ELECTRONIC VOTING SYSTEM

Crypto Report

15 Avril 2024

BOISSON Thomas
COCHEPIN Axel

"La cryptologie est le moteur d'internet. Aujourd'hui, plus personne ne regarde sous le capot de sa voiture." - Jean-Philippe Aumasson

1. Résumé de l'incident **3**
1.1. Rappel du Contexte 3
2. DSA Algorithm **3**
2.1. DSA signature implementations 3
2.2. Signature implementation test 5
3. El Gamal encryption algorithm **6**
3.1. Multiplicative version 6
3.2. Homomorphic encryption : multiplicative version 8
3.3. Homomorphic encryption : additive version 11
4. Elliptic Curves Cryptography **13**
5. ECDSA signature algorithm **13**
5.1. ECDSA signature implementations 13
5.2. Signature implementation test 14
6. EC El Gamal encryption algorithm **16**
6.1. Implementation 16
6.2. Homomorphic encryption : additive version 17
7. Electronic Voting **19**
7.1. Privacy 19
7.2. Eligibility 19
7.3. Implementation 19

1. Résumé de l'incident

1.1. Rappel du Contexte

Objective of this project is to implement an electronic voting system, based on cryptographic mechanisms. This voting system is simple and covers only a few properties of a real electronic voting system:

- Vote privacy.
- Vote elligibility.
- Homomorphic tally.

In particular, this voting system does not cover other important properties, such that:

- Voter authentication.
- Proof of valid vote.
- Individual verifiability.
- Universal verifiability.

2. DSA Algorithm

2.1. DSA signature implementations

Le fichier **dsa.py** resume l'utilisation de l'algorithme DSA possédant trois fonctions :

- DSA_generate_nonce
- DSA_generate_keys
- DSA_sign
- DSA_verify

Avant d'implémenter ces fonctions, nous avons besoin de connaître le fonctionnement théorique de l'algorithme (mathématiquement) avant de pouvoir le traduire en code python.

1. Génération de clés DSA

Le DSA commence par la génération d'une paire de clés, une publique et une privée. Ce processus suit ces étapes :

- Choix des paramètres : Choix de deux nombres premiers p et q ou q est un diviseur de p-1.
- On choisit également une base g telle que g soit un générateur d'un sous groupe de Z*p d'ordre q.
- Clé privée : La clé privée est un nombre x, choisi aléatoirement dans l'intervalle [1,q-1]
- Clé publique : La clé publique y est calculée comme y = g^x mod p

2. Génération de signature DSA

Pour signer un message m, on procède comme suit :

- Choix d'une nonce : Sélectionner un nombre aléatoire k dans l'intervalle [1, q-1]. Ce nombre doit être gardé secret et ne jamais être réutilisé avec la même clé privée.
- Calcul de r : Calculer r = (g^k mod p) mod q. Si r = 0 on sélectionne un autre k.
- Calcul de s : Calculer s = k^-1 (H(m)+xr) mod q ou H(m) est la valeur de hachage du message m et k^-1 est l'inverse modulaire de k modulo q. Si s = 0, sélectionner un autre k.

La paire (r, s) constitue la signature du message.

3. Vérification de la signature DSA

- Pour vérifier une signature (r,s) d'un message m avec une clé publique y, on suit ces étapes:

  Calcul de w : Calculer l'inverse modulaire de s, w = s^-1 mod q.
- Calcul du u1 et u2: Calculer u1 = H(m)*w mod q et u2 = (r*w) mod q.
- Calcul de v : Calculer v = ((g^u1*y^u2 mod p) mod q)

Si v = r, alors la signature est valide ; sinon, elle est invalide.

