Groupe de TD : DIA 6
Membres : Sara TIEBERGUE, Henri SERANO et Eloi SEIDLITZ
Rendu de 5 pages, 2 pages pour répondre aux exercices et 3 pour les annexes.

Projet Applied Cryptography

Exercice 1

Partie 1

Pour cet exercice, nous avons choisi si la chaine de caractère :
"Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6".

Le nonce que nous avons trouvé est : 408284928

Nous avons obtenu le hash suivant :
9f12fa3ea9dd4cdb820f093c22f67e89f866701f637d49163f50c88940000000

Obtenir cet hash contenant 7 zéros nous a pris 16 minutes et 50.8 secondes.
Nous avons fait une fonction utilisable dans la partie 2. Il est possible de retrouver le code en annexe 1.

Partie 2

Ici nous avons utilisé la même chaine de caractère que pour la première partie en ajoutant les caractères 97 (a) à 126 (~) de la table ascii. Par exemple la première chaine de caractère est "Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6a" et 0.29 secondes ont suffis à trouver un hash avec 5 zéros, cependant il a fallu 1 minute 18.49 secondes pour en trouver un avec 6 zéros.

Nous avons donc répété le processus 30 fois où n zéros est 5 et n+1 zéros est 6.
Le temps moyen pour trouver n zéros (5) était de 1.25 secondes
Le temps moyen pour trouver n+1 zéros (6) était de 30.85 secondes
Il a donc fallu en moyenne 24.70 fois plus de temps pour trouver 6 zéros que 5.

Il est intéressant de noter qu'il est possible que la première chaine de caractère contenant n zéros à la fin en contienne en réalité n+1. C'était le cas pour la chaine de caractère :
"Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6e".

Pour plus de détaille allez sur l'annexe 2.

Exercice 2

Pour obtenir nos password et iv, nous les avons générés aléatoirement pour plus de sécurité.
Nous avons donc obtenu notre entier N que vous pourrez retrouver en déchiffrant C.

Vous trouverez notre fichier encrypté sous le nom ES_HS_ST.enc que nous avons généré grâce à la commande openssl suivante ('password' et 'iv' ne vous sont pas donné):

openssl enc -aes-256-ctr -pbkdf2 -in text_to_encrypt.txt -out ES_HS_ST.enc -pass pass:password -iv iv

Nous avons ensuite généré notre clé privée ‘b’ aléatoirement puis notre clé public B, 308946179518882477945188 5, nous permettant d’obtenir notre chiffrement C de N : 58558499967230330702553 05.

Nous avons testé nos processus pour vérifier qu’ils fonctionnent.
Pour plus d’information n’hésitez pas à consulter l’annexe 3.

Exercice 3

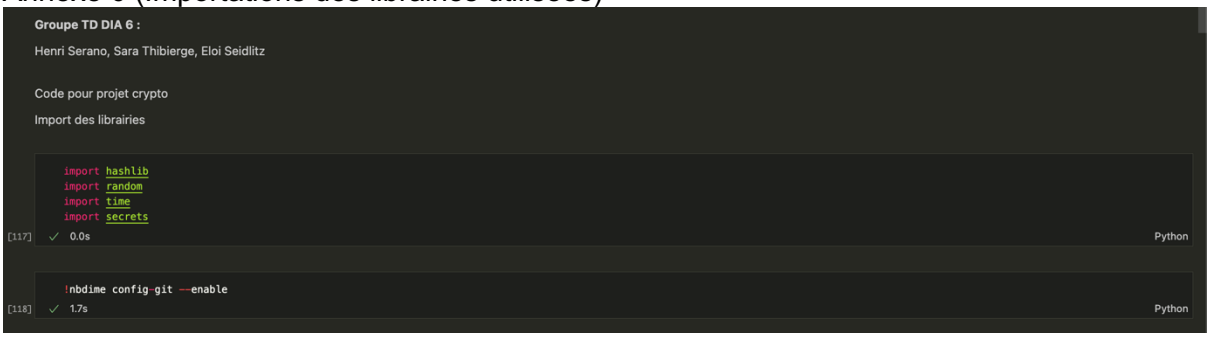
Partie 1

Pour cette partie, nous avons fait la démonstration de la signature ElGamal. Nous avons pris soin de ne pas utiliser de librairies ou packages qui faisaient tout à notre place.
Je vous invite à aller consulter l’annexe 4 pour comprendre comment nous avons procédé.
Pour vous aider le code a été documenté.

