Ecole supérieure en science et technologie de l'informatique et du numérique ESTIN

Compte rendu TP 05

Cryptographie à base des courbes elliptiques

Réalisé par :

Benmedakhene Souha (G2) .

Encadré par :

Dr . Bouchoucha Lydia

Année universitaire : 2023/2024

Introduction :

La cryptographie à base des courbes elliptiques représente un pilier fondamental de la sécurité informatique contemporaine, exploitant des principes mathématiques avancés pour garantir la confidentialité et l'intégrité des données échangées dans les environnements numériques. Contrairement aux techniques traditionnelles de cryptographie, telles que le chiffrement RSA, qui reposent sur la complexité de la factorisation de grands nombres premiers, la cryptographie à base des courbes elliptiques tire profit des propriétés algébriques des points situés sur des courbes elliptiques pour assurer la sécurité des communications.

Dans ce compte rendu, nous entreprendrons un examen approfondi des fondements théoriques de la cryptographie à base des courbes elliptiques, ainsi que de ses applications pratiques dans les infrastructures de sécurité modernes. Nous explorerons les mécanismes sous-jacents qui rendent cette méthode de cryptographie robuste et résiliente face aux attaques informatiques, ainsi que les avantages qu'elle offre en termes d'efficacité et de performance par rapport à d'autres techniques cryptographiques.

En outre, nous analyserons les défis et les limitations associés à l'implémentation de la cryptographie à base des courbes elliptiques, notamment en ce qui concerne la sélection de paramètres appropriés, la gestion des clés et la compatibilité avec les normes de sécurité existantes. Comprendre ces aspects est crucial pour évaluer de manière critique l'adoption et l'intégration de cette technologie dans différents contextes informatiques, allant de la sécurisation des transactions financières à la protection des données personnelles.

En synthèse, ce compte rendu vise à fournir une vue d'ensemble complète de la cryptographie à base des courbes elliptiques, en mettant en lumière son importance croissante dans le paysage de la sécurité numérique contemporaine et en soulignant son rôle central dans la préservation de la confidentialité et de l'intégrité des communications numériques.

Objectifs :

1. Comprendre les principes fondamentaux de la cryptographie à base des courbes elliptiques, en explorant les propriétés mathématiques des courbes elliptiques et leur utilisation dans les protocoles cryptographiques.
2. Acquérir une expérience pratique dans la génération de clés et l'utilisation des courbes elliptiques pour le chiffrement et la signature numérique.
3. Expliquer le processus de génération de clés sur les courbes elliptiques, y compris la sélection des paramètres et la création de clés publiques et privées.
4. Mettre en œuvre les opérations de chiffrement et de déchiffrement basées sur les courbes elliptiques en utilisant un langage de programmation tel que Python, pour comprendre l'application concrète de ces concepts théoriques.
5. Implémenter le processus de signature numérique sur les courbes elliptiques pour garantir l'authenticité des données échangées, en comprenant le rôle crucial de la signature dans la vérification de l'identité de l'émetteur.
6. Valider l'exactitude de l'implémentation en testant avec différents messages et en vérifiant que les résultats correspondent aux attentes théoriques, afin d'assurer la fiabilité et la sécurité des opérations cryptographiques basées sur les courbes elliptiques.

Implémentation :

Étape 1: vérification si la courbe est régulière ou bien singulière

Une courbe elliptique réelle est l'ensemble des solutions des couples $(x, y) \in R \times R$ de l'équation: $y^2 = x^3 + ax + b$ Auquel on ajoute un point spécial O, le point à l'infini. Si Le discriminant de la courbe $\Delta = 4a^3 + 27 b^2 = 0$, on dit que la courbe est singulière , c'est un cas particulier que l'on souhaite éviter (en général). Sinon, elle est régulière .

<u>Et voici le code en Python :</u>

```python
class EllipticCurve:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def is_singular(self):
        delta = 4 * self.a**3 + 27 * self.b**2
        return delta == 0

    def is_regular(self):
        return not self.is_singular()

    if curve.is_singular():
        print("The elliptic curve is singular.")
    else:
        print("The elliptic curve is regular.")
```

Étape 2: la création de dictionnaire de la courbe .