Avec la théorie nous pouvons désormais implémenter les fonctions demandés

```python
def DSA_generate_nonce(q):
    return random.randint(1, q-1)


def DSA_generate_keys(PARAM_P, PARAM_Q, PARAM_G):
    key = DSA.generate(2048, randfunc=None)
    key.p = PARAM_P
    key.q = PARAM_Q
    key.g = PARAM_G
    private_key = random.StrongRandom().randint(1, key.q - 1)
    public_key = pow(key.g, private_key, key.p)
    return private_key, public_key


def DSA_sign(message, private_key, k, PARAM_P, PARAM_Q, PARAM_G):
    r = pow(PARAM_G, k, PARAM_P) % PARAM_Q
    hash_value = int.from_bytes(SHA256.new(message.encode()).digest(), byteorder='big')
    s = (mod_inv(k, PARAM_Q) * (hash_value + private_key * r)) % PARAM_Q
    return r, s


def DSA_verify(message, r, s, public_key, PARAM_P, PARAM_Q, PARAM_G):
    w = mod_inv(s, PARAM_Q)
    hash_value = int.from_bytes(SHA256.new(message.encode()).digest(), byteorder='big')
    u1 = (hash_value * w) % PARAM_Q
    u2 = (r * w) % PARAM_Q
    v = ((pow(PARAM_G, u1, PARAM_P) * pow(public_key, u2, PARAM_P)) % PARAM_P) % PARAM_Q
    return v == r
```

Implémentation de dsa.py

2.2. Signature implementation test

We still use **SHA256** as hash function and **MODP Group 24** for public parameters. Let m (a message), k (the nonce used in signature generation) and x (signature private key) defined with:

$$
\begin{aligned}
m &= \texttt{An important message !} \\
k &= \texttt{0x7e7f77278fe5232f30056200582ab6e7cae23992bca75929573b779c62ef4759} \\
x &= \texttt{0x49582493d17932dabd014bb712fc55af453ebfb2767537007b0ccff6e857e6a3}
\end{aligned}
$$

Use your implementation of DSA signature algorithm and verify that you obtain (r, s) as signature, defined with:

$$
\begin{aligned}
r &= \texttt{0x5ddf26ae653f5583e44259985262c84b483b74be46dec74b07906c5896e26e5a} \\
s &= \texttt{0x194101d2c55ac599e4a61603bc6667dcc23bd2e9bdbef353ec3cb839dcce6ec1}
\end{aligned}
$$

Dans l'exercice suivant, nous allons enlever le "0x" de chaque valeur hex, nous transformons le type des valeurs attendus en hexa et nous comparons le tout en .lower() dans le but que python comprenne bien les valeurs comparés

```
47  # On teste les parametres m x et k et on regarde si on obtient les valeurs attendues
48
49  expected_r = 0x5ddf26ae653f5583e44259985262c84b483b74be46dec74b07906c5896e26e5a
50  expected_s = 0x194101d2c55ac599e4a61603bc6667dcc23bd2e9bdbef353ec3cb839dcce6ec1
51
52  m = "An important message !"
53  k = 0x7e7f77278fe5232f30056200582ab6e7cae23992bca75929573b779c62ef4759
54  x = 0x49582493d17932dabd014bb712fc55af453ebfb2767537007b0ccff6e857e6a3
55
56  r, s = DSA_sign(m, x, k, PARAM_P, PARAM_Q, PARAM_G)
57
58  print(hex(r)[2:])
59  print(hex(s)[2:])
60  expected_r_hex = hex(expected_r)[2:].lower()  # Convertit en chaîne hexadécimale et en minuscules.
61  expected_s_hex = hex(expected_s)[2:].lower()  # Convertit en chaîne hexadécimale et en minuscules.
62  print(hex(r)[2:].lower() == expected_r_hex)
63  print(hex(s)[2:].lower() == expected_s_hex)
64
65
66
```

```
PROBLÈMES 81   TERMINAL   PORTS   POLYGLOT NOTEBOOK   CONSOLE DE DÉBOGAGE

thomas-boisson@THOMAS-GAMER:/mnt/c/Users/Torte/Desktop/project$ python3 dsa.py
5ddf26ae653f5583e44259985262c84b483b74be46dec74b07906c5896e26e5a
194101d2c55ac599e4a61603bc6667dcc23bd2e9bdbef353ec3cb839dcce6ec1
True
True
thomas-boisson@THOMAS-GAMER:/mnt/c/Users/Torte/Desktop/project$
```

Test des valeurs attendus avec la fonction DSA_Sign

3. El Gamal encryption algorithm

3.1. Multiplicative version

Toujours dans le même thème que l'énoncé précédent, dans le fichier **elgamal.py** nous devons implémenter 3 fonctions sous l'algorithme de El Gamal :

- Générations de clefs
- Algorithme de Chiffrement
- Algorithme de Déchiffrement

Expliquons d'abord l'algorithme afin de comprendre et implémenter les fonctions exécutant le fonctionnement de l'algorithme

Génération de clefs [modifier | modifier le code]

Gamal (en bleu).