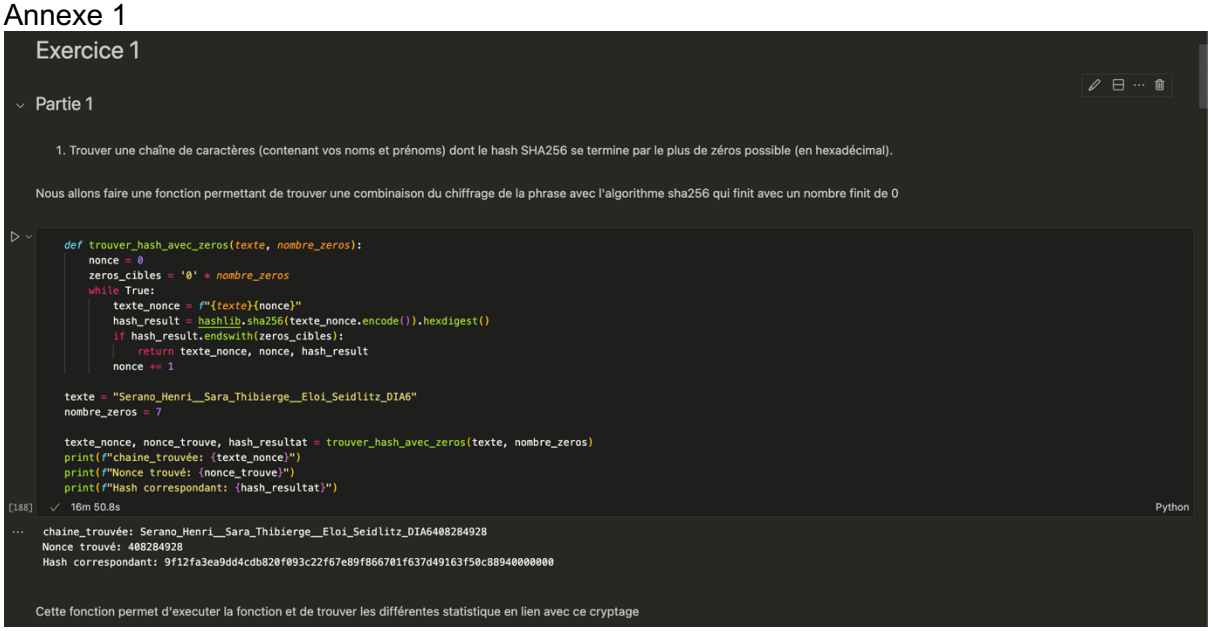
Partie 2

Pour cette seconde partie, nous avons fait un exemple d’utilisation du RSA dans openssl.
Comme pour la première partie, je vous invite à aller consulter l’annexe 5 pour comprendre comment nous avons procédé. Pour vous aider le code a été documenté.

Annexe 0 (Importations des librairies utilisées)



Annexe 1



Annexe 2

2. Mesurer le temps moyen pour obtenir n et n+1 zéros en fin de chaîne (n = 5) et calculer le rapport Tn+1/Tn.

```python
def trouver_chaine_et_stats(base, zeros_cibles, iterations_stat=30):

    temps_n = []
    temps_n_plus_1 = []
    print("|                Chaine de caractère               |      Temps (n)    \t | nonce pour n zéros\t|     Temps (n+1)\t| nonce pour n+1 zéros\t|")
    print("|\t\t\t\t\t\t\t    |\t\t\t\t | \t\t\t|\t\t\t| \t\t\t|")

    for i in range(iterations_stat):
        start = time.time()
        texte_nonce_n, nonce_trouve_n, hash_resultat_n = trouver_hash_avec_zeros(base+chr(97+i), zeros_cibles)
        stop = time.time()
        temps_n.append(stop - start)

        start = time.time()
        texte_nonce_n_plus_1, nonce_trouve_n_plus_1, hash_resultat_n_plus_1 = trouver_hash_avec_zeros(base+chr(97+i), zeros_cibles + 1)
        stop = time.time()
        temps_n_plus_1.append(stop - start)
        print(f"| {base+chr(97+i)} | {temps_n[-1]}    \t | {nonce_trouve_n}   \t\t| {temps_n_plus_1[-1]}\t| {nonce_trouve_n_plus_1}   \t\t| ")

    temps_moyen_n = sum(temps_n) / len(temps_n)
    temps_moyen_n_plus_1 = sum(temps_n_plus_1) / len(temps_n_plus_1)

    rapport_temps = temps_moyen_n_plus_1 / temps_moyen_n

    return {
        "nombre_n_de_zeros": zeros_cibles,
        "temps_moyen_n": temps_moyen_n,
        "temps_moyen_n+1": temps_moyen_n_plus_1,
        "rapport_temps": rapport_temps
    }
```

```python
base_chaine = "Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6"
zeros_cibles = 5 # Nombre de zéros à la fin du hash
resultats = trouver_chaine_et_stats(base_chaine, zeros_cibles)
resultats
```

