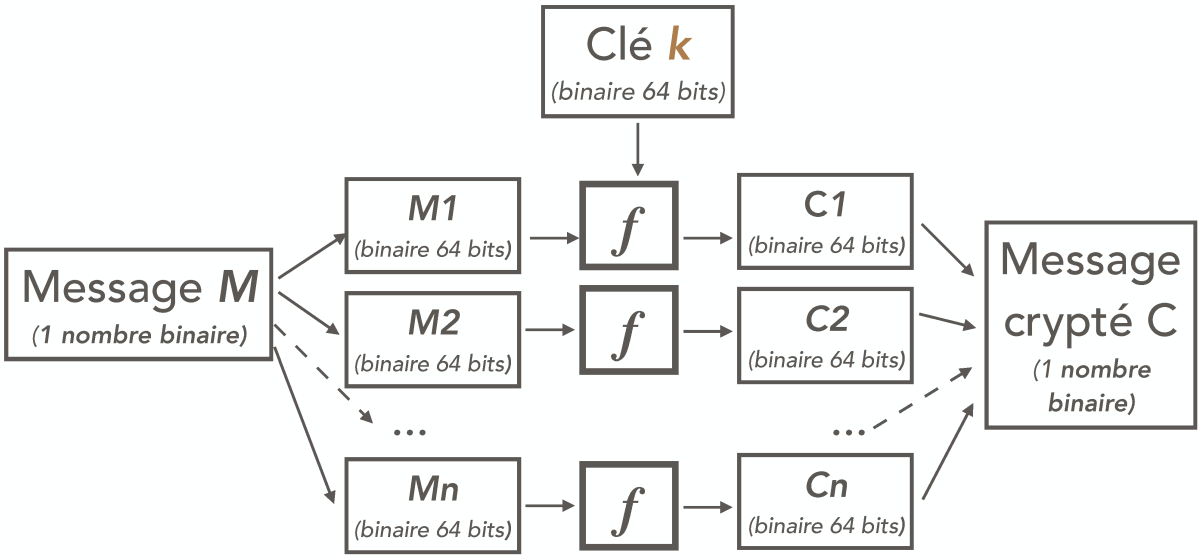

FIGURE 6 – Découpage en blocs pour appliquer DES

Le DES est constitué de 2 parties. Tout d'abord il y a **génération des 16 sous-clés** de 48 bits à partir d'une clé de 64 bits, puis il y a le **processus de chiffrement** pour chaque bloc de 64 bits.

3.4.3 Générations des sous-clés

3.4.4 Processus de chiffrement

Le chiffrement DES, constitué de permutations et de substitutions, peut être découpé en 3 étapes :

1. permutation initiale du bloc
2. 16 itérations appelées *Ronde*
3. permutation finale

Après une **permutation initiale Pi**, le bloc de 64 bits est **séparé en 2 blocs** de 32 bits (G0 à gauche et D0 à droite) qui vont effectuer **16 itérations** appelées *Ronde*. Chaque ***Ronde*** prend donc en entrée 2 blocs de 32 bits (un bloc à droite et un bloc à gauche) et retourne en sortie 2 autres blocs de 32 bits :

— le **bloc de gauche en sortie** est le **bloc de droite en entrée**.

— le **bloc de droite en sortie** est calculé de la manière suivante : le bloc de droite en entrée est d'abord **étendu** en un bloc de 48 bits à l'aide de la fonction **d'extension E**, puis un ***OU exclusif*** est effectué entre le **bloc étendu et la sous-clé K*i*** correspondant à l'itération. Ce nouveau bloc de 48 bits passe ensuite par une **fonction de substitution S** avant de réaliser un ***OU exclusif*** **avec le bloc de gauche en entrée** pour donner le **bloc de droite en sortie**.

Enfin, après les 16 *Rondes*, les blocs de 32 bits (G16 et D16) vont être regroupé pour reformer un bloc de 64 bits qui va subir une **permutation finale Pf**, donnant ainsi un bloc crypté.

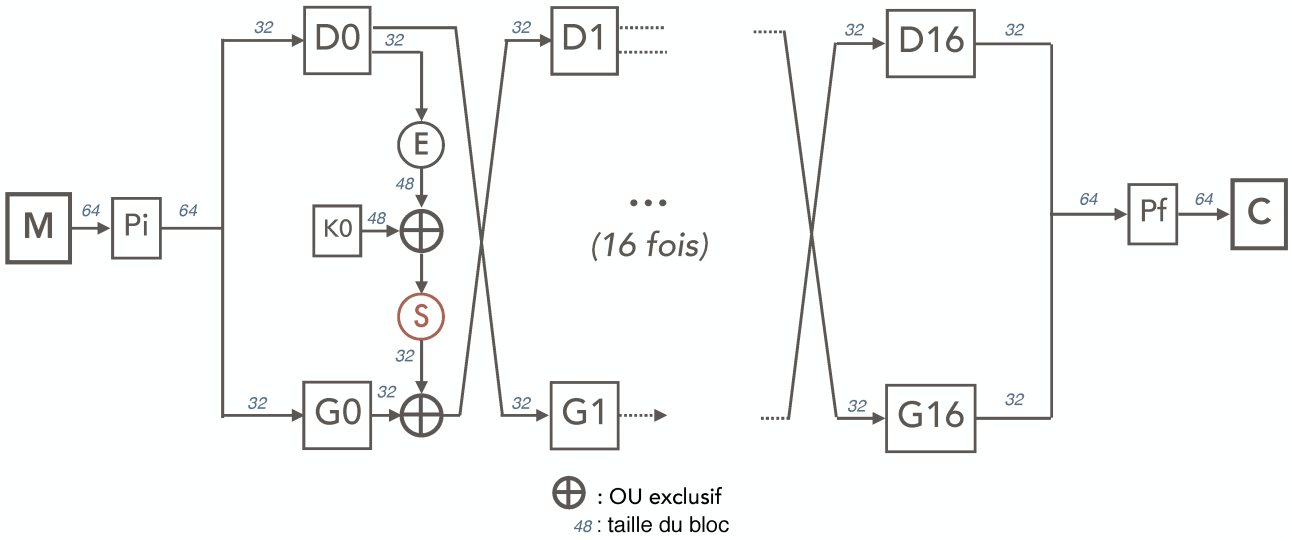

FIGURE 7 – Chiffrement DES

Fonction de sélection S :
La fonction de sélection S prend en entrée un bloc de 48 bits et renvoie en sortie un bloc de 32 bits. Tout d'abord, le bloc de 48 bits est décomposé en 8 bloc de 6 bits, numérotés de 1 à 6. Ensuite, pour chacun des 8 blocs de 6 bits, un nombre de 4 bits est retournée après sélection dans la matrice S_i (i allant de 1 à 8) correspondante. La sélection de la valeur se fait de la manière suivante :

— les bits 1 et 6 correspondent aux 2 bits de positionnement *ligne*, converti en entier pour donner le numéro de la ligne de la matrice S_i

— les bits 2 à 5 correspondent aux 4 bits de positionnement *colonne*, converti en entier pour donner le numéro de la colonne de la matrice S_i

Une fois l'entier récupéré dans la matrice S_i associée, celui ci est converti en nombre binaire de 4 bits.
Enfin, les 8 nombres binaires de 4 bits sont concaténé pour former un bloc de 32 bits.