La première étape du schéma de chiffrement consiste à produire une paire de clefs : la *clef publique*, et la *clef secrète*. La première servira à chiffrer les messages et la deuxième à les déchiffrer.

- Pour générer sa paire de clefs, Alice va commencer par prendre un groupe cyclique G d'ordre q dans lequel le problème hypothèse décisionnelle de Diffie-Hellman est difficile, ainsi qu'un générateur g de ce groupe.
- Alice va ensuite tirer un élément **sk** $\triangleq x \hookleftarrow U(\mathbb{Z}_q^*)$ qui va être sa *clef privée*, et va calculer $h = g^x$.
- Pour terminer, Alice va publier **pk** $\triangleq (G, q, g, h)$ comme étant sa *clef publique*.

Algorithme de Chiffrement [modifier | modifier le code]

Bob a donc accès à la clef publique d'Alice : **pk** $= (G, q, g, h)$. Pour chiffrer un message m encodé comme un élément du groupe G, Bob commence par tirer un *aléa* $r \hookleftarrow U(\mathbb{Z}_q^*)$ et va l'utiliser pour couvrir le message m en calculant $c_2 = m \cdot h^r$. Pour permettre à Alice de déchiffrer le message, Bob va adjoindre à cette partie du message une information sur l'aléa : $c_1 = g^r$.

Enfin le chiffré sera composé de ces deux morceaux : $C = (c_1, c_2)$, et Bob envoie $C \in G^2$ à Alice.

Algorithme de Déchiffrement [modifier | modifier le code]

Ayant accès à $C = (c_1, c_2)$ et à **sk** $= x$, Alice peut ainsi calculer :

$$\frac{c_2}{{c_1}^x} = \frac{m \cdot h^r}{g^{r \cdot x}} = \frac{m \cdot \cancel{g^{r \cdot x}}}{\cancel{g^{r \cdot x}}} = m$$

Et est donc en mesure de retrouver le message m.

Source: [wikipedia]

Imaginons un scénario pour expliquer l'algorithme a travers les fonctions illustrées ci dessus:

Imaginons que vous avez un coffre-fort avec un verrou qui a deux clés :

- une clé publique qui verrouille le coffre
- une clé privée qui le déverrouille

Seul le destinataire du message à la clé privée pour déverrouiller le coffre, mais tout le monde peut utiliser la clé publique pour y mettre un message.

1. Création des clés :

Le destinataire crée un coffre-fort (génération de clés) et garde la clé privée secrète.
Il donne ensuite une copie de la clé publique à quiconque souhaite lui envoyer un message.

2. Chiffrement du message :

L'expéditeur prend son message et l'emballe dans une boîte qu'il va verrouiller avec la clé publique du destinataire (chiffrement).
Il ajoute aussi un peu de mystère à la boîte pour s'assurer que même si quelqu'un a beaucoup de ces boîtes verrouillées, elles semblent toutes différentes à l'extérieur (c'est la nonce k).
Il envoie ensuite cette boîte verrouillée au destinataire.

3. Chiffrement du message :

Le destinataire reçoit la boîte verrouillée et utilise sa clé privée secrète pour l'ouvrir (déchiffrement).
Il retire le mystère ajouté par l'expéditeur et récupère le message original.

Ainsi nous avons une idée mathématiques et théorique de comment fonctionne l'algorithme, l'implémentation donne ainsi :

```python
def EG_generate_keys():
    x = randint(1, PARAM_Q - 1)
    y = pow(PARAM_G, x, PARAM_P)
    return (x, y)

## multiplicative version
def EGM_encrypt(m, y, PARAM_P, PARAM_G):
    k = randint(2, PARAM_P - 1)
    c1 = pow(PARAM_G, k, PARAM_P)
    c2 = (m * pow(y, k, PARAM_P)) % PARAM_P
    return (c1, c2)

## additive version
def EGM_encrypt2(m, y, PARAM_P, PARAM_G):
    TODO

def EG_decrypt(c1, c2, x, PARAM_P):
    s = pow(c1, x, PARAM_P)
    m = (c2 * mod_inv(s, PARAM_P)) % PARAM_P
    return m
```

Implémentation de elgamal.py

3.2. Homomorphic encryption : multiplicative version

L'exercice demande d'utiliser l'algorithme d'El Gamal pour chiffrer deux messages et démontrer la propriété homomorphe de la version multiplicative de l'algorithme.