[131] ✓ 16m 3.0s

Chaine de caractère	Temps (n)	nonce pour n zéros	Temps (n+1)	nonce pour n+1 zéros
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6a	0.28757810592651367	162278	78.48772192001343	43659971
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6b	4.660536050796509	2722191	4.6642560958862305	2722191
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6c	1.3069190979003906	827772	2.218435049057007	1315884
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6d	0.30907392501831055	193292	9.991142988204956	5704604
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6e	1.0553538799285889	664677	1.0682461261749268	664677
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6f	4.193653106689453	2452655	28.836936950683594	16628857
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6g	0.08840608596801758	55416	0.19530701637268066	120432
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6h	1.009490966796875	640851	2.4475419521331787	1481607
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6i	0.8160569667816162	515655	91.97642016410828	51300004
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6j	0.454132080078125	280173	51.734090089797974	28564737
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6k	0.6808598041534424	260034	94.52153491973877	45895019
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6l	0.034340858459472656	21821	48.162293910980225	27320939
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6m	2.45575213432312	1506486	51.74160695075989	29235813
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6n	1.8820819854736328	1161735	8.013759136199951	4583459
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6o	1.0005040168762207	626284	31.895952939987183	18130895
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6p	2.3474302291870117	1140445	89.96188426017761	48215548
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6q	0.5639002323150635	344653	41.31315040588379	22958483
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6r	0.9218370914459229	561347	0.9156239032745361	561347
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6s	0.9829459190368652	593827	20.338923931121826	10338289
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6t	0.7596499919891357	395832	68.97696995735168	37718374
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6u	1.2025880813598633	747274	11.91415810585022	6738483
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6v	1.3096160888671875	800341	25.673504114151	14764669
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6w	2.1440908908843994	1312489	56.4529709815979	31830812
...				
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6{	0.32736897468566895	223082	21.149573802947998	12887729
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6|	0.16135215759277344	107006	6.778410196304321	4181061
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6}	2.5789082050323486	1668744	17.919098138809204	10998943
Serano_Henri__Sara_Thibierge__Eloi_Seidlitz_DIA6~	0.9464640617370605	656374	1.067173957824707	738004

Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...

```
{'nombre_n_de_zeros': 5,
 'temps_moyen_n': 1.249000573158264,
 'temps_moyen_n+1': 30.853226733207702,
 'rapport_temps': 24.702331925429956}
```

Annexe 3

Exercice 2

1. Creation of a function that will create a number with a chosen lenth (ex: 6 to 9) and ramdom digits using only the ones given (ex: 1, 2, 3, 4, 5, 6, 7, 8, 9)

```python
def generate_selected_digit_number(length,selected_digit):
    return int(''.join(secrets.choice(selected_digit) for _ in range(length)))
```

[182] ✓ 0.0s Python

2. Creation of the password and the initial vector

```python
non_zero = "123456789"
password = generate_selected_digit_number(random.randint(6, 9),non_zero)
iv = generate_selected_digit_number(random.randint(6, 9),non_zero)
```

[183] ✓ 0.0s Python

3. get the integer N

```python
N = int(str(password) + "0000" + str(iv))
```

[184] ✓ 0.0s Python

+ Code + Markdown

```python
# print(f"Password:\t{password}")
print(f"IV:\t\t{iv}")
# print(f"N:\t\t{N}")
```

[195] ✓ 0.0s Python

4. Now lets encrypt our text with AES 256 CTR PBKDF2 using our keys

Keep in mind that we remplaced our N and iv by there name to avoid you to see there value

```bash
!openssl enc -aes-256-ctr -pbkdf2 -in text_to_encrypt.txt -out ES_HS_ST.enc -pass pass:password -iv iv
!openssl enc -d -aes-256-ctr -pbkdf2 -in ES_HS_ST.enc -out text_to_encrypt2.txt -pass pass:password -iv iv
```