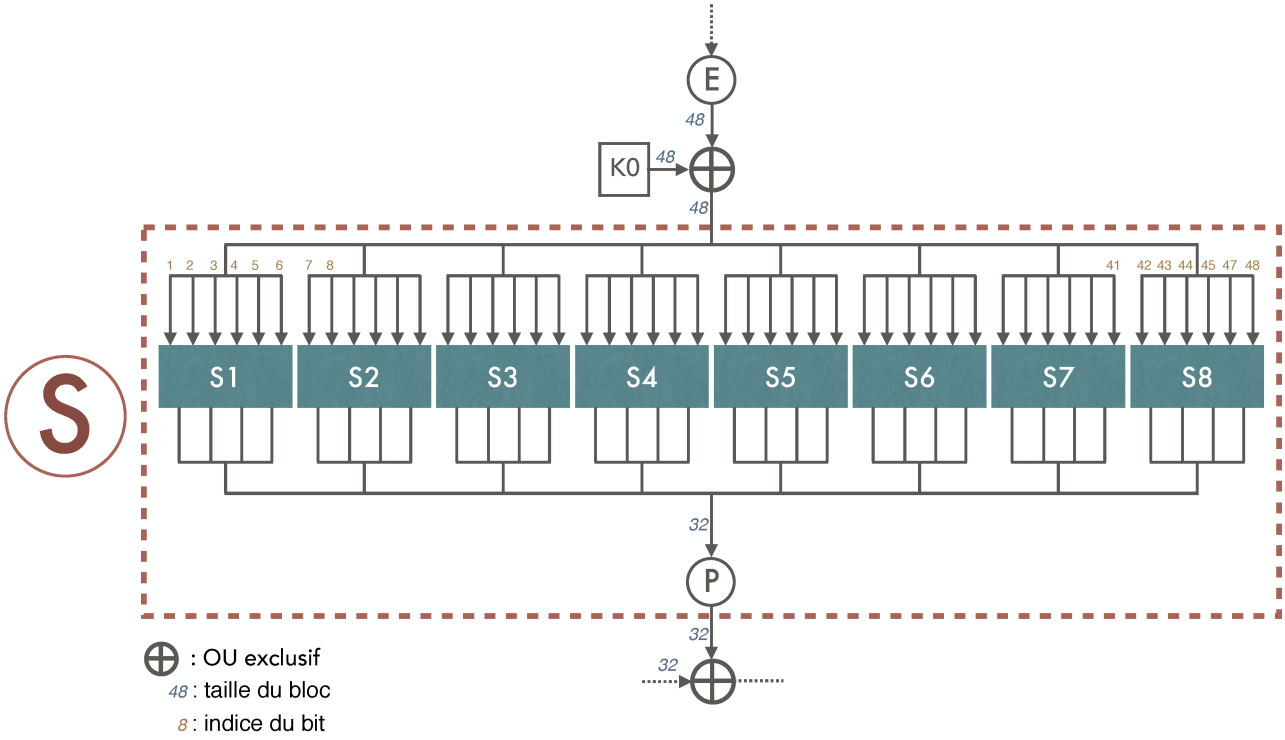

FIGURE 8 – Fonction de sélection S

4 Analyse de la messagerie sécurisée

4.1 Attaque du cryptosystème d'El Gamal : l'algorithme de Shanks

Il s'agit d'un algorithme de **calcul du logarithme discret**.

"Pas de bébé, pas de géant.", Shanks

Soit G un groupe cyclique d'ordre N et g un générateur. Etant donné un élément x dans G, il s'agit de **trouver l'entier n tel que** $x = g^n$. On pose $K = \sqrt{N}$. L'idée est de parcourir les éléments de G de deux façons :

— lors du **premier parcours, à pas-de-bébé**, on établit la **liste des puissances successives du générateur** : $A = \{g^i \mid i = 0, 1, ..., K-1\}$

— lors du **second parcours, à pas-de-géant**, on **saute de K en K** :
$B = \{xg^{-Kj} \mid j = 0, 1, ..., K-1\}$

Ces deux listes ont nécessairement un **élément commun** $g^i = xg^{-Kj}$, c'est-à-dire $x = g^{i+Kj} = g^n$ qui correspond à la division euclidienne de n par K.
La **complexité de l'algorithme est exponentielle** puisqu'il nécessite le stockage de $\sqrt{N}$ éléments et en **moyenne** $\frac{1}{2}\sqrt{N}$ **calculs** avant de trouver l'élément commun à A et à B.

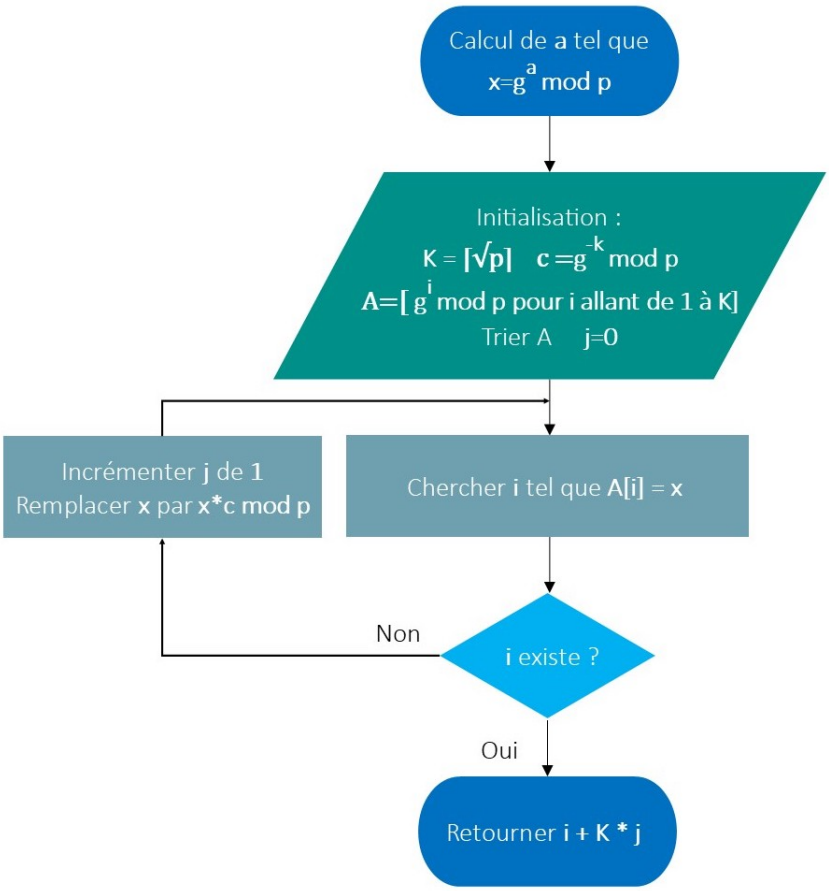

FIGURE 9 – Fonctionnement de l'algorithme de Shanks

4.2 Attaque du cryptosystème d'El Gamal : calcul de l'indice

Algorithme **le plus efficace** actuellement mais **difficile à implémenter** dont la **complexité est sous-exponentielle**.