Voici les messages m1 et m2 que nous utiliserons dans l'exercice

```python
# Valeurs données par l'exercice pour le test
m1 = 0x26616b7368f687c5c3142f806d500d2ce57b1182c9b25bf4efa09529424b
m2 = 0x1c1c871caabca15828cf08ee3aa3199000b94ed15e743c3
```

messages m1 et m2

Dans l'ordre nous utiliserons les fonctions implémenter pour :

Créer une clé publique et une clé privée

```
# Génération des clés El Gamal
private_key, public_key = EG_generate_keys()
```

Chiffrer les messages m1 et m2 avec la clé publique crée

```
# Chiffrement des messages m1 et m2
r1, c1 = EGM_encrypt(m1, public_key, PARAM_P, PARAM_G)
r2, c2 = EGM_encrypt(m2, public_key, PARAM_P, PARAM_G)
```

Effectuer une opération homomorphe sur les messages chiffrés

```
# Opération homomorphe sur les messages chiffrés
r3 = (r1 * r2) % PARAM_P
c3 = (c1 * c2) % PARAM_P
```

On déchiffre le résultat homomorphe en message m3

```
# Déchiffrement du résultat homomorphe
m3 = EG_decrypt(r3, c3, private_key, PARAM_P)
```

On vérifie que m3 est égale au produit de m1 et m2 modulo PARAM_P

```
# Vérification que m3 est égal à m1 multiplié par m2 modulo PARAM_P
assert m3 == (m1 * m2) % PARAM_P, "Le déchiffrement homomorphe ne correspond pas au produit des messages."
```

Si l'assert ne se déclenche pas alors la condition est vrai

```python
# Homomorphic encryption : multiplicative version

# Valeurs données par l'exercice pour le test
m1 = 0x26616b7368f687c5c3142f806d500d2ce57b1182c9b25bf4efa09529424b
m2 = 0x1c1c871caabca15828cf08ee3aa3199000b94ed15e743c3

def homomorphic_multiplication(m1, m2):
    # Génération des clés El Gamal
    private_key, public_key = EG_generate_keys()

    # Chiffrement des messages m1 et m2
    r1, c1 = EGM_encrypt(m1, public_key, PARAM_P, PARAM_G)
    r2, c2 = EGM_encrypt(m2, public_key, PARAM_P, PARAM_G)

    # Opération homomorphe sur les messages chiffrés
    r3 = (r1 * r2) % PARAM_P
    c3 = (c1 * c2) % PARAM_P

    # Déchiffrement du résultat homomorphe
    m3 = EG_decrypt(r3, c3, private_key, PARAM_P)

    # Vérification que m3 est égal à m1 multiplié par m2 modulo PARAM_P
    if m3 == (m1 * m2) % PARAM_P:
        return True
    else:
        return False

print(homomorphic_multiplication(m1, m2))
```

```
PROBLÈMES 87  TERMINAL  PORTS  POLYGLOT NOTEBOOK  CONSOLE DE DÉBOGAGE

thomas-boisson@THOMAS-GAMER:/mnt/c/Users/Torte/Desktop/project$ python3 elgamal.py
True
thomas-boisson@THOMAS-GAMER:/mnt/c/Users/Torte/Desktop/project$
```

Implémentation de homomorphic_multiplication

Ainsi nous validons l'implémentation des fonctions de l'algorithme El Gamal.

3.3. Homomorphic encryption : additive version

3.3 Homomorphic encryption : additive version

You have implemented El Gamal encryption such that for two messages m_1 and m_2, **EG_Encrypt**$(m_1) \times$ **EG_Encrypt**$(m_2) =$ **EG_Encrypt**$(m_1 \times m_2)$.

Explain how you can turn your previous implementation into an *additive* version, i.e: **EGA_Encrypt**$(m_1) \times$ **EGA_Encrypt**$(m_2) =$ **EGA_Encrypt**$(m_1 + m_2)$. Implement it !

In the context of electronic voting, we will use the additive version of El Gamal where messages are 0 or 1. We still use **MODP Group 24** for public parameters.

Let $m_1 = 1$, $m_2 = 0$, $m_3 = 1$, $m_4 = 1$, $m_5 = 0$ five messages.