1. j'ai choisi le corp p=19 pour faire les calculs .
2. Définition des fonctions de multiplications et additions dans les courbes elliptiques

pour l'addition :

$$x_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)^2 - x_1 - x_2 \quad (mod \quad p)$$

$$y_3 = \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_1 - x_3) - y_1 \quad (mod \quad p)$$

et pour la multiplication :

$$x_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)^2 - 2x_1 \quad (mod \quad p).$$

$$y_3 = \left(\frac{3x_1^2 + a}{2y_1}\right)(x_1 - x_3) - y_1 \quad (mod \quad p).$$

<u>Et voici le code en Python :</u>

```python
# Define the point addition function
def point_addition(x1, y1, x2, y2):
    # Check if one of the points is at infinity
    if x1 == float('inf'):
        return x2, y2
    if x2 == float('inf'):
        return x1, y1

    # Check if the points are equal and handle special cases
    if x1 == x2 and (y1 != y2 or y1 == 0):
        return float('inf'), float('inf')

    # Calculate the slope of the line passing through the points
    if x1 == x2:
        # Case of point doubling
        l = ((3 * x1**2 + a) * pow(2 * y1, -1, p)) % p
    else:
        # Case of point addition
        l = ((y2 - y1) * pow(x2 - x1, -1, p)) % p

    # Calculate the new point coordinates
    x3 = (l**2 - x1 - x2) % p
    y3 = (l * (x1 - x3) - y1) % p

    return x3, y3
```

```python
    return x3, y3

# Double a point on the elliptic curve
def point_double(x, y):
    # Check if the point is at infinity
    if y == float('inf'):
        return float('inf'), float('inf')

    # Calculate the slope of the tangent line at the point
    l = ((3 * x**2 + a) * pow(2 * y, -1, p)) % p

    # Calculate the new point coordinates
    x3 = (l**2 - 2 * x) % p
    y3 = (l * (x - x3) - y) % p

    return x3, y3

# Multiply a point by a scalar
def scalar_multiply(k, x, y):
    k = k % (p - 1)  # Use modulo p-1 for scalar multiplication
    result = float('inf'), float('inf')
    while k > 0:
        if k & 1 == 1:
            result = point_addition(result[0], result[1], x, y)
        x, y = point_double(x, y)
        k >>= 1
    return result
```

- la définition de la courbe .

```python
import random
import math
import numpy as np
import matplotlib.pyplot as plt

# Génération de la courbe elliptique
def generate_elliptic_curve(a, b, mod):
    if 4 * a**3 + 27 * b**2 == 0:
        raise ValueError("Invalid parameters for elliptic curve")
    points = {}
    for x in range(mod):
        for y in range(mod):
            if (y**2 - x**3 - a*x - b) % mod == 0:
                points[(x, y)] = None
    points[(0, 0)] = None
    return points
```

- choisir un point générateur comme base .

```python

# Génération du générateur
def generate_generator(points, mod):
    G = random.choice(list(points.keys()))
    if G == (0, 0):
        return generate_generator(points, mod)
    return G
```

- construire le dictionnaire

```python

# define the dictionary

dictionary(G,p )

```

Étape 3: la Génération des clés

```python

# Fonction pour générer une paire de clés ECC
def generate_key_pair(curve, base_point):
    # Choisir un grand nombre premier aléatoire comme clé privée
    private_key = random.randint(1, curve.order - 1)

    # Calculer les coordonnées du point public en multipliant la clé privée par le point de base
    public_key = curve.scalar_multiply(private_key, base_point)

    return private_key, public_key

# Exemple d'utilisation
curve = EllipticCurve(a, b)
base_point = (5, 1)  # Point de base sur la courbe

# Générer la paire de clés pour Alice
Alice_private_key, Alice_public_key = generate_key_pair(curve, base_point)

# Générer la paire de clés pour Bob
Bob_private_key, Bob_public_key = generate_key_pair(curve, base_point)

print("Alice's private key:", Alice_private_key)
print("Alice's public key:", Alice_public_key)
print("Bob's private key:", Bob_private_key)
print("Bob's public key:", Bob_public_key)
```

Étape 4: Le chiffrement

- Convertir le message clair M en une liste de points sur la courbe elliptique.
- Générer un grand nombre entier aléatoire k.
- Ajouter le premier élément de la liste de messages chiffrés, qui est le résultat de la multiplication scalaire du point de base par k.
- Pour chaque point dans la liste de points du message, ajouter à la liste de messages chiffrés le résultat de l'addition du point avec la multiplication scalaire de la clé publique de Bob par k.

