Elgamal

RAYAN

November 6, 2022

Contents

1 Elgamal

1.1 Origine

Elgamal est un protocole ou chiffrement de cryptographie asymétrique crée par Taher ElGamal en 1981 et a été inventé pour résoudre un certain type de problème (logarithme discret).

Et tout comme RSA, il présente un protocole de chiffrement, déchiffrement, signature et de vérification, mais les deux ne sont pas pareil.

1.2 Expliquation code

1.2.1 Elgamal

1. Génération de clef

   (a) Describtion

   Le but de créer une clef est qu'on envoie une partie de la clef dite **publique** à tout le monde, mais seulement nous avons accès à l'autre partie de la clef dite **privé**.

   La clef publique permettra aux autre personne de pouvoir en chiffré des message que seule notre partie privé pourra déchiffrer.

   Ce qui nous assure une protection lors d'envoyer de message. On va générer la paire de clefs qui permettront de chiffrer, déchiffrer, signer, et vérifier nos messages et ceux des autres.

   "a" sera par la suite pour nous notre clef privée et "p, alpha, h", la clef publique qu'on transmettra à tout le monde.

   (b) Argument

   - k = la taille de la clef

   (c) Variables

   - q = est un chiffre premier choisi aléatoirement entre 1 et k -1.
   - p = est un chiffre premier avec q et sera par la suite la taille de notre groupe Zp.
   - alpha = est choisi aléatoirement dans (le groupe Zp d'ordre n)

   et est testé si le résultat entre (alpha à la puissance 2 mod de p) et (alpha à la puissance q mod de p) Et bien différent de 1. Alors on choisi ce p. On appeler aussi alpha le générateur du Zp.

   - a est un chiffre choisi aléatoirement entre 1 et p-2.
   - h est le résultat d'alpha à la puissance de a modulo p

   cf :

   - Un Groupe Zp d'ordre n.

   En mathématiques, et plus précisément en théorie des groupes, un groupe cyclique est un groupe qui est à la fois fini et monogène et d'ordre n, car n'est pas un élément du groupe tel que tous les éléments de ce groupe puissent être retrouvés sous la forme d'un multiple de n, on peut l'appeler aussi générateur.

- Presque tous les calculs se feront modulo p pour éviter un trop grand temps de calcul.
- Le chiffrement asymétrique n'est pas fait pour l'envoi de message, mais plutôt l'échange de clef symétrique. Ici, j'expliquerai chaque étape comme si on envoyait des messages pour une meilleure compréhension de la chose.
- La taille de clef dite sure actuellement est "1024" bits.

(d) Code

La fonction gen est une fonction qui génère la clé publique (p, alpha, h) de longueur k, et on fait des tests comme expliqué précédemment pour voir si alpha est un générateur de notre groupe Zp avec un p qui doit être premier avec q.

```
def gen(k):
    p=4
    while isPrime(p) !=1:
q = getPrime(k-1)  #Select a random (k  1)-bit prime q
p = 2*q+1 # Compute p←2q + 1, and test whether p is prime
    b= 1
    n=2*q
    while b==1: #If b = 1 then go to step 1
alpha =  random.randint(1,p-1)
if (pow(alpha,n//2,p) !=1 and pow(alpha,n//q,p) != 1):
    b = 0

    return p,alpha
```

On assemble cette fonction avec genkey pour nous donner le resultat final sous la forme d'un tuple (public, privé).
"a" est ici comme expliqué auparavant généré aléatoirement entre 1 et p-2, car il doit être un élément de notre groupe Zp.

```
def genkey(s):
    p,alpha = gen(s)
    a = random.randint(1,p-2)
    h = pow(alpha,a,p)
    return (a,(p,alpha,h))
```

(e) Exemple

```
print(genkey(10)) #taille de la clef 10
```

Resulte une clef:

```
(839, (863, 557, 563))
```

Avec comme clef privé a = 839, choisie donc entre 1 et p(863). Avec comme clef pulique p = 863, alpha = 557 et h = 563.

2. Chiffrement

   (a) Describtion

   Après avoir généré nos clefs Alice voudrait envoyer un message à Bob, mais ne veut pas que son ennemie Smanal ne le découvre.

   Une solution s'offre à elle, utiliser la clé publique de Bob pour chiffrer son message que **seul** Bob pourra déchiffrer.