4.3 Comparaison avec le chiffrement RSA

4.3.1 Fonctionnement du RSA

Le système RSA (1977) est un cryptosystème qui repose sur le **problème de factorisation des grands entiers**. C'est sans doute le plus populaire des systèmes à clés publiques.

La **clé privée** est constituée de **deux nombres premiers** p et q et d'un entier d premier avec $\phi(n) = (p-1) \cdot (q-1)$.
La clé publique est constituée du produit n de p et q et d'un entier e inverse de d modulo $\phi(n)$.
L'ensemble des messages est l'ensemble U_n des éléments inversibles modulo n.
L'opération de chiffrement est l'élévation à la puissance e modulo n. L'entier e étant l'exposant public.
L'opération de déchiffrement est l'élévation à la puissance d. L'entier d étant l'exposant privé.
D'après le théorème d'Euler, la composition de ces deux opérations est l'identité.

Le principe de la **signature RSA** est d'inverser les rôles des exposants e et d par rapport à leur utilisation dans le schéma de chiffrement. L'exposant privé d sert à chiffrer le condensé h du message. Seul le détenteur de la clé privée peut le faire. La signature est le résultat de ce chiffrement : $\sigma = h^d$. La vérification consiste à tester $\sigma^e = h$.

4.3.2 Attaque du cryptosystème RSA : l'algorithme rho de Pollard

L'idée de l'algorithme Rho est la suivante : si on parvient à trouver deux entiers distincts x et x' inférieurs à n et tels que $x \equiv x'[p]$, alors $x - x'$ est multiple de p. Comme n est également multiple de p, le PGCD de $x - x'$ et n sera multiple de p, ce qui signifie qu'il sera égal à p ou n. Ainsi, on teste successivement des paires (x, x') d'entiers de $[0, n-1]$ jusqu'à trouver une paire telle que le PGCD de $x - x'$ et n est supérieur à 1 mais inférieur à n.

Complexité : $O(n^{\frac{1}{4}})$

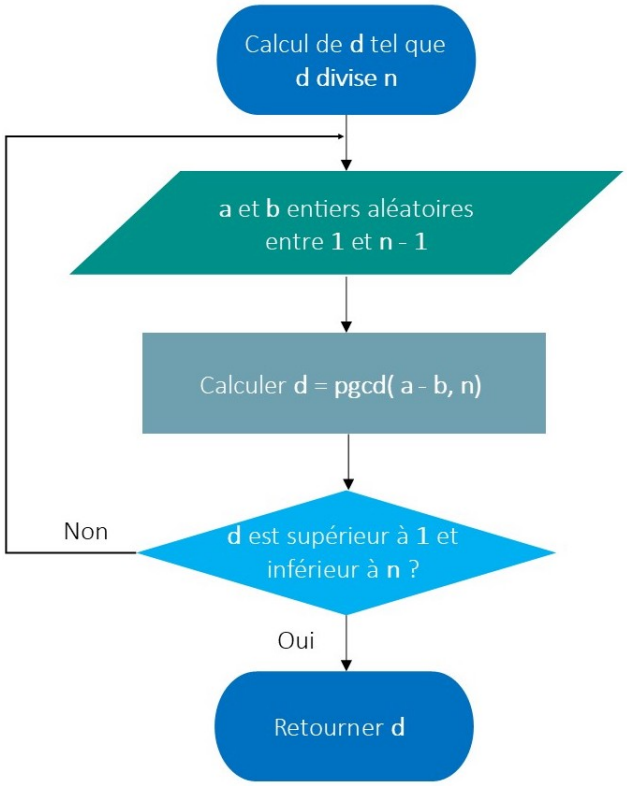

FIGURE 10 – Fonctionnement de l'algorithme rho de Pollard

4.4 Attaque sur la signature d'El Gamal

L'aléa de signature k doit être autant confidentiel que la clé privée, car la connaissance de ce dernier (en plus des données publiques) permet de remonter jusqu'à la clé privée x.
En effet si x est connu, et connaissant (r, s) (données publiques) alors il est aisé de trouver la solution x à : $x = (h - ks)r^{-1}$ dans $(\mathbb{Z}/(p-1)\mathbb{Z}^*, \times)$ (si r et $p-1$ sont premiers entre eux).

Par ailleurs, il est crucial que **l'aléa de signature k soit différent** pour chaque message. Dans le cas contraire, on peut également **remonter à la clé privée** x.
En effet si le même k est utilisé pour deux messages différents m_1 et m_2, de signature (r_1, s_1) et (r_2, s_2), alors il est possible de remonter à k et donc à x. k est solution des deux équations suivantes dans $(\mathbb{Z}/(p-1)\mathbb{Z}^*, \times)$:

- $m_1 = xr + ks_1$
- $m_2 = xr + ks_2$

Donc k est solution de : $k(s_1 - s_2) \equiv m_2 - m_1\ [p-1]$, qui est facilement résoluble si $s_1 - s_2$ et $p-1$ sont premiers entre eux.

5 Programmes (Python 3)

```
from random import randint
from math import log

def puissmod(a,d,n):
    """a**d mod n"""
    #iteratif : beaucoup plus efficace (10^-5) 600 chiffres -> 0.024571632396359178 s
    dbin=bin(d)
    L=[int(dbin[-i-1]) for i in range(len(dbin)-2)]
    res=1
    while L!=[]:
        k=L.pop(0)
        if k>0:
            res=res*a%n
        a=a**2%n
    return res

def inversmod(a,p):
    """a**(-1) mod p"""
    #algorithme d'euclide etendu (solution de l'equation de Bezout)
    r1,u1,v1,r2,u2,v2=a,1,0,p,0,1
    while r2!=0:
        q=r1//r2
        r1,u1,v1,r2,u2,v2=r2,u2,v2,r1-q*r2,u1-q*u2,v1-q*v2
    if r1!=1:
        return 'pas inversible'
    else:
        if u1<1:
            while u1<1:
                u1+=p
        if u1>p:
            while u1>p:
                u1-=p
        return u1

#Fonctions de ElGamal
def chiffrement(p,g,cle_dest,message_clair,sign):
    k1=randint(1,p-1)
    cle_sess=puissmod(cle_dest,k1,p)
    message_crypt=(message_clair*cle_sess)%p
    entete=puissmod(g,k1,p)
    if sign:
        r,s=signature(p,g,k1,message_crypt)
        return r,s,entete,message_crypt
    else:
        return entete,message_crypt
```

```
46
47  def dechiffrement(p,g,cle_priv,entete,message_crypt):
48      cle_sess=puissmod(entete,cle_priv,p)
49      message_clair=(message_crypt*inversmod(cle_sess,p))%p
50      return message_clair
51
52  def signature(p,g,k1,message_crypt):
53      k2=randint(1,p-1)
54      k2inv=inversmod(k2,p-1)
55      while type(k2inv)==str: #tant que le k aleatoire n'est pas
        inversible mod p-1
56          k2=randint(1,p-1)
57          k2inv=inversmod(k2,p-1)
58      r=puissmod(g,k2,p)
59      h=message_crypt
60      s=((h-k1*r)*k2inv)%(p-1)
61      return r,s
62
63  def verification_signature(p,g,r,s,entete,message_crypt):
64      h=message_crypt
65      a=puissmod(g,h,p)
66      b=(puissmod(entete,r,p)*puissmod(r,s,p))%p
67      if a==b:
68          return True
69      else:
70          return False
71
72  def generation_cle(p,g):
73      """p premier,g generateur"""
74      cle_priv=randint(1,p-1)
75      cle_publ=puissmod(g,cle_priv,p)
76      return cle_priv,cle_publ
```

