Advanced Cryptography

Lab 2 : Elliptic Curves Cryptography

3A informatique majeure MSI

Notation : la note de TP prend essentiellement en compte l'efficacité et la quantité de travail réalisé pendant la séance, de sorte qu'une note de zero est attribuée à la séance en cas d'absence injustifiée et que la note baisse proportionnellement en cas de retard. Néanmoins la fonctionnalité et la qualité du code rendu sont aussi pris en compte dans la note.

Rendu attendu : les étudiants doivent développer (en python) de manière individuelle le travail demandé. Celui-ci doit être une archive au format zip contenant au minimum quatres fichiers : un fichier contenant les classes, un fichier contenant les fonctions de tests, un fichier contenant les autres fonctions (si elles existent) et un fichier readme.md. L'archive devra aussi contenir les fichiers binaires générés par openssl s'il y en a.

Le projet se prête bien à l'utilisation de classe abstraite, portant sur un groupe générique que l'on instanciera plus tard avec une loi de groupe donnée. Néanmoins le sujet ne prévoit pas cette direction qui risque d'augmenter la charge de travail. A réserver aux étudiants les plus motivés.

Consignes sur le plagiat : regarder le code source d'une autre personne, chercher du code sur internet ou sur des outils tels que chatgpt, ou travailler à plusieurs n'est pas interdit. Cela ne doit toutefois pas servir d'excuse pour du plagiat. Récupérer du code source extérieur en le modifiant à la marge (nom de variable, commentaires, etc...) est du plagiat. Au moindre doute, écrire l'origine des sources extérieures en commentaires (code récupéré de tel site internet, auprès de telle personne, ...). Le cas échéant, laisser votre propre code source en commentaire, en expliquant que celui-ci ne fonctionnant pas, vous avez du vous tourner vers un autre code.

Part 1. Elliptic curves on prime fields.

Let P-256 be the elliptic curve defined on $\mathbf{F}_p$ by the Weierstrass equation $Y^2 = X^3 + AX + B$, where p is the (prime) generalized Mersenne number 2^{256} - 2^{224} + 2^{192} + 2^{96} - 1, $A = -3$ and
`B = 0x5ac635d8aa3a93e7b3ebbd55769886bc651d06b0cc53b0f63bce3c3e27d2604b`.
The order N of the group of points is
`N = 0xffffffff00000000ffffffffffffffffbce6faada7179e84f3b9cac2fc632551`.
Let $G = = [G_x, G_y]$ be the group generator where,
`Gx = 0x6b17d1f2e12c4247f8bce6e563a440f277037d812deb33a0f4a13945d898c296`,
`Gy = 0x4fe342e2fe1a7f9b8ee7eb4a7c0f9e162bce33576b315ececbb6406837bf51f5`.
The identity element O_∞ is represented by the list [0,0].
Remark : The curve P256 is defined in FIPS 186-4 (*Digital Signature Standard (DSS)*) : `https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.186-4.pdf`

Add two arguments A and B to the class `Group` and complete the method `checkParameters`. Write a method `verify()` in the class `SubGroup`, with a point P as input, which verifies if P belongs to the curve or not (don't forget the point at the infinity), by returning `True` or `False`. This method should test if the group is based on elliptic curve.

Recover the ECDSA public key in certificates of `google` and verify that this point is in the curve P256. To do this, the public key is the hexadecimal sequence `04:08:AD:...37:B0:1A:5A:...:BC:6A`, which corresponds to the point (`08AD...37B0, 1A5A...BC6A`). Remark : according to the cryptographic standard `http://www.secg.org/sec1-v2.pdf`, the public key $A = (A_x, A_y)$ is encoded as $0x04 \parallel A_x \parallel A_y$ in hexadecimal, when the point is uncompressed (in other case, the byte $0x02$ (y is even) ou $0x03$ (y est odd) is used).