   Le but est donc de chiffré pour que personne ne puisse lire les messages avec la clef publique d'une personne, hors la personne avec la clef privé.

   (b) Argument

   - m = message de la personne
   - pk = La clé publique avec laquelle on va chiffrer le message

   (c) Variable

   - k = chiffre choisi entre 1 et p-2
   - c1 = une partie du message chiffré, qui est alpha à la puissance k modulo p.
   - c2 = l'autre partie du message chiffré, qui est (h*[2] à la puissance k modulo p) multiplié par le message en int*[1], le tout modulo p

   cf : *[1] Ici et par la suite, je ne fais pas de conversion de String vers int pour le message, car je n'en ai pas vraiment ici besoin, mais j'aurais pu le rajouter si nécessaire. *[2] h est le resultat de alpha à la puissance de a modulo p

   (d) Code

   La fonction "encrypt" prend donc un message m et la clé publique d'une personne. Puis renvoie : c1 et c2 comme un tuple du message chiffré

```
def encrypt(m, pk):
    k = random.randint(1,pk[0]-2)
    c1 = pow(pk[1],k,pk[0])
    z = pow(pk[2],k,pk[0])
    c2 = (m*z) % pk[0]
    return (c1, c2)
```

(e) Exemple Ici la taille est comme avant de k = 10, et on chiffre le message "10", avec la clef publique key[1] ->(p,alpha,h).

```
key = genkey(10)
print(encrypt(10,key[1]))
```

Le résultat du chiffrement de 10 est donc le tuple :

```
(974, 125)
```

3. Déchiffrement

(a) Expliquation

Après avoir chiffré le message, Bob aimerait bien donc lire ce message d'Alice. Il va donc user de sa clé privé que seul lui détient pour déchiffrer le message et avoir le contenu de ce dernier.

(b) Argument

- c = le message chiffré en tuple -> (c1,c2)
- key = la clef contenant : la clef publique et la privée sous la forme d'un tuple.

(c) Variable

s = est une variable temporaire juste pour bien séparer les calcules, pour rendre le tous plus claire.

(d) Code

La fonction prend, donc le chiffré et la clé de la personne. Puis, on prend c1 à la puissance a (la clé privée) le tout modulo p. Enfin, on multiplie c2 avec l'inverse modulaire de s (le résultat d'avant) modulo p. Ce qui nous renvoie bien le message de basse.

```
def decrypt(c,key):
  s =  pow(c[0],key[0],key[1][0])
  return  c[1] * inverse(s, key[1][0]) % key[1][0]
```

(e) Exemple

On chiffre le message de 10 et on essaye d'afficher le résultat du déchiffrement.

```
key = genkey(10)
e = encrypt(10,key[1])
print(decrypt(e,key))
```

Résultat, on a réussi, ça nous renvoie bien le message 10.

```
10
```

4. Signature

(a) Expliquation On veut maintenant veut pouvoir signer le message pour bien **prouver** que soit nous qui envoyons le message pour pouvoir faire du chiffrement authentifié.

(b) Argument

- keys = la clef privé
- keyp = la clef publique
- m = le message binaire

(c) Variable

- k = un nombre choisi aléatoirement entre 2 et p-1 et qui gcd(k,p-1) == 1.
- mh = le message hashé.
- r = est alpha à la puissance k modulo de p.
- inv,t1,t2 = variables temporaires
- s = k1{h(m) ar}

(d) Code

La fonction prend, donc la clef publique, privé et le message. Pour le tous, nous renvoyer le tuple avec le message hashé et de l'autre r et s.

```
def signature(keys,keyp,m):
    k = 0
    while gcd(k,keyp[0]-1) != 1:
k = random.randint(2, keyp[0]-1)
    mh = h(m)
    r = pow(keyp[1],k,keyp[0])
    inv = inverse(k,keyp[0]-1)
```

```
    t1 = (keys * r) %(keyp[0]-1)
    t2 = (mh - t1) %(keyp[0]-1)
    s = (inv * t2) %( keyp[0]-1)
    return (mh,(r,s))
```

(e) Exemple

Ici, je vais juste prendre un exemple simple pour présenter la chose qui montre bien la signature.

```
Alice = genkey(10)
s = signature(Alice[0],Alice[1],1001)
print(s)
```

Resultat:

```
(1306133012808677871027210991478931373809016626146826806641444115040600202838
```

mh= 1306133012808677871027210991478931373809016626146826806641444115040600202838
r = 34
s = 538

5. Vérification

(a) Expliquation Maintenant, on veut vérifier la signature

(b) Argument

s = le tuple de la signature
keyp = la clef publique

(c) Variable

v1 = $h^{r}*r^{s}$ mod p.
v2 = $alpha^{h}$(m) mod de p.