Et voici le code en Python :

```python

# Function to encrypt a plaintext message M
def encrypt_message(M, curve, base_point, Bob_public_key):
    encrypted_message = []

    # Convert the plaintext message into a list of points on the curve
    points = [curve.point_from_char(char) for char in M]

    # Generate a large random integer k
    k = random.randint(1, curve.order - 1)

    # Add the first element of the encrypted message list: P * k
    encrypted_message.append(curve.scalar_multiply(k, base_point))

    # Encrypt each point in the message points list
    for point in points:
        # Add the element L + Qb * k to the encrypted message list
        encrypted_message.append(curve.point_addition(point, curve.scalar_multiply(k, Bob_public_key)))

    return encrypted_message

encrypted_message = encrypt_message(M, curve, A, Bob_public_key)
print("Encrypted message:", encrypted_message)


```

Étape 5: Le Déchiffrement

1. La fonction decrypt_message prend le message chiffré, la courbe elliptique et la clé privée de Bob comme entrée.
2. Elle calcule le point S en multipliant le premier élément du message chiffré (chiffré avec la clé publique de Bob) par la clé privée de Bob.
3. Ensuite, elle déchiffre chaque point dans le message chiffré en soustrayant S.
4. Enfin, elle convertit la liste des points déchiffrés en texte clair, représentant le message original.

Et voici le code en Python :

```python


# Function to decrypt an encrypted message
def decrypt_message(encrypted_message, curve, Bob_private_key):
    decrypted_message = []

    # Calculate the point S as the first element of the encrypted message * Bob's private key
    S = curve.scalar_multiply(Bob_private_key, encrypted_message[0])

    # Decrypt each point in the encrypted message
    for point in encrypted_message[1:]:
        # Add the element Lc - S to the decrypted message list
        decrypted_point = curve.point_subtraction(point, S)
        decrypted_message.append(decrypted_point)

    # Convert the list of decrypted points back to plaintext
    plaintext_message = "".join([curve.char_from_point(point) for point in decrypted_message])

    return plaintext_message


decrypted_message = decrypt_message(encrypted_message, curve, Bob_private_key)
print("Decrypted message:", decrypted_message)
```

Étape 6: La signature numérique

- La fonction sign_message prend le message à signer, la courbe elliptique, le point de base et la clé privée d'Alice comme entrées.
- Elle choisit un nombre entier aléatoire k dans l'intervalle [1, n - 1].
- Elle calcule k * P pour obtenir les coordonnées (x1, y1) d'un point.
- Ensuite, elle calcule r = x1 mod n. Si r = 0, elle recommence avec un nouveau k.
- Elle hache le message avec une fonction de hachage (dans cet exemple, SHA-256).
- Elle calcule S en utilisant la formule spécifiée dans l'algorithme ECDSA.
- Si S != 0, la signature est valide et elle la renvoie sous forme de tuple (r, s).

```python
import hashlib

# Fonction pour signer un message M avec ECDSA
def sign_message(message, curve, base_point, Alice_private_key):
    # Choisir un nombre entier aléatoire k dans [1, n - 1]
    k = random.randint(1, curve.order - 1)

    while True:
        # Calculer k * P = (x1, y1)
        kP = curve.scalar_multiply(k, base_point)
        x1, _ = kP

        # Calculer r = x1 mod n
        r = x1 % curve.order

        if r == 0:
            continue

        # Calculer H(M) avec une fonction de hachage
        hashed_message = int.from_bytes(hashlib.sha256(message.encode()).digest(), byteorder='big')

        # Calculer s = (k^(-1))(H(M) + kA * r) mod n
        k_inv = pow(k, -1, curve.order)
        s = (k_inv * (hashed_message + Alice_private_key * r)) % curve.order

        if s != 0:
            break

    return r, s


signature = sign_message(message, curve, base_point, Alice_private_key)
print("Signature:", signature)
```