5.1 Génération des nombre premiers : déterministe et probabiliste (Miller-Rabin)

```
import random as rd
from time import perf_counter
import math

## Generation d'un nombre premier de maniere deterministe

def est_premier(n):
  if n == 2 or n == 3: return True
  if n < 2 or n%2 == 0: return False
  if n < 9: return True
  if n%3 == 0: return False
  r = int(n**0.5)
  f = 5
  while f <= r:
    if n%f == 0: return False
    if n%(f+2) == 0: return False
    f +=6
  return True

def premier(n): # Plus appliquable au dela de 20 chiffres
    while n>10:
        if est_premier(n):
            return n
        n-=1

def premier_Pollard(n):
    eps=10**(-100)
    while n>10:
        if (n-1)/2-int((n-1)/2)<eps:
            if est_premier(int((n-1)/2)):
                if est_premier(n):
                    return n
        n-=1

## Generation d'un nombre premier basee sur le test de primalite de
    Miller-Rabin

def MillerRabin_generation(b): #longueur en base 10 du nombre
   premier (proba pas premier 10**-300)
    i=0
    while True:
        i+=1
        n=rd.choice([1,3,5,7,9]) #candidat premier
        for k in range(1,b-1) :
            n+=rd.randint(0,9)*10**k
```

```
44          n+=rd.randint(1,9)*10**b
45          if MillerRabin_test(n,500):
46              return i,n
47
48  def MillerRabin_temoin(a,n):
49      #Calcul de s et d tels que n-1=2**s*d
50      d=(n-1)//2
51      s=1
52      while d%2==0:
53          d=d//2
54          s+=1
55      #Premier test
56      x=puissmod(a,d,n)
57      if x==1 or x==n-1 :
58          return False
59      #Boucle principale
60      while s>1:
61          x=x**2%n
62          if x==n-1:
63              return False
64          s-=1
65      return True
66
67  def MillerRabin_test(n,k): #primalite de n a tester et k nombre de
        boucles
68      for t in range(k):
69          a=rd.randint(2,n-2)
70          if MillerRabin_temoin(a,n):
71              return False
72      return True
73
74  ## Fabriquer un generateur g et un grand nombre premier p associe
75
76  def generateur_ElGamal(n):
77      while True:
78          q=MillerRabin_generation(n)[1]
79          for k in range(2,500):
80              p=k*q+1
81              if MillerRabin_test(p,100):
82                  while True:
83                      g=rd.randint(2,p) #Candidat generateur
84                      i=1
85                      grandordre=True
86                      while grandordre:
87                          if puissmod2(g,i,p)==1:
88                              grandordre=False
89                          elif i<k:
90                              i+=1
```

```
                            else:
                                return p,g

def generateur_RSA(n):
    while True:
        q=MillerRabin_generation(n)[1]
        for k in range(1,500):
            p=k*q+1
            if MillerRabin_test(p,100):
                return p
```

5.2 Client-serveur

5.2.1 Serveur

```
1  # Threads
2
3  from threading import Thread
4
5  class ThreadServeurClient(Thread):
6
7      def __init__(self,connexion_principale,connexion_client,
       L_connexions):
8          Thread.__init__(self)
9          self.connexion_principale=connexion_principale
10         self.connexion_client=connexion_client
11         self.L_connexions=L_connexions
12         self.pseudo=''
13
14     def run(self):
15         self.connexion_client.send('Entrez votre pseudo : '.encode
       ())
16         self.pseudo=self.connexion_client.recv(1024).decode()
17         print( 'Client '+str(self.connexion_client.getsockname())+'
       alias '+self.pseudo)
18         Continue=True
19         while Continue :
20             message_recu=self.connexion_client.recv(1024).decode()
21             if message_recu=='Deconnexion':
22                 Continue=False
23                 self.L_connexions.remove(self.connexion_client)
24             else:
25                 for client in self.L_connexions:
26                     if client != self.connexion_client:
27                         client.send((self.pseudo+' : '+message_recu
       ).encode())
28         self.connexion_client.send(b"Deconnexion")
29         print("Deconnecte avec le client {}".format(self.pseudo))
30         self.connexion_client.close()
31
32 # Serveur
33
34 import sys
35 import socket
36
37 hote='IP_DE_L_HOTE'
38 port = 12800
39
40 connexion_principale = socket.socket(socket.AF_INET, socket.
```

```
    SOCK_STREAM)
connexion_principale.bind((hote, port))
connexion_principale.listen(5)
connexion_principale.settimeout(20)
print("Le serveur ecoute a present sur le port {}".format(port))

L_connexions=[]
try:
    connexion_client, infos_connexion = connexion_principale.accept
    ()
except socket.timeout:
    print("Il n'y a pas de clients : Deconnection")
    connexion_principale.close()
    sys.exit()
print("Connecte avec le client {}".format(infos_connexion))
L_connexions.append(connexion_client)
ThreadServeurClient(connexion_principale,connexion_client,
    L_connexions).start()

while L_connexions != []:
    try:
        connexion_client, infos_connexion = connexion_principale.
    accept()
    except socket.timeout :
        continue
    print("Connecte avec le client {}".format(infos_connexion))
    L_connexions.append(connexion_client)
    ThreadServeurClient(connexion_principale,connexion_client,
    L_connexions).start()

print("Il n'y a plus de clients : Deconnection")
connexion_principale.close()

sys.exit()
```