(d) Code On prend, donc la signature et la clef, et on fait des tests pour voir si le résultat de v1 et v2 corresponde, si oui, c'est **insérer un nom de la personne**, qui a envoyé le message.

```
def verify(s,keyp):
    if (1 > s[0]  and s[0] > keyp[0]-1):
return -1
    tmp1 =pow(keyp[2],s[1][0],keyp[0]) # h^r%p ou h = y dans le chapitre 11
    tmp2 = pow(s[1][0],s[1][1],keyp[0]) # r^s%p
    v1 = (tmp1*tmp2) % keyp[0]
    v2 = pow(keyp[1],s[0], keyp[0]) #alpah ^h(m) % p
```

```
    if v1 == v2:
return 1
    else:
return -1
```

(e) Exemple

Alice crée sa clef et signe un message, mais Smanal veut se faire passer pour Alice, en envoyant aussi une signature à Bob. Bob, lui vérifie les deux signatures avec la clef publique d'Alice et regarde lequel des deux est la bonne.

```
Alice = genkey(10) # Alice signe un message
s = signature(Alice[0],Alice[1],1001)

Smanal = genkey(10) # Smanal essaye de se prendre pour alice
s2 = signature(Smanal[0],Smanal[1],1001)

Bobverif = verify(s,Alice[1])
Bobverif2 = verify(s2,Alice[1])

print(Bobverif)
print(Bobverif2)
```

Résulta: La première signature vient bien de Alice alors que le second non.

```
1
-1
```

1.2.2 Multiplicative Homomorphic E-Voting

1. Explication La Multiplicative Homomorphic est le fait de garder la structure entre deux chiffrements et d'applique une multiplication entre les deux chiffrées et ainsi obtenir chiffrer avec le résultat des deux chiffrer sans même avoir eu besoin de déchiffrer pour faire la multiplication sur le claire.

   exemple:

   $EH[1] \cdot EH[2] = (gr1, m1hr1)\ (gr2, m2hr2) = (gr1+r2, m1m2hr1+r2) = [m1 \cdot mx2]$

   (a) E-Voting

i. Explication

J'ai donc crée une fonction qui reproduit un système de vote ou les électeurs (nombre choisi par l'utilisateur), vote de manière random pour un candidat (nombre choisi par l'utilisateur) de manière à chiffrer jusqu'à qu'un candidat ai une majorité de vote. Sinon, un nouveau tour de vote est relancé.
La multiplicative Homomorphic est parfaite ici, car les personnes peuvent voter pour le candidat en chiffrant un certain nombre qui sera multiplié par le nombre spécial du candidat. Donc, on ne sait jamais qui a voté pour qui, sauf le moment où on déchiffre le nombre de votes pour les candidats.

ii. Variable

$\text{length}_{\text{key}}$ = taille de la clef
$\text{candidat}_{\text{win}}$ = variable du numéro du candidat qui a gagné.
tmp = variable temporaire.
tour = nombre signifiant qu'elle tour on est dans les votes.
$\text{next}_{\text{tour}}$ = tableau contenant les prochains candidats au prochain tour en cas d'égalité.
Candidat = nombre de candidats pour les votes donné par l'utilisateur.
electeur = nombre de électeurs pour les votes donné par l'utilisateur.
urne = tableau qui va contenir les votes chiffrés de chaque électeur pour le candidat qu'ils ont choisi.
tmp2 = tableau sauvegardant le numéro de chaque candidat, au cas où un nouveau tour est engagé.

iii. Code

On demande initialement le nombre de candidats et d'électeurs à l'utilisateur. On initialise le tableau de candidat avec le chiffre chiffré correspondant. On enregistre chaque id de candidat dans tmp2.

```
Candidat = int(input("Entrer le nombre de candidat : "))
    if Candidat < 2:
return
    electeur = int(input("Entrer le nombre de électeur : "))
    if electeur >= length_key:

for i in range (2,Candidat+2):#  Initialise le tableau des scores des canc
nbCandidat.append(encrypt(i,key[1]))
```

```
next_tour.append(i-2)
    tmp2 = next_tour.copy()
```