Then, transform the certificate in DER format :
```
$ openssl x509 -in google.pem -outform DER -out google.der
```
Retrieve the address of the beginning and the end of the X-coordinate of the public key in the binary file :
```
$ xxd google.der | grep 0408
000000b0: 0106 082a 8648 ce3d 0301 0703 4200 0408
$ xxd google.der | grep 37\ b0
000000d0: b406 75fa f208 167e 3293 11d2 c737 b01a
```
Then retrieve the address of the end of the Y-coordinate :
```
$ xxd google.der | grep bc\ 6a
000000f0: f225 63c8 c858 9f8a c6e2 b73e f0bc 6aa3
```
Remark : look why we can not directly search the pattern `1a5a` of the beginning of the Y-coordinate as previously.

In this case, we get the public key with :
```
GooglePublicKey = open("google.der", 'rb')
PK = GooglePublicKey.read()
Pkx = int.from_bytes(PK[0xbf:0xdf], byteorder='big')
Pky = int.from_bytes(PK[0xdf:0xff], byteorder='big')
GooglePublicKey.close()
```
Check that the public key has been entirely recovered :
```
print(hex(Pkx), hex(Pky))
```

Part 2. The law group for points on elliptic curves on prime fields.

Complete the method `law` with a new case `l = "ECConZp"`, by implementing the group law (and `checkParameters`). Remark : for modular inverse mod p in the law group, define an object of class group with `l = "Zpmultiplicative"` and use `exp` on this object. Call the method `testDiffieHellman` in a function `testLab2_part1` on the P-256 curve.

Part 3. ECDSA on P-256.

Write a method `ecdsa_sign()`, with inputs a message m and a private key s_k, which returns an ECDSA signature. m should be hashed with SHA-256 before the call to this method (use the `hashlib` library) and encoded as an integer (use for example `int(h,16)` or `int.from_bytes(h, byteorder='big')`, depending if `hexdigest()` or `digest()` is used for the hash).

For debug, test the function with the vector test (from NIST) :

https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/P256_SHA256.pdf

```
m=Example of ECDSA with P-256
h=a41a41a12a799548211c410c65d8133afde34d28bdd542e4b680cf2899c8a8c4
sk=c477f9f65c22cce20657faa5b2d1d8122336f851a508a1ed04e479c34985bf96
k=7a1a7e52797fc8caaa435d2a4dace39158504bf204fbe19f14dbb427faee50ae
t=2b42f576d07f4165ff65d1f3b1500f81e44c316f1f0b3ef57325b69aca46104f
s=dc42c2122d6392cd3e3a993a89502a8198c1886fe69d262c4b329bdb6b63faf1
```

Write a method `ecdsa_verif()`, with inputs the message m, the signature and the public key, which verify if the signature is correct (m should be hashed and encoded as an integer as for `ecdsa_sign()`).

For debug, we note $t_1 = hs^{-1} \bmod N$, $t2 = ts^{-1} \bmod N$, $Q_1 = t_1 P$ and $Q_2 = t_2 p_k$ where p_k is the public key, computed from s_k :

```
t1=b807bf3281dd13849958f444fd9aea808d074c2c48ee8382f6c47a435389a17e
t2=1777f73443a4d68c23d1fc4cb5f8b7f2554578ee87f04c253df44efd181c184c
Q1.x=9e2c4384537e1872fb420057aac0f0afd3b44128eb8ad091a58a0adc342989f8
Q2.x=f825bc60b347d1b3bde9f2c8d92ca3d55cd2f5e325c7d7cc50ed0452c68d82fe
```

Call the methods `ecdsa_sign()` and `ecdsa_verif()` in `testLab2_part2` with the message `Example of ECDSA with P-256`.

Generate the public key and the private key of Alice for ECDSA :

```
$ openssl ecparam -name prime256v1 -genkey -outform DER -out ecdhkeyAlice.der
$ openssl ec -inform DER -in ecdhkeyAlice.der -text -noout
```

Recover the private and the public key of Alice as for the google certificate, sign `Lab2.pdf` with the private key and verify the signature with the public key.