5.2.2 Client

```
# Threads
from threading import Thread

class ThreadClientReception(Thread):

    def __init__(self,connexion_serveur):
        Thread.__init__(self)
        self.connexion_serveur=connexion_serveur

    def run(self):
```

```
11          Continue=True
12          while Continue:
13              message_recu=self.connexion_serveur.recv(1024).decode()
14              print(message_recu)
15              if message_recu=='Deconnexion':
16                  Continue=False
17          self.connexion_serveur.send(b'Deconnexion')
18
19  class ThreadClientEmission(Thread):
20
21      def __init__(self,connexion_serveur):
22          Thread.__init__(self)
23          self.connexion_serveur=connexion_serveur
24
25      def run(self):
26
27          while True:
28              message=input()
29              try:
30                  self.connexion_serveur.send(message.encode())
31              except:
32                  break
33
34  # Client
35
36  import sys
37  import socket
38
39  hote = "localhost" #Remplacer par l'adresse IP
40  port = 12800
41
42  connexion_serveur = socket.socket(socket.AF_INET, socket.
        SOCK_STREAM)
43  connexion_serveur.connect((hote, port))
44  print("Connexion etablie avec le serveur sur le port {}".format(
        port))
45
46  thread_reception=ThreadClientReception(connexion_serveur)
47  thread_emission=ThreadClientEmission(connexion_serveur)
48
49  Continue=True
50  thread_reception.start()
51  thread_emission.start()
52
53  thread_reception.join()
54  sys.exit()
```

5.3 Opérations élémentaires

```
1  def puissmod(a,d,n):
2      """a**d mod n"""
3      #iteratif : beaucoup plus efficace (10^-5) 600 chiffres ->
       0.025 s
4      dbin=bin(d)
5      L=[int(dbin[-i-1]) for i in range(len(dbin)-2)]
6      res=1
7      while L!=[]:
8          k=L.pop(0)
9          if k>0:
10             res=res*a%n
11         a=a**2%n
12     return res
13
14 def inversmod(a,p):
15     """a**(-1) mod p"""
16     #algorithme d'Euclide etendu (solution de l'equation de Bezout)
17     r1,u1,v1,r2,u2,v2=a,1,0,p,0,1
18
19     while r2!=0:
20         q=r1//r2
21         r1,u1,v1,r2,u2,v2=r2,u2,v2,r1-q*r2,u1-q*u2,v1-q*v2
22     if r1!=1:
23         return "pas inversible"
24     else:
25         if u1<1:
26             while u1<1:
27                 u1+=p
28         if u1>p:
29             while u1>p:
30                 u1-=p
31         return u1
```

