Advanced Cryptography

Lab 3 : 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. Certificate and ECDSA.

Recover the certificate of `https://fr.wikipedia.org` and verify it with your signature algorithm and the public key of the certificate authority. To do this, display the certificate and transform it into the DER format :

```
$ openssl x509 -in wikipedia-org.pem -text -noout
$ openssl x509 -in wikipedia-org.pem -outform DER -out wikipedia.der
```

Print the first line of the binary certificate and check the compliance of the four first bytes with the length of the file (the four first bytes `3082 065f` means a *sequence of 0x65f = 1631 bytes*) :

```
$ xxd wikipedia.der | more -1
00000000: 3082 065f 3082 05e5 a003 0201 0202 1206
$ wc wikipedia.der
   7   40 1635 wikipedia.der
```

The certificate is composed of two main parts : the first one is the certificate without signature and the second one is composed by a description of the signature algorithm and the signature itself. Thus, the second block of four bytes `3082 05e5` means a *sequence of 0x5e5 = 1509 bytes*, corresponding to the certificate without signature. Thus the certificate which should be hashed before signature begins at the 4-th octet and has a length of 0x5e5+4 = 0x5e9 = 1513 bytes. Consequently this hash value is :

```
cat wikipedia.der | tail -c +5 | head -c 1513 | openssl dgst -sha384
01c61c9f693846678ce029fa62663baed9cee   ... 4ea9dd36a088742789d40a
```

The second part is composed by the description of the signature algorithm, defined by the byte sequence `30 0a06 082a 8648 ce3d 0403 03` which means *ecdsa-with-sha384*, as detailed in RFC 7427. Then we have the signature, more precisely two integers s and t described by `0230 s 0231 00 t` (because s = `0x0ffd24 ... 66fb` is an integer of 0x30 = 48 bytes and t = `0x0086ea..a8fc` is an integer of 0x31 = 49 bytes with a null byte at the beginning.

We can print this second part (the first part ends at byte 0x5ed) :

```
$ xxd wikipedia.der | grep 5e0 -A8
000005e0: 5657 a081 dddc cd6c 61d9 463c ad30 0a06
000005f0: 082a 8648 ce3d 0403 0303 6800 3065 0230
00000600: 0ffd 24ac a5b7 3920 177e 1f41 eb0c f851
00000610: f3a0 e606 a76d 9a3f 0501 a4af 01d3 bf11
00000620: 6afb 0366 70d3 682e 3ec3 d2a0 3e8c 66fb
00000630: 0231 0086 ea19 1b11 b726 af5d a378 9283
00000640: 1a2b f83e 386d f74d 01cd cb3d 63c9 b886
00000650: d3c4 deb5 87d0 dd32 6d87 568f 757b 194b
00000660: fba8 fc
```

Consequently, write a function `testLab3_part1` where you recover the certificate without signature, hash it and compare the result with the previous hash value, recover s and t and recover the public key from the certificate authority. Finally, verify the signature, using the previous lab (recover P-384 parameters from FIPS 186-4, appendix D).

Part 2. Elliptic Cryptography on Binary Fields (B-163).

The elliptic curve B163 defined in FIPS 186-4 is described as follows. Remark : this curve has been deprecated in TLS 1.3 with 22 other curves (RFC 8422).

The irreductible polynomial used for $\mathbf{F}_{2^{163}}$ is $x^{163}+x^7+x^6+x^3+1$. This curve is defined on $\mathbf{F}_{2^{163}}$ by the equation $Y^2+XY = X^3+X^2+B$ and the base point is $G = (G_X, G_Y)$, where the integer representations of $B, G_X, G_Y \in \mathbf{F}_{2^{163}}$ are :
B = 2982236234343851336267446656627785008148015875581
G_X = 5759917430716753942228907521556834309477856722486
G_Y = 1216722771297916786238928618659324865903148082417
The order N of this curve is `40000000000000000000292fe77e70c12a4234c33`.

Complete the law group if `l = "ECC_F2^n"` and call `Verify` with $[G_x, G_Y]$ and `testDiffieHellman()` using the curve B-163. Call the methods `ecdsa_sign()` and `ecdsa_verif()` on the message `"Example of ECDSA with B-163"`.