Use your implementation of El Gamal encryption algorithm (*additive version*) to encrypt these five messages and compute the following process, where **EGA_Encrypt** denotes El Gamal encryption (additive version) and **EG_Decrypt** denotes El Gamal decryption:

- $(r_1, c_1) =$ **EGA_Encrypt**(m_1)
- $(r_2, c_2) =$ **EGA_Encrypt**(m_2)
- $(r_3, c_3) =$ **EGA_Encrypt**(m_3)
- $(r_4, c_4) =$ **EGA_Encrypt**(m_4)
- $(r_5, c_5) =$ **EGA_Encrypt**(m_5)
- Let $(r, c) = (r_1 \times r_2 \times r_3 \times r_4 \times r_5, c_1 \times c_2 \times c_3 \times c_4 \times c_5)$
- Compute **EG_Decrypt**(r, c). Beware that the result of this computation is not directly $m = m_1 + m_2 + m_3 + m_4 + m_5$! Indeed you obtain g^m and not m (where g is **MODP Group 24** public parameter). But as in this situation m is small, a direct brute force search is possible. You will find in file `elgamal.py` an implementation of a brute force search, named `bruteLog`. Use it to compute m.
- Assess $m = m_1 + m_2 + m_3 + m_4 + m_5 = 3$

La différence avec l'opération Homomorphique est dans l'opération des messages chiffrés, en effet ici l'énoncé nous propose d'additionner 5 messages dans le but d'obtenir une somme.

Dans la version additive, lorsqu'on additionne deux messages chiffrés, le résultat est le chiffrement de la somme des messages originaux. Pour que cela soit possible, nous devons adapter le processus de chiffrement pour qu'il supporte cette opération.

Pour cela nous avons besoin d'implémenter une nouvelle version de EGM_encrypt que nous nommerons EGM_encrypt2 qui fera l'opération en addition tout simplement.

```python
## additive version
def EGM_encrypt2(m, y, PARAM_P, PARAM_G):
    k = randint(2, PARAM_P - 1)
    c1 = pow(PARAM_G, k, PARAM_P)
    c2 = pow(y, k, PARAM_P)*pow(PARAM_G, m, PARAM_P) % PARAM_P
    return (c1, c2)
```

Implémentation de EGM_encrypt2 #addition version

Toujours en se basant sur l'implémentation de l'opération homomorphique, nous utiliserons la version additive du chiffrement sur les 5 message et nous retrouverons m à partir de g^m avec l'aide de la fonction BruteLog (donné de base) et enfin comparer le m trouvé a la somme des 5 message pour voir si cela est correct (Nous sommes censé retrouver des deux coté 3).

```python
def homomorphic_addition(L):

    # Génération des clés privées et publiques
    private_key, public_key = EG_generate_keys()
    # Chiffrement des messages avec les clés publiques fournies
    r, c = 1, 1
    for m in L:
        r_i, c_i = EGM_encrypt2(m, public_key, PARAM_P, PARAM_G)
        r = (r * r_i) % PARAM_P
        c = (c * c_i) % PARAM_P

    # Déchiffrement du résultat homomorphe
    gm = EG_decrypt(r, c, private_key, PARAM_P)

    # Utilisation de bruteLog pour trouver m à partir de g^m
    m_decoded = bruteLog(PARAM_G, gm, PARAM_P)

    # Vérification que la somme décodée est égale à la somme des messages originaux
    return m_decoded == sum(L)
```

Implémentation de homomorphic_addition

```
thomas-boisson@THOMAS-GAMER:/mnt/c/Users/Torte/Desktop/project$ python3 elgamal.py
True
thomas-boisson@THOMAS-GAMER:/mnt/c/Users/Torte/Desktop/project$
```

Verification de l'implementation

4. Elliptic Curves Cryptography

Cette section est à titre informatif.

> Consider RFC 7748 (https://www.rfc-editor.org/rfc/rfc7748). This RFC contains some python code and some pseudocode. These allow to implement scalar multiplication in `curve25519`, which is usually named **X25519**.
>
> In file `rfc7748.py` you will find implementations for these functions. They both use an internal function, named **mul**, whose pseudocode is given in RFC 7748. Function **mul** itself uses another function, named **cswap**, for which a proposed implementation is given in file `rfc7748.py`. Implementation also uses encoding and decoding functions, whose python code is given in RFC 7748 and in file `rfc7748.py`.