5.4 Data Encryption Standard (DES)

```
# ************* ENCODAGE DES CARACTERES *************

#Codage du message: 1 caractere = 8 bits
def encodage(m):
    """Message en STRING retourne en binaire 8 bits (table AISCII)
    """
    if type(m) != str :
        raise TypeError
    M = list(m) # on split tout en caracteres individuels
    Mord = [ord(M[i]) for i in range(len(M))]
    Mbin = [bin(Mord[i]) for i in range(len(M))]
    # traitement du Mbin pour enlever le '0b' du debut et taille
    constante
    Mfin=[]
    for binaire in Mbin:
        binaire=binaire[2:]
        n=len(binaire)
        if n>8:
            return 'Erreur len(binaire)'
        for i in range(8-n): # complete avec des zeros
            Mfin.append(0)
        for i in binaire:
            Mfin.append(int(i))
    return Mfin

def decodage(M):
    n=len(M)//8
    Mfin=[str(i) for i in M]
    Mbin=[]
    for i in range(n):
        binaire=Mfin[i*8]
        for j in range(1,8):
            binaire+=Mfin[i*8+j]
        Mbin.append(binaire)
    Mord=[int(binaire,2) for binaire in Mbin]
    Mchr=[chr(i) for i in Mord]
    m=Mchr[0]
    for i in range(1,len(Mchr)):
        m+=Mchr[i]
    return m


# ************* GENERATION DES CLES *************

def permutationCP1(K):
    """Effectue la permutation initiale de la cle de 64 bits,
```

```
    retourne les listes G0 et D0"""
45      if len(K) != 64 : return "Erreur len(K)"
46      else:
47          K1 = []
48          CP1=[57,49,41,33,25,17, 9, 1,58,50,42,34,26,18,
49               10, 2,59,51,43,35,27,19,11, 3,60,52,44,36,
50               63,55,47,39,31,23,15, 7,62,54,46,38,30,22,
51               14, 6,61,53,45,37,29,21,13, 5,28,20,12, 4]
52          for i in CP1:
53              K1.append(K[i-1])
54          return K1[:28], K1[28:]
55
56  def decalageGauche(L, n):
57      """ Decale les elements de la liste L de n places vers la
    GAUCHE"""
58      Ldec = []
59      for i in range(len(L)-n):
60          Ldec.append(L[i+n])
61      for i in range(n):
62          Ldec.append(L[i])
63      return Ldec
64
65  def regroupe(G,D):
66      return G+D
67
68  def permutationCP2(K):
69      """Effectue la permutation CP2 du bloc de 56 bits en un bloc de
    48 bits qui est la cle Ki"""
70      if len(K) != 56 : return "Erreur len(K)"
71      else:
72          Ki = []
73          CP2=[14,17,11,24, 1, 5, 3,28,15, 6,21,10,
74               23,19,12, 4,26, 8,16, 7,27,20,13, 2,
75               41,52,31,37,47,55,30,40,51,45,33,48,
76               44,49,39,56,34,53,46,42,50,36,29,32]
77          for i in CP2:
78              Ki.append(K[i-1])
79          return Ki
80
81  def genKey(K):
82      """ Retourne les 16 cles Ki a partir de la cle de 64 bits"""
83      decalage = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1] #
    Nombre de decalage a gauche pour les 16 iterations (de 0 a 15)
84      cles = []
85      #verification cle est de 64 bits:
86      if checkBINn(K, 64) == False: return "Erreur sur la cle
    initilale K"
87      # permutation initiale CP1:
```

```
88      G, D = permutationCP1(K)
89      if checkBINn(G, 28) == False: return "Erreur sur la cle
        initiale G"
90      if checkBINn(D, 28) == False: return "Erreur sur la cle
        initiale D"
91      for i in range(16):
92          Gi = decalageGauche(G, decalage[i])
93          Di = decalageGauche(D, decalage[i])
94          Ki = permutationCP2(regroupe(Gi, Di))
95          if checkBINn(Ki, 48) == False: return "Erreur sur la cle Ki
        "
96          cles.append(Ki)
97          G = Gi
98          D = Di
99      return cles
100
101 def checkBINn(L, n):
102     """ verifie que la liste L est une liste contenant n elements
        binaire (0 ou 1)"""
103     binaire = [0,1]
104     if len(L)==n:
105         for i in range(len(L)):
106             if L[i] not in binaire:
107                 return False
108         return True
109     else:
110         return False
111
112 def gen(n):
113     return ([(i+1)*10 for i in range(n)])
114
115
116 # ************* PERMUTATIONS *************
117
118 def permutation_initiale(L):
119     """Permute les elements de L selon l'ordre de permutation
        defini par PI"""
120     PI=[58,50,42,34,26,18,10, 2,
121         60,52,44,36,28,20,12, 4,
122         62,54,46,38,30,22,14, 6,
123         64,56,48,40,32,24,16, 8,
124         57,49,41,33,25,17, 9, 1,
125         59,51,43,35,27,19,11, 3,
126         61,53,45,37,29,21,13, 5,
127         63,55,47,39,31,23,15, 7] #Liste de permutation initiale
128     Q=[L[i-1] for i in PI]
129     return Q
130
```

```
131 def scindement2(L):
132     """Retourne le scindement de L en 2 listes G0 et D0 de de 32
    bits"""
133     return L[0:32], L[32:64]
134
135 def permutation_inverse(L):
136     """Permute les elements de L selon l'ordre de permutation
    defini par PII = permutation initiale inverse"""
137     PII=[40,8,48,16,56,24,64,32,
138          39,7,47,15,55,23,63,31,
139          38,6,46,14,54,22,62,30,
140          37,5,45,13,53,21,61,29,
141          36,4,44,12,52,20,60,28,
142          35,3,43,11,51,19,59,27,
143          34,2,42,10,50,18,58,26,
144          33,1,41, 9,49,17,57,25]
145     Q=[L[i-1] for i in PII]
146     return Q
147
148 # ************* RONDE *************
149
150 def expansion(D):
151     """Etend les 32 bits du bloc D0 en 48 bits dans Q"""
152     E=[32, 1, 2, 3, 4, 5,
153         4, 5, 6, 7, 8, 9,
154         8, 9,10,11,12,13,
155        12,13,14,15,16,17,
156        16,17,18,19,20,21,
157        20,21,22,23,24,25,
158        24,25,26,27,28,29,
159        28,29,30,31,32, 1]
160     Q = [D[i-1] for i in E]
161     return Q
162
163 def XOR(b1,b2):
164     """OU exclusif entre 2 bits b1 et b2"""
165     if b1==1:
166         if b2==1:
167             return 0
168         else:
169             return 1
170     elif b2==1:
171         return 1
172     else:
173         return 0
174
175 def XORL(Dprime,K):
176     """Ou exclusif entre Dprime et cle K (48 bits)"""
```

```
177         D0 = []
178         for i in range(len(Dprime)):
179             D0.append(XOR(Dprime[i], K[i]))
180         return D0
181 
182     def scindement8(D0):
183         """Scinde le bloc de 48 en 8 blocs de 6"""
184         return [D0[0:6], D0[6:12], D0[12:18], D0[18:24], D0[24:30], D0
        [30:36], D0[36:42], D0[42:48]]
185 
186     def selection1(D01):
187         """Selection pour D01"""
188         S1=[[14, 4,13, 1, 2,15,11, 8, 3,10, 6,12, 5, 9, 0, 7],
189             [ 0,15, 7, 4,14, 2,13, 1,10, 6,12,11, 9, 5, 3, 8],
190             [ 4, 1,14, 8,13, 6, 2,11,15,12, 9, 7, 3,10, 5, 0],
191             [15,12, 8, 2, 4, 9, 1, 7, 5,11, 3,14,10, 0, 6,13]]
192         ligne = int(str(D01[0]*10 + D01[5]), 2)
193         colonne = int(str(D01[1]*1000 + D01[2]*100 + D01[3]*10 + D01
        [4]), 2)
194         return S1[ligne][colonne]
195 
196     def selection2(D02):
197         """Selection pour D02"""
198         S2=[[15, 1, 8,14, 6,11, 3, 4, 9, 7, 2,13,12, 0, 5,10],
199             [ 3,13, 4, 7,15, 2, 8,14,12, 0, 1,10, 6, 9,11, 5],
200             [ 0,14, 7,11,10, 4, 3, 1, 5, 8,12, 6, 9, 3, 2,15],
201             [13, 8,10, 1, 3,15, 4, 2,11, 6, 7,12, 0, 5,14, 9]]
202         ligne = int(str(D02[0]*10 + D02[5]), 2)
203         colonne = int(str(D02[1]*1000 + D02[2]*100 + D02[3]*10 + D02
        [4]), 2)
204         return S2[ligne][colonne]
205 
206     def selection3(D03):
207         """Selection pour D03"""
208         S3=[[10, 0, 9,14, 6, 3,15, 5, 1,13,12, 7,11, 4, 2, 8],
209             [13, 7, 0, 9, 3, 4, 6,10, 2, 8, 5,14,12,11,15, 1],
210             [13, 6, 4, 9, 8,15, 3, 0,11, 1, 2,12, 5,10,14, 7],
211             [ 1,10,13, 0, 6, 9, 8, 7, 4,15,14, 3,11, 5, 2,12]]
212         ligne = int(str(D03[0]*10 + D03[5]), 2)
213         colonne = int(str(D03[1]*1000 + D03[2]*100 + D03[3]*10 + D03
        [4]), 2)
214         return S3[ligne][colonne]
215 
216     def selection4(D04):
217         """Selection pour D04"""
218         S4=[[ 7,13,14, 3, 0, 6, 9,10, 1, 2, 8, 5,11,12, 4,15],
219             [13, 8,11, 5, 6,15, 0, 3, 4, 7, 2,12, 1,10,14, 9],
220             [10, 6, 9, 0,12,11, 7,13,15, 1, 3,14, 5, 2, 8, 4],
```

```
221             [ 3,15, 0, 6,10, 1,13, 8, 9, 4, 5,11,12, 7, 2,14]]
222         ligne = int(str(D04[0]*10 + D04[5]), 2)
223         colonne = int(str(D04[1]*1000 + D04[2]*100 + D04[3]*10 + D04
        [4]), 2)
224         return S4[ligne][colonne]
225
226     def selection5(D05):
227         """Selection pour D05"""
228         S5=[[ 2,12, 4, 1, 7,10,11, 6, 8, 5, 3,15,13, 0,14, 9],
229             [14,11, 2,12, 4, 7,13, 1, 5, 0,15,10, 3, 9, 8, 6],
230             [ 4, 2, 1,11,10,13, 7, 8,15, 9,12, 5, 6, 3, 0,14],
231             [11, 8,12, 7, 1,14, 2,13, 6,15, 0, 9,10, 4, 5, 3]]
232         ligne = int(str(D05[0]*10 + D05[5]), 2)
233         colonne = int(str(D05[1]*1000 + D05[2]*100 + D05[3]*10 + D05
        [4]), 2)
234         return S5[ligne][colonne]
235
236     def selection6(D06):
237         """Selection pour D06"""
238         S6=[[12, 1,10,15, 9, 2, 6, 8, 0,13, 3, 4,14, 7, 5,11],
239             [10,15, 4, 2, 7,12, 9, 5, 6, 1,13,14, 0,11, 3, 8],
240             [ 9,14,15, 5, 2, 8,12, 3, 7, 0, 4,10, 1,13,11, 6],
241             [ 4, 3, 2,12, 9, 5,15,10,11,14, 1, 7, 6, 0, 8,13]]
242         ligne = int(str(D06[0]*10 + D06[5]), 2)
243         colonne = int(str(D06[1]*1000 + D06[2]*100 + D06[3]*10 + D06
        [4]), 2)
244         return S6[ligne][colonne]
245
246     def selection7(D07):
247         """Selection pour D07"""
248         S7=[[ 4, 1, 2,14,15, 0, 8,13, 3,12, 9, 7, 5,10, 6, 1],
249             [13, 0,11, 7, 4, 9, 1,10,14, 3, 5,12, 2,15, 8, 6],
250             [ 1, 4,11,13,12, 3, 7,14,10,15, 6, 8, 0, 5, 9, 2],
251             [ 6,11,13, 8, 1, 4,10, 7, 9, 5, 0,15,14, 2, 3,12]]
252         ligne = int(str(D07[0]*10 + D07[5]), 2)
253         colonne = int(str(D07[1]*1000 + D07[2]*100 + D07[3]*10 + D07
        [4]), 2)
254         return S7[ligne][colonne]
255
256     def selection8(D08):
257         """Selection pour D08"""
258         S8=[[13, 2, 8, 4, 6,15,11, 1,10, 9, 3,14, 5, 0,12, 7],
259             [ 1, 5,13, 8,10, 3, 7, 4,12, 5, 6,11, 0,14, 9, 2],
260             [ 7,11, 4, 1, 9,12,14, 2, 0, 6,10,13,15, 3, 5, 8],
261             [ 2, 1,14, 7, 4,10, 8,13,15,12, 9, 0, 3, 5, 6,11]]
262         ligne = int(str(D08[0]*10 + D08[5]), 2)
263         colonne = int(str(D08[1]*1000 + D08[2]*100 + D08[3]*10 + D08
        [4]), 2)
```

```
264         return S8[ligne][colonne]
265
266     def quatre(chaine):
267         """ transforme un nombre (en str) en liste contenant chaque
            chiffre en int ET normalise en liste de 4 elements avec des 0
            devants"""
268         L = []
269         for i in chaine:
270             L.append(int(i))
271         n = len(L)
272         if n > 4:
273             return 'Erreur longueur liste'
274         else:
275             for i in range(4-n):
276                 L.insert(0, int(0))
277         return L
278
279     def somme(D01, D02, D03, D04, D05, D06, D07, D08):
280         """Rassemble les valeurs obtenues, cree la table de 32 bits"""
281         Valeurs = [selection1(D01), selection2(D02), selection3(D03),
            selection4(D04), selection5(D05), selection6(D06), selection7(
            D07), selection8(D08)]
282         ValeursBIN = [bin(Valeurs[i])[2:] for i in range(8)]
283         ValeursBINlist =[]
284         for i in range(8):
285             ValeursBINlist.append(quatre(ValeursBIN[i]))
286         table32 = []
287         for i in range(8):
288             table32 = table32 + ValeursBINlist[i]
289         return table32
290
291     def selection(D0):
292         """Etape de selection d'une liste D0 de 48 bits en une liste de
            32 bits"""
293         #verification
294         if checkBINn(D0, 48) == False : return "Erreur L0"
295         #scindement :
296         D0i = scindement8(D0)
297         return somme(D0i[0], D0i[1], D0i[2], D0i[3], D0i[4], D0i[5],
            D0i[6], D0i[7])
298
299     def permutation32(L):
300         P = [16, 7,20,21,29,12,28,17,
301               1,15,23,26, 5,18,31,10,
302               2, 8,24,14,32,27, 3, 9,
303              19,13,30, 6,22,11, 4,25,]
304         Q = [L[i-1] for i in P]
305         return Q
```

```
306
307
308 # ************* ENCODAGE CLE *************
309
310 def intbin(K):
311     """ Retourne l'entier positif K en binaire dans une liste L"""
312     if type(K) != int: return "Erreur type K"
313     if K < 0: return "Erreur 'K est negatif'"
314     kbinstr = list(bin(K))
315     kbinstr = kbinstr[2:]
316     # kbin = [int(kbinstr[i]) for i in range(len(kbinstr))]
317     kbin = [int(i) for i in kbinstr]
318     return kbin
319
320 def binint(L):
321     """ Retourne la liste binaire L en entier positif """
322     if checkBINn(L, len(L)) == False : return "Erreur liste L"
323     N =''
324     for i in range(len(L)):
325         N+=str(L[i])
326     return int(N,2)
327
328 def min64(L):
329     """ Verifie et normalise avec des 0 devants la liste L tel que
    contienne au moins 64 bits """
330     if len(L)>63 :
331         return L
332     else:
333         n = len(L)
334         Q = L[:]
335         Q.reverse()
336         for i in range(64-n):
337             Q.append(int(0))
338         Q.reverse()
339         return Q
340
341
342 # ************* FONCTIONS DE DECOUPAGE *************
343
344 espace = [0, 0, 1, 0, 0, 0, 0, 0] #(ord 32) Caractere [espace] en
    binaire 8 bits AISCII
345
346 def decoupe64(l):
347     """ decoupe une chaine binaire en bloc de 64 bits et completant
    les vides par des 'espaces' """
348     B = [] #Liste BLOCs, resultat
349     L = l[:] #copie L
350     n  = len(L)
```

```
351     r = n % 64 # reste de la division euclidienne de n par 64
352     nesp = (64-r)//8 #nombre de caractee 'espace' a  ajouter pour
        complater L
353     if r != 0 : #si len(L) n'est pas un multiple de 64, on complate
        L avec des 'espaces'
354         for i in range(nesp): #on ajoute le caractare 'espace' nesp
        fois
355             L+=espace
356     q = len(L) // 64 # quotient : [nombre elements de L]/64 =
        nombre de sous liste a creer
357     for i in range(q):
358         SB =[] #Sous bloc (temp)
359         for j in range(64):
360             SB.append(L.pop(0)) #on pop et on ajoute en meme temps
361         B.append(SB)
362     return B
363
364 def assembler(B):
365     """regroupe liste contenant des sous-listes (bloc de 64) en une
        liste"""
366     L = [] #resultat
367     for i in range(len(B)):
368         L+=B[i]
369     return L
370
371
372 # ************* FONCTION DE VERIFICATION *************
373
374 def checkBINn(L, n):
375     """ verifie que la liste L est une liste contenant n elements
        binaire (0 ou 1)"""
376     binaire = [0,1]
377     if len(L)==n:
378         for i in range(len(L)):
379             if L[i] not in binaire:
380                 return False
381         return True
382     else:
383         return False
384
385
386 # ************* MAIN *************
387
388 def DESc(M, K):
389     """ Crypte le message M (binaire de 64 bits) avec la cle K (
        binaire de 64 bits) par la methode du DES"""
390     #Verification
391     if checkBINn(M, 64) == False: return 'Erreur M'
```

```
392         if checkBINn(K, 64) == False: return 'Erreur K'
393         #Calcul des cles
394         key = genKey(K)
395         MPI = permutation_initiale(M)
396         #scindement
397         G,D =  scindement2(MPI)
398         #initialisation de la ronde
399         Gi = G
400         Di = D
401         #ronde
402         for i in range(16):
403             Dexpand= expansion(Di)
404             Xi = XORL(Dexpand, key[i])
405             Si = selection(Xi)
406             Ti = permutation32(Si)
407             Gi, Di = Di,  XORL(Gi, Ti)
408         #regroupement INVERSE
409         R = Di + Gi
410         #permutation inverse
411         return permutation_inverse(R)
412
413
414 def DESd(M, K):
415         """ Decrypte le message M (binaire de 64 bits) avec la cle K (
        binaire de 64 bits) par la methode du DES"""
416         #Verification
417         if checkBINn(M, 64) == False: return 'Erreur M'
418         if checkBINn(K, 64) == False: return 'Erreur K'
419         #Calcul des cles
420         key = genKey(K)
421         key.reverse()
422         MPI = permutation_initiale(M)
423         #scindement
424         G,D =  scindement2(MPI)
425         #initialisation de la ronde
426         Gi = G
427         Di = D
428         #ronde
429         for i in range(16):
430             Dexpand= expansion(Di)
431             Xi = XORL(Dexpand, key[i])
432             Si = selection(Xi)
433             Ti = permutation32(Si)
434             Gi, Di = Di,  XORL(Gi, Ti)
435         #regroupement INVERSE
436         R = Di + Gi
437         #permutation inverse
438         return permutation_inverse(R)
```

```
439
440 # ------------------------------------------------
441
442 def aide():
443     print("-----------------------------------------")
444     print("************* FONCTIONEMENT *************")
445     print("-----------------------------------------")
446     print()
447     print("FONCTION CRYPTAGE DES(m, K)")
448     print("_____________________")
449     print("--> permet de crypter un message m avec K")
450     print("1. m est un message en clair - str")
451     print("2. K est la cle de cryptage - entier naturel")
452     print()
453     print()
454     print("FONCTION DECRYPTAGE DES_(M, K)")
455     print("_____________________")
456     print("--> permet de decrypter un message M cryte avec K")
457     print("1. M est un message crypte avec K - liste de contenant
    des 0 et 1")
458     print("2. K est la cle de decryptage - entier naturel")
459     print()
460     print("-----------------------------------------")
461     print("****************** FIN ******************")
462     print("-----------------------------------------")
463
464 # ------------------------------------------------
465
466 def DES(m, K):
467     """ Crypte le message m (str) avec la cle K (entier), retourne
    liste binaire correpondant a m crypter en AISCII """
468     #Verification:
469     if type(m) != str: return "Erreur type m"
470     if type(K) != int: return "Erreur type K"
471     Kentiere = intbin(K)
472     Kentiere = min64(Kentiere) #pour avoir une cle binaire de
    taille 64 bits minimum
473     Kentiere.reverse()
474     key = Kentiere[:64]
475     M = decoupe64(encodage(m))
476     n = len(M) # nombre de blocs de 64 bits
477     Mc = [] # M crypte
478     for i in range(n):
479         Mc.append(DESc(M[i], key))
480     return assembler(Mc)
481
482 def DES_(M, K):
483     """ Crypte le message m (str) avec la cle K (entier), retourne
```

```
    liste binaire correpondant a m crypter en AISCII """
484      #Verification:
485      if checkBINn(M, len(M)) == False : return "Erreur M"
486      if type(K) != int: return "Erreur type K"
487      Kentiere = intbin(K)
488      Kentiere = min64(Kentiere) #pour avoir une cle binaire de
    taille 64 bits minimum
489      Kentiere.reverse()
490      key = Kentiere[:64]
491      print("Cle =", K,"-->", "key =",key )
492      print()
493      Mc = decoupe64(M)
494      Mclair = [] # M decrypte
495      for i in range(n):
496          Mclair.append(DESd(Mc[i], key))
497      return decodage(assembler(Mclair))
498
499  print("Entrer 'aide()' pour le fonctionnement")
```