Par la suite on boucle tant qu'il n'y a pas de vainqueur au élection.

```
while candidat_win == -1:
```

Les électeurs vote aléatoirement pour un candidat et chiffre le chiffre spéciale qu'il multiplie à la case correspondant du candidat.

```
for i in range(2,electeur+2): # Les electeur vote pour le candidat de leur

    vote_pour = random.randint(0,Candidat-1)
    votecrypt = encrypt(vote_pour+2,key[1])
    urne[vote_pour] = (urne[vote_pour][0] * votecrypt[0],urne[vote_pour][1
```

On déchiffre le résultat des votes dans l'urne et on affiche les résultats pour chaque candidat.

```
for i in range(0,Candidat): # On regarde les resultats des votes
    dec = decrypt(urne[i],key)
    result =  int( log((dec+2 /(i+2)) ,i+2)+0.1 ) -1
    if result == tmp:
next_tour.append(tmp2[i])
    if result > tmp :
next_tour.clear()
tmp = result
candidat_win = tmp2[i]
next_tour.append(tmp2[i])

    print("candidat ",tmp2[i], ": ",result, " Vote" )
```

On vérifie si il y a plus de deux candidat pour le prochain tour, on démarré le prochain tour et on réinitialisé les variables comme l'urne, on clear les tableau, on sauvegarde le candidat restant.Sinon, on affiche le gagnant.

```
if len(next_tour) > 1: # On annonce les résulta des votes
    print("Aucun candidat n'a été sélectionner un nouveau tour avec les ca
    tour+=1 # on remet les variables pour initialiser un nouveau tour
    Candidat = len(next_tour)
    urne.clear()
    tmp2.clear()
```

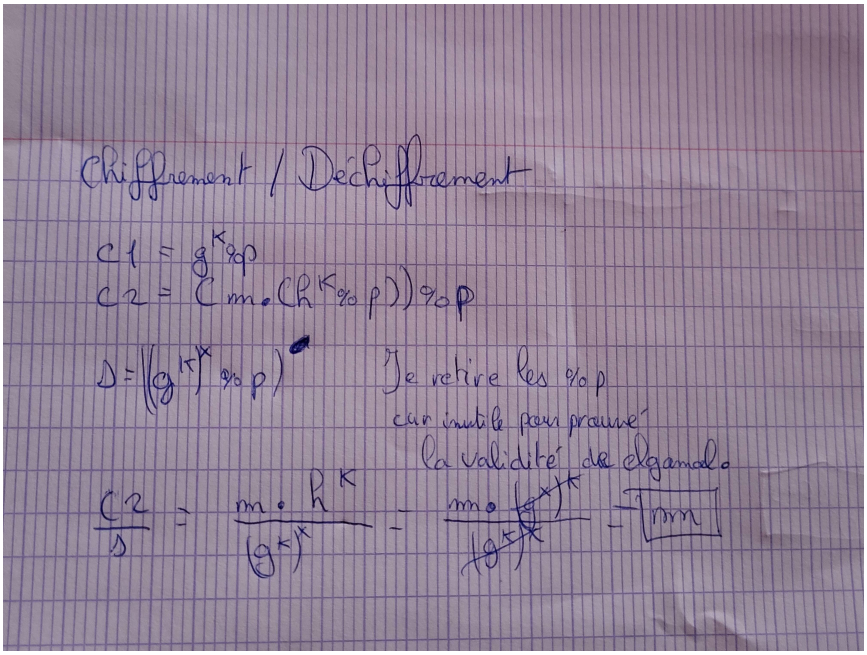
```
    for i in range(0,len(next_tour)):
urne.append(encrypt(i+2,key[1]))
tmp2.append(next_tour[i])
candidat_win =-1
    next_tour.clear()
    time.sleep(2)

else:
    print("Le gagnant des élections est le candidat :",candidat_win,", ave
```