5. ECDSA signature algorithm

5.1. ECDSA signature implementations

```python
def ECDSA_generate_keys(pkey):
    res = mult(pkey, BaseU, BaseV, p)
    return pkey, res
```

Génération de clés

La fonction **ECDSA_generate_keys** prend un paramètre pkey, qui représente la clé privée. À l'intérieur de la fonction, la clé publique est calculée en utilisant une fonction de multiplication de points sur la courbe elliptique, désignée ici par **mult**. La clé publique résultante est le produit de la clé privée et du point de base (BaseU, BaseV) de la courbe elliptique, effectué modulo p. La fonction retourne ensuite la paire de clés privée et publique.

```python
def ECDSA_sign(message, private_key, nonce):
    z = H(message)
    k = nonce
    k_inv = mod_inv(k, ORDER)
    r = mult(k, BaseU, BaseV, p)[0] % ORDER
    s = (k_inv * (z + r * private_key)) % ORDER
    return r, s
```

Signature d'un message

Pour signer un message, l'émetteur calcule d'abord le hash du message à l'aide d'une fonction de hachage cryptographiquement sécurisée.
Il choisit ensuite un entier k aléatoire entre 1 et n-1 ensuite le point R=kG est calculé, ou G est le point générateur de la courbe.
La coordonnée x de ce point, notée r, et le résultat de s = k**-1(hash+r*cléprivée) mod n composent la signature, Si = 0, un autre k est choisi est le processus est répété.

```python
def ECDSA_verify(message, public_key, r, s):
    z = H(message)
    w = mod_inv(s, ORDER)
    u1 = (z * w) % ORDER
    u2 = (r * w) % ORDER
    x1, y1 = mult(u1, BaseU, BaseV, p)
    x2, y2 = mult(u2, public_key[0], public_key[1], p)
    x, y = add(x1, y1, x2, y2, p)
    return r == x % ORDER
```

Vérification de la signature

Pour vérifier une signature, le vérificateur calcule le hash du message. En utilisant les coordonnées r et s de la signature, il calcule deux valeurs temporaires u1 = (hash * s^(-1)) mod n et u2 = (r * s^(-1)) mod n. Le point R' est calculé comme (u1 * G) + (u2 * clé publique). Si le point R' est à l'infini ou si la coordonnée x de R' n'est pas égale à r, la signature est considérée comme invalide.

5.2. Signature implementation test

We still use **SHA256** as hash function. Let m (a message), k (the nonce used in signature generation) and x (signature private key) defined with:

$m =$ `A very very important message !`
$k =$ `0x2c92639dcf417afeae31e0f8fddc8e48b3e11d840523f54aaa97174221faee6`
$x =$ `0xc841f4896fe86c971bedbcf114a6cfd97e4454c9be9aba876d5a195995e2ba8`

Use your implementation of ECDSA signature algorithm and verify that you obtain (r, s) as signature, defined with:

$r =$ `0x429146a1375614034c65c2b6a86b2fc4aec00147f223cb2a7a22272d4a3fdd2`
$s =$ `0xf23bcdebe2e0d8571d195a9b8a05364b14944032032eeeecd22a0f6e94f8f33`

Check also your signature verification algorithm.

```python
# On teste les parametres m x et k et on regarde si on obtient les valeurs attendues

expected_r = 0x429146a1375614034c65c2b6a86b2fc4aec00147f223cb2a7a22272d4a3fdd2
expected_s = 0xf23bcdebe2e0d8571d195a9b8a05364b14944032032eeeecd22a0f6e94f8f33

m = b"A very very important message !"
x = 0xc841f4896fe86c971bedbcf114a6cfd97e4454c9be9aba876d5a195995e2ba8
k = 0x2c92639dcf417afeae31e0f8fddc8e48b3e11d840523f54aaa97174221faee6

private_key, public_key = ECDSA_generate_keys(x)
r, s = ECDSA_sign(m, x, k)

print(hex(r)[2:])
print(hex(s)[2:])
expected_r_hex = hex(expected_r)[2:].lower()  # Convertit en chaîne hexadécimale et en minuscules.
expected_s_hex = hex(expected_s)[2:].lower()  # Convertit en chaîne hexadécimale et en minuscules.
print(hex(r)[2:].lower() == expected_r_hex)
print(hex(s)[2:].lower() == expected_s_hex)
print(ECDSA_verify(m, public_key, r, s))
```