5.5 Attaques El Gamal : algorithme de Shanks et calcul de l'indice

```
from math import exp, sqrt, log

#Logarithme discret naif

def lognaif(p,g,x):
    for i in range(p):
        if puissmod(g,i,p)==x:
            return i

#Pas de bebe,pas de geant (SHANKS)

def shanks(p,g,x): #retourne n tel que x=g**n mod p
    K=int(sqrt(p))+1
    c=inversmod(puissmod(g,K,p),p)
    A=[(puissmod(g,i,p),i) for i in range(1,K)]
    A.sort(key=lambda x:x[0])
    b=x
    for j in range(K):
        if b<int(p/2): #On parcours A dans l'ordre croissant
            i=0
            while i<K-2 and b>A[i][0]:
                i+=1
            if A[i][0]==b:
                return A[i][1]+K*j
            else:
                b=(b*c)%p
        else: #On parcours A dans l'ordre decroissant
            i=K-2
            while i>0 and b<A[i][0]:
                i-=1
            if A[i][0]==b:
               return A[i][1]+K*j
            else:
                b=(b*c)%p

#Calcul de l'indice (Complexe)

def Modele_CalculIndice(i,j,pas):
    for x in range(i,j,pas):
        print(x,exp(sqrt(2*log(10**x))*sqrt(log(log(10**x)))),sep=';')
```