Étape 7: La vérification de La signature numérique

- La fonction `verify_signature` prend le message, la signature, la courbe elliptique, le point de base et la clé publique d'Alice comme entrée .
- Elle vérifie d'abord que la clé publique d'Alice n'est pas le point à l'infini et qu'elle appartient à la courbe elliptique.
- Ensuite, elle vérifie que n * QA n'est pas le point à l'infini.
- Elle vérifie que les valeurs de r et s sont dans l'intervalle [1, n - 1].
- Elle calcule w = s^(-1) (mod n) et H(M) avec une fonction de hachage.
- Elle calcule u1 = H(M) * w (mod n) et u2 = r * w (mod n).
- Elle calcule u1 * P + u2 * QA pour obtenir le point (x0, y0).
- Elle vérifie si r = v où v = x0 (mod n).
- Si r = v, la signature est considérée comme valide.

tpcrypto.py > ...

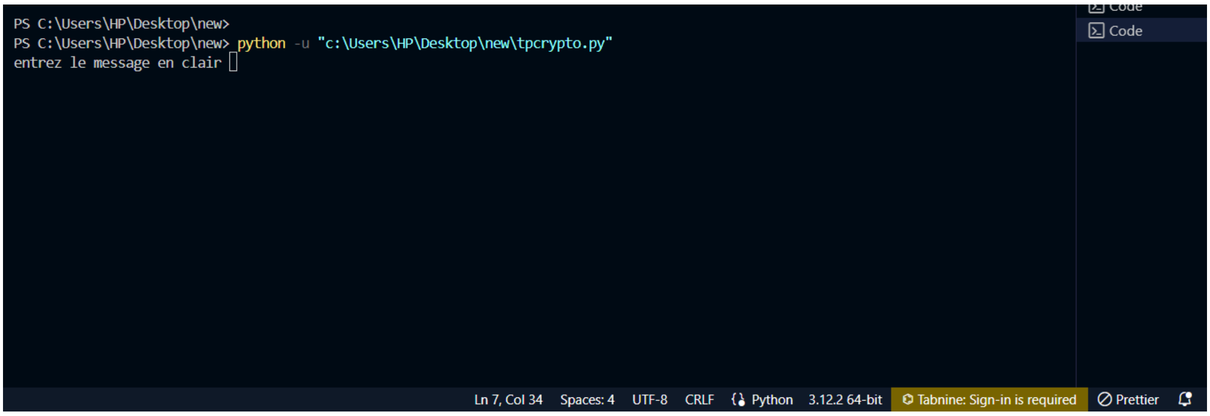
```python
# Function to verify the signature of a message M by Bob
def verify_signature(message, signature, curve, base_point, Alice_public_key):
    r, s = signature

    # Check if QA ≠ ∞ and QA ∈ E(Fp)
    if Alice_public_key == (float('inf'), float('inf')):
        return False

    # Check if n * QA = ∞
    if curve.scalar_multiply(curve.order, Alice_public_key) == (float('inf'), float('inf')):
        return False

    # Check if r and s are in the range [1, n - 1]
    if not (1 <= r < curve.order and 1 <= s < curve.order):
        return False

    # Calculate w = s^(-1) (mod n) and H(M)
    w = pow(s, -1, curve.order)
    hashed_message = int.from_bytes(hashlib.sha256(message.encode()).digest(), byteorder='big')

    # Calculate u1 = H(M) * w (mod n) and u2 = r * w (mod n)
    u1 = (hashed_message * w) % curve.order
    u2 = (r * w) % curve.order

    # Calculate u1 * P + u2 * QA
    u1P = curve.scalar_multiply(u1, base_point)
    u2QA = curve.scalar_multiply(u2, Alice_public_key)
    x0, _ = curve.point_addition(u1P[0], u1P[1], u2QA[0], u2QA[1])

    # Calculate v = x0 (mod n)
    v = x0 % curve.order

    # Check if r = v
    return r == v
is_valid = verify_signature(message, signature, curve, base_point, Alice_public_key)
print("Is the signature valid?", is_valid)
```

la vérification et validation :