Test des valeurs recherchés avec les fonctions implémentées

Nous reprenons le meme type d'implémentation effectué dans **dsa.py**.
Dans l'exercice suivant, nous allons enlever le "0x" de chaque valeur hex, nous transformons le type des valeurs attendus en hexa et nous comparons le tout en .lower() dans le but que python comprenne bien les valeurs comparés

Avant, nous générons deux clés, nous signons les messages de test et nous vérifions que les signatures sont cohérentes et validées et enfin nous testons les hash pour voir s' ils sont similaires.

```
thomas-boisson@THOMAS-GAMER:/mnt/c/Users/Torte/Desktop/project$ python3 ecdsa.py
429146a1375614034c65c2b6a86b2fc4aec00147f223cb2a7a22272d4a3fdd2
f23bcdebe2e0d8571d195a9b8a05364b14944032032eeeecd22a0f6e94f8f33
True
True
True
```

Vérification de l'implémentation

6. EC El Gamal encryption algorithm

6.1. Implementation

```python
def ECEG_generate_keys():
    private_key = randint(1, ORDER-1)
    public_key = mult(private_key, BaseU, BaseV, p)
    return private_key, public_key
```

1. **ECEG_generate_keys**

Il s'agit de créer une paire de clés publique et privée. La clé privée est un nombre aléatoire choisi dans l'intervalle de 1 à l'ordre de la courbe elliptique moins un. La clé publique est le résultat de la multiplication de la clé privée par le point de base de la courbe elliptique.

```python
def ECEG_encrypt(message, public_key):
    r = randint(1, ORDER-1)
    R = mult(r, BaseU, BaseV, p)
    shared_secret = mult(r, public_key[0], public_key[1], p)
    S = add(message[0], message[1], shared_secret[0], shared_secret[1], p)
    return R, S
```

2. **ECEG_encrypt**

Pour chiffrer un message, le point correspondant au message est d'abord calculé. Si le message est 0, il est mappé au point à l'infini de la courbe, et si le message est 1, il est mappé au point de base de la courbe. Un nombre aléatoire est choisi et multiplié par le point de base pour créer un point de chiffrement. Ensuite, le point de message est additionné au produit de ce nombre aléatoire et de la clé publique.

```python
def ECEG_decrypt(x, y, private_key):
    shared_secret_point = mult(private_key, x[0], x[1], p)
    decrypted_message_point = sub(y[0], y[1], shared_secret_point[0], shared_secret_point[1], p)
    return decrypted_message_point
```

3. **ECEG_decrypt**

Pour déchiffrer le message, la clé privée est multipliée par le premier point de chiffrement, et le résultat est soustrait du second point de chiffrement. Ceci permet de retrouver le point de message original.

6.2. Homomorphic encryption : additive version

Cette partie du code définit une fonction **test_homomorphic_add** qui teste la propriété homomorphe additive :

- La fonction génère une paire de clés publique et privée pour l'algorithme EC El Gamal.
- Elle transforme une liste de messages binaires (L) en points sur la courbe elliptique en utilisant la fonction EGencode.
- Chaque message de la liste transformée est chiffré avec la clé privée. Les résultats chiffrés sont stockés dans une liste encrypted_results, où chaque élément est un tuple contenant deux points (r, c).
- Les points correspondant aux parties r de tous les messages chiffrés sont additionnés ensemble, ainsi que les points correspondant aux parties c. Cela donne un point r_index et un point c_index qui sont les résultats cumulatifs de l'addition.
- Le point résultant de l'addition homomorphe (r_index, c_index) est déchiffré avec la clé publique.
- La fonction bruteECLog est utilisée pour trouver la valeur originale du message à partir du point déchiffré.
- La fonction retourne un booléen indiquant si la somme des messages décodée est égale à 3, ce qui devrait être le résultat attendu étant donné les messages originaux.

```python
def test_homophoric_add(L):
    publickey, privatekey = ECEG_generate_keys()
    r_index = (0, 0)
    c_index = (0, 0)
    L = [EGencode(m) for m in L]

    encrypted_results = []
    for message in L:
        r, c = ECEG_encrypt(message, privatekey)
        encrypted_results.append((r, c))

    r_list = [r for r, c in encrypted_results]
    c_list = [c for r, c in encrypted_results]

    for r in r_list:
        r_index = add(r_index[0], r_index[1], r[0], r[1], p)

    for c in c_list:
        c_index = add(c_index[0], c_index[1], c[0], c[1], p)

    decrypted_message = ECEG_decrypt(r_index, c_index, publickey)
    m_result = bruteECLog(decrypted_message[0], decrypted_message[1], p)
    return m_result == 3


print(test_homophoric_add(L))
```