[186] ✓ 0.7s bash

```
hex string is too short, padding with zero bytes to length
hex string is too short, padding with zero bytes to length
```

```python
# Le nombre premier p :
p = 79468513246798546132458 23
# Le « générateur » d'un groupe d'ordre élevé :
g = 5
# Clé publique A de Herbert Groscot:
A = 75795017959881223934229 86

b = random.SystemRandom().randint(2, p-2)
```

[189] ✓ 0.0s Python

```python
# print(f"Clé privée b de Eloi, Henri et Sara: \t{b}")
```

[194] ✓ 0.0s Python

```python
# Calcul de B et C pour ElGamal
B = pow(g, b, p)
K = pow(A, b, p)
C = (K * N) % p

print(f"Clé publique B de Eloi, Henri et Sara: \t{B}")
print(f"Chiffrement C de lentier N: \t\t{C}")
```

[191] ✓ 0.0s Python

```
Clé publique B de Eloi, Henri et Sara:  308946179518882477945 1885
Chiffrement C de lentier N:             585584999672303307025 5305
```

Annexe 4

Exercice 3

- Démonstration de la signature ElGamal ou RSA (avec des nombres différents de ceux du cours).
- Exemple d'utilisation du RSA, Diffie-Hellman, ElGamal ou de la signature ElGamal dans Python ou openssl (contexte réel).

1. Démonstration de la signature ElGamal (avec des nombres différents de ceux du cours).

```python
# Fonction pour calculer l'inverse modulo
def inverse_modulo(a, m):
    m0, x0, x1 = m, 0, 1
    while a > 1:
        q = a // m
        m, a = a % m, m
        x0, x1 = x1 - q * x0, x0
    return x1 + m0 if x1 < 0 else x1

# Fonction pour calculer le PGCD
def pgcd(a, b):
    while b:
        a, b = b, a % b
    return a

# Fonction pour calculer la signature
def sign_elgamal(m, x, p, g):
    # Sélection d'un nombre aléatoire k
    k = random.randint(2, p - 2)
    while pgcd(k, p - 1) != 1:  # Assure que k est premier avec p-1
        k = random.randint(2, p - 2)
    r = pow(g, k, p)
    k_inv = inverse_modulo(k, p - 1)
    s = (m - x * r) * k_inv % (p - 1)
    return r, s

# Fonction pour vérifier la signature
def verify_elgamal(m, r, s, y, p, g):
    v1 = pow(g, m, p)
    v2 = (pow(y, r, p) * pow(r, s, p)) % p
    return v1 == v2
# Paramètres ElGamal
p = 257
g = 3
x = 97

# Clé publique
y = pow(g, x, p)

# Message à signer
m = 123

# Signature
r, s = sign_elgamal(m, x, p, g)

# Vérification de la signature
valid = verify_elgamal(m, r, s, y, p, g)

print("Signature valide :", valid)
```

[181] ✓ 0.0s Python

```
Signature valide : True
```

Annexe 5

2. Exemple d'utilisation du RSA dans openssl (contexte réel).

Supposons que nous ayons un fichier texte appelé "format.txt" contenant le message à signer.

a) Génération des clés RSA : Tout d'abord, nous devons générer une paire de clés RSA privée/publique à l'aide d'OpenSSL.

```bash
!openssl genpkey -algorithm RSA -out private_key.pem
!openssl rsa -pubout -in private_key.pem -out public_key.pem
```

[196] ✓ 0.5s bash

```
..+....+...+...........+.......+.....+.......+..+.........+....+...+.....+...+.......+..+............+...+.......+...+..+.+.....+...+..................++++++++++++++++++++++++++++++++++++++++
...+...+..............+....+..........+.+.....+.+.....+...+++++++++++++++++++++++++++++++++++++++++*....+......+.........+..+.......+...+..+++++++++++++++++++++++++++++++++++++++*...+....
writing RSA key
```

Ces commandes génèrent une clé privée private_key.pem et une clé publique correspondante `public_key.pem`.

b) Signature du message :

Ensuite, nous signons le message avec notre clé privée.

+ Code + Markdown

```bash
!openssl dgst -sha256 -sign private_key.pem -out signature.bin format.txt
```

[197] ✓ 0.1s bash

Cela crée une signature du message dans le fichier signature.bin.

c) Vérification de la signature :

Pour vérifier la signature avec la clé publique correspondante, nous utilisons :

```bash
!openssl dgst -sha256 -verify public_key.pem -signature signature.bin format.txt
```

[198] ✓ 0.2s bash

```
Verified OK
```

Si la vérification réussit, OpenSSL affichera "Verified OK".