Vector tests for signature using the B163 curve are given here (see also below) :
`https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/ECDSA_Characteristic2.pdf`

```
m = Example of ECDSA with B-163
h = 728d59bbe028509dd5d2ce480f458e2925232ac3 (SHA-1 is used)
d = 348d138c2de9447bd288feed177222ee377fb7bea (private key)
Q.x = 66b015c0b72b0f81b1ecba6f58e7545d94744644c
Q.y = ba6d4d62419155b186a29784f4aa4b8e8e1e7f76 (public key)
k = 8ed0f93f7d492bb3991847d0e96f9cc3947259aa
K.x = 760938a97d88b30fdfb2cce1a4c59783ad0ed8fde
inv(K.x) = 36dc66491684211373aaa8bd16024dd0a12a8ff11
t = 360938a97d88b30fdfb2a3b1bd4726c282cca43ab
s = 19a7b5043d93a13d714b4717fc0698e6791cf7f7c

s^{-1} mod N = 35417609847921e6d0e2691d924335adf62c1b6b2
hs^{-1} mod N = 3c4099db241a80c807e02d81be10955b2eb2e5d8c
ts^{-1} mod N = 160ccedabb7e3e767a604d8b042a65751708cc262
R.x = 760938a97d88b30fdfb2cce1a4c59783ad0ed8fde
```

Generate a pair of private key/public key for ECDSA with B-163 and verify that the public key has been correctly generated from the private key and that the public key is on the curve B-163 :

```
$ openssl ecparam -outform DER -out b163key.der -name sect163r2 -genkey
$ openssl ec -inform DER -in b163key.der -text -noout
$ xxd b163key.der
```

Part 3. X25519 curve.

The Curve25519 is defined by the Weierstrass equation $Y^2 = X^3 + AX^2 + X$ on $\mathbf{F}_p$ where p is the prime number $2^{255} - 19$ and $A = 486662$. The order of the group of points is $N = 2^{252} + $ 0x14def9dea2f79cd65812631a5cf5d3ed. Let $G = [G_x, G_Y]$ be the generator of the group, where $G_x = 9$ and

`Gy = 0x20ae19a1b8a086b4e01edd2c7748d14c923d4d7e6d7c61b229e9c5a27eced3d9`.
The identity element is $0_\infty = [0, 1]$.

Complete the law group if `l = "X25519"` and call the methods `Verify` with G and `testDiffieHellman()`. Verify that the inverse of G is $(G_x, -G_y)$.

Generate keys of Alice and Bob for Diffie-Hellman :

```
$ openssl genpkey -algorithm x25519 -out keyAlicex25519.pem
$ openssl genpkey -algorithm x25519 -out keyBobx25519.pem
$ openssl pkey -in keyAlicex25519.pem -text
$ openssl pkey -in keyBobx25519.pem -text
$ openssl pkey -in keyAlicex25519.pem -pubout -out pubkeyAlicex25519.pem
$ openssl pkey -in keyBobx25519.pem -pubout -out pubkeyBobx25519.pem
$ openssl pkey -pubin -in pubkeyAlicex25519.pem -text -noout
$ openssl pkey -pubin -in pubkeyBobx25519.pem -text -noout
$ openssl pkeyutl -derive -inkey keyAlicex25519.pem -peerkey pubkeyBobx25519.pem
  -out X25519key1.bin
$ openssl pkeyutl -derive -inkey keyBobx25519.pem -peerkey pubkeyAlicex25519.pem
 -out X25519key2.bin
```

Verify that the public key of Alice (resp. Bob) are on the X25519 curve (Euler criteria) and if it corresponds to the multiplication of the private key and G (use reverse_byte and the mask, see the course and below). Modify the method `Diffie-Hellman` and call it using the following description :
Let a and b be the private keys, $A = s(a)G$, $B = s(b)G$ where s is defined by `s(x) = reverse_bytes_25519(x) & ((1 << 255) - 8) | (1 << 254)`. The public keys are `reverse_bytes_25519(A[0]))` and `reverse_bytes_25519(B[0]))`.
Finally, the shared key is :
`reverse_bytes_25519((s(a)B)[0])` $=$ `reverse_bytes_25519((s(b)A)[0])`.

RSA key generation with backdoor (alternative to parts 2 and 3).