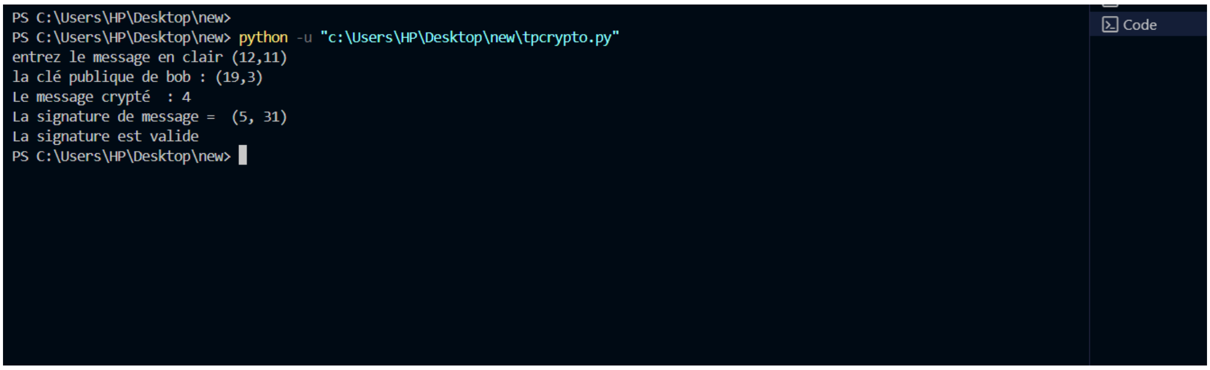
- Alice veut envoyer un message à Bob, donc elle signe le message en utilisant ECDSA.
- Bob reçoit le message et la signature, puis vérifie si la signature est valide.

1. **Signature du Message (par Alice) :**
   - Alice choisit un message à envoyer à Bob.
   - Elle signe le message en utilisant ECDSA, produisant une signature (r, s).
2. **Vérification de la Signature (par Bob) :**
   - Bob reçoit le message et la signature (r, s).
   - Il récupère la clé publique d'Alice à partir d'une source de confiance.
   - Bob vérifie si la signature est valide pour le message reçu en utilisant ECDSA.
   - Si la vérification réussit, Bob accepte le message comme authentique et non altéré.

Ce processus de validation garantit que le message reçu par Bob a effectivement été signé par Alice et n'a pas été altéré pendant la transmission.



choisir de tester avec le message m=(12,11)



Conclusion :

Dans ce compte rendu, nous avons exploré les principes fondamentaux de la cryptographie à base de courbes elliptiques (ECC), ainsi que l'algorithme ECDSA (Elliptic Curve Digital Signature Algorithm) pour la signature numérique.

Nous avons commencé par comprendre les concepts de base des courbes elliptiques et leur utilisation dans la cryptographie. Nous avons examiné le processus de génération de clés

ECC, y compris la génération de clés privées et publiques pour Alice et Bob, ainsi que l'échange sécurisé de clés publiques pour la communication sécurisée.

Ensuite, nous avons plongé dans l'algorithme ECDSA, qui est largement utilisé pour signer numériquement les messages dans les communications sécurisées. Nous avons détaillé les étapes nécessaires pour signer et vérifier un message, en mettant en évidence l'importance de la vérification de la signature pour garantir l'authenticité et l'intégrité des messages.

En utilisant des exemples concrets et des implémentations en Python, nous avons démontré comment Alice peut signer un message et comment Bob peut vérifier la signature pour s'assurer de l'authenticité du message et de son origine légitime.

En conclusion, la cryptographie à base de courbes elliptiques, en particulier l'algorithme ECDSA, joue un rôle crucial dans la sécurité des communications numériques en permettant la signature numérique des messages, assurant ainsi l'authenticité et l'intégrité des données échangées entre les parties communicantes.