2. Limite

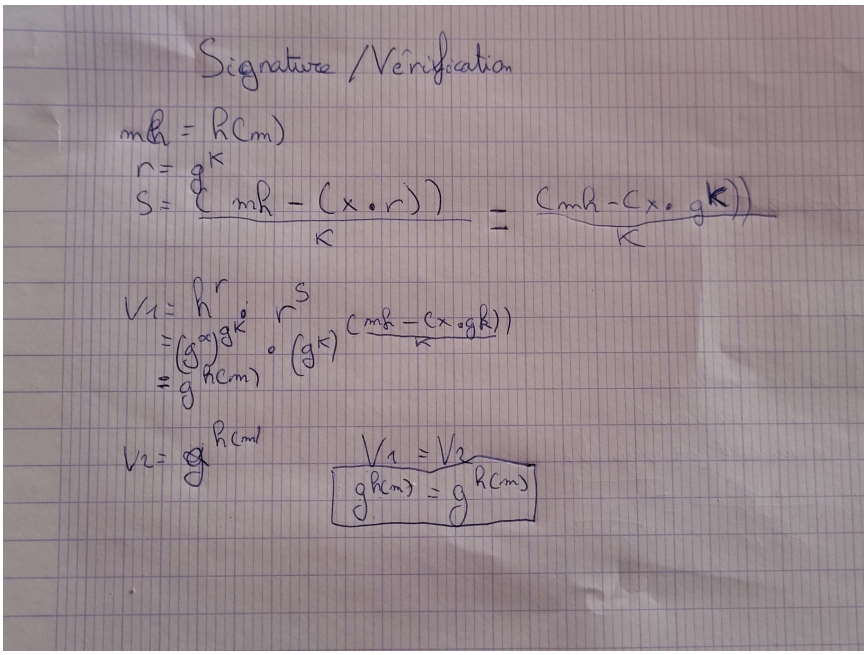
   - La taille de la clef doit être de la même taille que celle du nombre des électeurs.
   - On ne peut pas vérifier si une personne a déjà voté.
   - Le vote n'est pas vraiment sécurisé dans cet exemple, car on sait qui a voté pour qui chaque candidat à son urne.

1.3 Vérification Elgamal

1.3.1 Chiffrement/Déchiffrement



1.3.2 Signature/Vérification



1.4 Avantage

- Algorithme plus sûr que RSA dû au rajout d'aléatoire dans les fonctions. Nous protégeant d'une meilleure façon des attaques par fréquence.
- Possibilité d'effectuer des opérations sur des messages chiffrés sans même besoin d'avoir à les déchiffrer.

1.5 Coût

Le coût est celui de deux exponentiations modulaires :

- ElGamal est beaucoup plus lent que RSA.

Exemple : comparaison entre ElGamal et RSA avec un essaye de génération de clefs, chiffrement et déchiffrement.

```
temps d'execution de rsa 0.006497859954833984
temps d'execution de rsa 0.07082200050354004
temps d'execution de rsa 0.22266364097595215
temps d'execution de elgamal 0.4394526481628418
```

```
temps d'execution de elgamal 28.80167508125305
temps d'execution de elgamal 28.87247657775879
```

- La taille des données chiffrées représente 2 fois celle des données en clair

2 Ce qui étais dure

Comprendre ! Le plus difficile fut vraiment de comprendre tout le vocabulaire mathématique que j'ai encore du mal à maîtriser. De plus, le comprendre et l'utiliser pour pouvoir créer ElGamal étaient la tâche la plus difficile. Entre des heures et des heures à explorer des documents pour comprendre.

3 Ce que j'ai reussi à faire

Je pense avoir accompli la plus par des choses que je voulais faire lorsque je me suis décidé de refaire ElGamal, j'ai même fait un peu plus que ce que j'espérais.

4 ce que je n'ai pas reussi a faire

Peut-être faire un saignement anonyme. Avoir bien implémenté le vote électronique.

5 Mention Honorable

Merci grandement à Julien Lavauzel de m'avoir répondu à mes questions et guider grâce à de la documentation. Lien que j'ai le plus utilisé :

- ElGamal au vote électronique : `https://iopscience.iop.org/article/10.1088/1742-6596/1544/1/012036/pdf`
- signature ElGamal, chiffrement authentifié : `https://cacr.uwaterloo.ca/hac/about/chap11.pdf`
- ElGamal encryption clef : `https://cacr.uwaterloo.ca/hac/about/chap8.pdf`
- Aide pour la génération de clef : `https://cacr.uwaterloo.ca/hac/about/chap4.pdf`

- Wikipédia : https://fr.wikipedia.org/wiki/Cryptosyst%C3%A8me_de_ElGamal
- Quelques video youtube :
  - https://youtu.be/QrsGVeZV7q8
  - https://youtu.be/WQf_mh7h6sk
  - https://youtu.be/Bs3ITCajSZ8