The objective of this part is a black-box generation of RSA keys such that :

1. the output (the public keys $(e,\ N)$ and the private keys (p, q, d)) should be indistinguishable from a standard key generation.,
2. the generation algorithm can be processed several times without reveal any problem (a generator with fixed values for primes numbers p or q does not accomplish this),
3. the attacker should be able to recover the private keys from the knowledge of the public keys and the backdoor.

Remark : these conditions mean that statistical tests on many outputs should not reveal any statistical differences with standard generation.

Let k be the security parameter. The primes numbers p and q should have a size of k bits, providing a modulus N of size $2k$ bits. if possible, we use $k = 512$ in this lab, but use $k = 128$ in a first time.

This lab uses the `Cryptodome` library. If `import Cryptodome` provides an error, use `pip3 install cryptodomex`.

Key generation for the attacker.

The attacker generates in a first time one RSA keys pair : two primes numbers p_{att} and q_{att}, each of binary size $k/2$, the k-bits integer $N_{att} = p_{att}q_{att}$, an integer e_{att}, prime with $\phi(N_{att})$, and d_{att} the inverse of e_{att} modulo $\phi(N_{att})$. Write a function `KeyGenerationAttacker(k)` which returns $p_{att}, q_{att}, N_{att}, e_{att}$ and d_{att}.

This function uses `nb.getPrime(k)` for the prime generation of an integer of exactly k bits (with `from Cryptodome.Util import number as nb`). Remark : for the random generation, `random.getrandbits(k)` generates an atmost k-bits integer. Verify during the generation that the size of N_{att} is k with the function `nb.size()`, because the product of two k bits integers is an integer of size at least $2k - 1$, but is not necessary a $2k$-bits integer.

First RSA keys generation with backdoor.

A backdoor in the RSA keys generation can be installed as follows :

1. Let p and q be two prime numbers of k bits, with $p < N_{att}$, and $N = pq$.
2. Let $e \leftarrow p^{e_{att}} \bmod N_{att}$ with $\gcd(e, \phi(N)) = 1$ (if not, return to step 1) and $d \leftarrow e^{-1} \bmod \phi(N)$.
3. Return the public key (e, N) and the private key (p, q, d).

Write a function `backdoorGeneration1(N_att, e_att, k)` which returns the previous RSA key.

In this case, the attacker recovers p by computing $e^{d_{att}} \bmod N_{att}$ because $e^{d_{att}} \bmod N_{att} = p^{e_{att}d_{att}} \bmod N_{att} = p^{1+k\phi(N_{att})} \bmod N_{att} = p \bmod N_{att}$, using the Euler Theorem. Write a function `attack1(e, N, d_att, N_att)` which returns `True` if we have the equality `N % pow(e, d_att, N_att) == 0` and `False` otherwise. This attack is considered as a test function for `backdoorGeneration1`.

Second RSA keys generation with backdoor.

The first strategy is limited because the public exponent e is not always randomly generated by the user. In fact, the default value $e = 2^{16} + 1 = 65537$ is generally used. Consequently, it is not a good idea to hide the backdoor into e. The modulus N would be a better choice.

A backdoor in the RSA keys generation can be installed in N as follows :

1. Let p be a prime number of (exactly) k bits with $p < N_{att}$, r_2 be a random integer of (exactly) k bits and $r_1 = p^{e_{att}} \bmod N_{att}$.
2. Let q be the quotient of the euclidean division between the integer $r_1 \parallel r_2$ ($\parallel$ means the concatenation) and p. If q is not prime return to step 1.
3. Let $N = pq$, e is a public exponent (for example 65537), prime with $\phi(N)$ and d the inverse of $e \bmod \phi(N)$.
4. Return the public key (e, N) and the private key (p, q, d).

Write a function `backdoorGeneration1(N_att, e_att, k)` which returns the previous RSA key. You can use `getrandbits(k)` for r_2 if the concatenation in the next step is realized with a shift of k bits, regardless of the real length of r_2. For primality test, use `nb.isPrime`.

The attacker recovers p by computing in a first time the k upper orders bits of N in order to recover r_1 ($r_1 \parallel r_2$ is an integer of $2k$ bits) and retrieves p by computing $r_1^{d_{att}} \bmod N_{att} = (p^{e_{att}})^{d_{att}} = p \bmod N_{att}$, using Euler Theorem. Remark : the k upper orders bits of N can be easily achieved with a shift. Write a function `attack2(N, k, d_att, N_att)` which computes r_1 and returns `True` if the following equality is correct `N % pow(r1, d_att, N_att) == 0`, and `False` otherwise.