5.6 RSA

```
# Donnees :
# Privees : p, q, d premier avec (p-1)*(q-1)=phi(p*q)
# Publiques : n=p*q, e tel que e*d=1 mod phi(n)

def pgcd(a,b): #algorithme d'Euclide
    while a%b!=0:
        a,b=b,a%b
    return b

def generateur(n):
    while True:
        q=MillerRabin_generation(n)
        for k in range(1,500):
            p=k*q+1
            if MillerRabin_test(p,100):
                return p

def RSA_generation(n):
    p=generateur(n)
    q=generateur(n) #pas B friable
    phi=(p-1)*(q-1)
    e=3
    while pgcd(e,phi)!=1:
        e+=2
    d=inversmod(e,phi)
    n=p*q
    return n,e,d

def RSA_chiffrement(m,e,n):
    c=puissmod(m,e,n)
    return c

def RSA_dechiffrement(c,d,n):
    m=puissmod(c,d,n)
    return m

def RSA_signature(h,d,n):
    s=puissmod(h,d,n)
    return s

def RSA_verification(s,h,e,n):
    if puissmod(s,e,n)==h:
        return True
    else:
        return False
```

5.7 Attaques RSA : Crible algébrique et algorithme rho de Pollard

```
from math import exp, sqrt, log
from fractions import gcd

# Attaque p-1 de Pollard

def Pollard_p_1(n, B):
    a=2
    for i in range(2,B+1):
        a=puissmod(a,i,n)
    d=gcd(a-1, n)
    if d>1 and d<n:
        return d
    return None

def Pollard_rho(n, x1=1, f=lambda x: x**2+1):
    x=x1
    y=f(x)%n
    p=gcd(y-x,n)
    while p==1:
        x=f(x)%n
        y=f(f(y))%n
        p=gcd(y-x,n)
    if p==n:
        return None
    return p

#Crible algebrique : Complexe

def Modele_CribleAlgebrique(i,j,pas):
    for x in range(i,j,pas):
        print(x,exp(((64/9)*log(10**x))**(1/3)*(log(log(10**x)))
    **(2/3)),sep=';')
```