Implémentation de test_homophoric_add

```
thomas-boisson@THOMAS-GAMER:/mnt/c/Users/Torte/Desktop/project$ python3 ecelgamal.py
True
True
thomas-boisson@THOMAS-GAMER:/mnt/c/Users/Torte/Desktop/project$
```

output de test_homophoric_add

7. Electronic Voting

7.1. Privacy

Dans notre système de vote électronique, la confidentialité des votes est assurée par le chiffrement de chaque élément du bulletin de vote. Chaque vote, représentant un choix parmi cinq candidats, est transformé en une liste de cinq éléments où seul l'élément

correspondant au candidat choisi est marqué par un '1', les autres étant des '0'. Comme illustré dans notre fonction **simulate_voting_system**, chaque partie de ce vote est ensuite cryptée individuellement en utilisant la fonction **EGM_encrypt**. Cette méthode garantit que chaque message chiffré reste confidentiel et indéchiffrable sans accès à la clé privée correspondante.

```
encrypted_vote = [EGM_encrypt(v, public_keys[random.randint(0, num_voters-1)], PARAM_P, PARAM_G) for v in vote]
```

De plus, pour renforcer la confidentialité et empêcher tout lien possible entre le vote chiffré et l'électeur, nous utilisons la propriété homomorphique de l'encryption pour seulement décrypter la somme agrégée de tous les choix, plutôt que les bulletins individuels. Cela permet d'obtenir le total des votes pour chaque candidat sans jamais révéler le contenu des votes individuels.

7.2. Eligibility

Pour assurer l'éligibilité des électeurs, chaque bulletin de vote chiffré est signé à l'aide de la clé de signature privée de l'électeur, distribuée par le système de vote. Le code suivant montre comment chaque vote est signé après le chiffrement, utilisant la fonction **DSA_sign**, et comment ces signatures sont ensuite vérifiées pour chaque vote reçu.

```
vote_str = ''.join(str(v[1]) for v in encrypted_vote)

k = DSA_generate_nonce(PARAM_Q)
r, s = DSA_sign(vote_str, private_key, k, PARAM_P, PARAM_Q, PARAM_G)
```

7.3. Implementation

- Génération et distribution des clés de signatures des électeurs : une paire de clés de signature pour chaque électeur.

```
voters_keys = [DSA_generate_keys(PARAM_P, PARAM_Q, PARAM_G) for _ in range(num_voters)]

votes = []
public_keys = [public_key for _, public_key in voters_keys]
```

- Génération des bulletins de vote : chaque bulletin de vote contient 5 messages cryptés.

```
for private_key, _ in voters_keys:
    chosen_candidate = random.randint(0, num_candidates-1)
    vote = [0] * num_candidates
    vote[chosen_candidate] = 1


    encrypted_vote = [EGM_encrypt(v, public_keys[random.randint(0, num_voters-1)], PARAM_P, PARAM_G) for v in vote]

    vote_str = ''.join(str(v[1]) for v in encrypted_vote)

    k = DSA_generate_nonce(PARAM_Q)
    r, s = DSA_sign(vote_str, private_key, k, PARAM_P, PARAM_Q, PARAM_G)

    votes.append((encrypted_vote, (r, s)))
```

- Multiplication des bulletins : cela génère 5 messages cryptés (un pour chaque candidat).

```
aggregated_votes = [1] * num_candidates
for vote, _ in votes:
    for i in range(num_candidates):
        aggregated_votes[i] = (aggregated_votes[i] * vote[i][1]) % PARAM_P
```

- Cinq décryptages et recherches par force brute pour récupérer le résultat de l'élection (nombre de voix par candidat).

```
decrypted_aggregates = [EG_decrypt(r, aggregated_votes[i], private_key, PARAM_P) for i in range(num_candidates)]

results = []
for i in range(num_candidates):
    results.append(bruteLog(PARAM_G, decrypted_aggregates[i], PARAM_P))
```