Remark : by construction $r_1 \parallel r_2 = pq + R$, where R is the reminder of the euclidean division, of size atmost k. It is not impossible than the $k+1$-th bit of pq be 0, whereas the $k+1$-th bit of $r_1 \parallel r_2$ be 1 (the other bits are necessary equal). Consequently, if the value of p, computed in the attack, does not divide N, add 1 to the k upper orders bits of N and recover the correct value of p.

Example : k =4, $p = 11$, $r_1 = r_2 = 9$, $r_1 \parallel r_2 = 153 = 10011001$ in binary. So $r_1 \parallel r_2 = p \times 13 + 10$, with $q = 13$, $R = 10$ and $N = pq = 143 = 10001111$ in binary. Thus the 5-th bit of N is 0, whereas the 5-th bit of $r_1 \parallel r_2$ is 1.

Third RSA keys generation with backdoor.

The previous algorithm has a default because p is a k-bits random prime integer verifying $p < N_{att}$. Consequently p is not exactly a k-bits random prime integer and a statistical test is able to reveal it. We modify the previous algorithm as follows in order to propose the last algorithm for the backdoor generation :

In the step 1, we generate some k-bits random prime integers p until $r = AES(p)$ be lower than N_{att} (the encryption uses a fixed key of 16 bytes, known by the attacker, which can be hardcoded in the file). Then we compute $r_1 = r^{e_{att}} \bmod N_{att}$. During the attack, we retrieve r by computing $r_1^{d_{att}} \bmod N_{att}$, and we decrypt r into p. Write a function `backdoorGeneration3(N_att, e_att, k)` which returns the previous RSA key. and a function `attack3(N, k, d_att, N_att)` which returns `True` if p is recovered and `False` otherwise.

We only use AES as pseudo random function on $\{0,1\}^k$. Use the `Cryptodome` library for AES Finally, for the encryption, more specially for the conversion byte/integer, use `nb.long_to_bytes()` and `nb.bytes_to_long` for the conversion of an integer into a byte (and conversely).

Use the following data for debug : $k = 128$.

P_{att} = 0xd16aae70e26101f1
Q_{att} = 0xa95d0f6887fdc0df
N_{att} = 0x8a8b8d84a74298d7cfc668eaf62270ef
e_{att} = 0x442165b9174609bdc4aaf5a7dbc1b21b
d_{att} = 0x61117b50aa37371dd96cb600cb84d33

First attack :
p = 0x8179b0c42e8e7b383c3b18b3d7f7bd43
q = 0xa3f0cc6e8beb1c3c9247281053494309

N = 0x52ea4507837aa36599b03ae2f6b2eccf0babddce8ab0e6efd87c240a1f59305b
e = 0x44113d137b3979fdaddf9673e8440257
d = 0x2016352296a4b4580790a78d87e5ff23038bbc373ca956acf74fc8b00e50aa87

Second attack :
p = 0x81a48990e7a1c6655af016f2c4e7f46d
q = 0x8b8fc3d4413178ac504e3e8068a695bd
r_2 = 0x4ffab4ed9352a7251b19e74284cc8d9e
N = 0x46ad24c657c55b0f6a5576d311203f11ce878b74f2f609208c68c40b2730e579
e = 0x10001
d = 0x21665399b402dd2ca4c8acb5c44323ea8c2205933aa8442ad1f7d679d58c0771

Third attack :
p = 0xe195aa3b367eb77678958ab5579bfeff
q = 0x5741eca8d2214b6998f981e5f9387e89
r = 0x5a30f8a811b877c3e2ba6927f352b650
AES Key = "KKKKKKKKKKKKKKKK"
r_2 = 0xd7aad4a1d8cb8f27081b701e19a9a23e
N = 0x4ce3f46514de5001d6bdeb36f2f5587238e000e663f0e977443efa6b78c4f877
e = 0x10001
d = 0x46dc2985e236830b3ec8e5f54cf0d861c342fdd7502a7d67c819a4139a4b2031