Advanced Cryptography

Lab 5 : Sidechannel attacks

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.

Introduction to Numpy (with `import numpy as np`).

`tab = np.zeros(n)` returns an array with n elements, initialized to zero (type `float` by default) and `mat = np.zeros((n, m))` returns a matrix with n rows and m columns (initialized to zero). The element at the row i and column j is accessed with `mat[i][j]` or `mat[i,j]`. The row i of `mat` is obtained with `mat[i,:]` as an array witn m elements and the column j of `mat` is obtained with `mat[:,j]` as an array witn n elements.

The function `np.mean(tab)` returns the mean of the elements of `tab`. The function `np.mean(mat, axis = 0)` returns an array (of size m) containing the mean of each column of the matrix `mat`, whereas `np.mean(mat, axis = 1)` returns an array (of size n) containing the mean of each row. Addition/subtraction/product/division, element by element, of two arrays (same size) is directly obtained with `tab1+tab2`, `tab1-tab2`, `tab1*tab2` and `tab1/tab2`. You can also replace one array by a constant `e` : `tab+e`, `tab-e`, `tab*e` et `tab/e`.

The function `abs(tab)` returns an array with the absolute value of each element, `max(tab)` returns the greatest element of `tab` and `max(mat)` returns an array of size n with the greatest element of each row (of the matrix `mat` with n rows). The function `np.argmax(tab)` returns the index of the greatest element of `tab`. The function `np.sqrt(tab)` returns an array with the square root of each element.

Part 1. SPA Attack (one hour).

Realize the challenge *Double and Broken* from the CryptoHack platform (end of the section *Elliptic Curves*, subsection *Side Channel*) : *We've managed to get power readings from scalar multiplication on the Secp256k1 curve. Can you recover the private key from the data dump from 50 repeated multiplications ?* Registration on the website is not mandatory for this lab.

Remark : double-and-add is the name of the square-and-multiply algorithm in the case of elliptic curves.

Part 2. CPA Attack (two hours).

Data for the lab. The first file `plaintext.npy` contains N plaintexts, where each plaintext is represented by an array of 16 bytes (128 bits). These plaintext have been encrypted with AES using the same key (the ciphertexts are not given). The second file `traces.npy` contains $N = 50$ power traces, with $N' = 9996$ points for each trace. These traces have been acquired during the encryption. The objective of this lab is to recover this key by implementing a CPA attack (*Correlation Power Analysis*) on the first round of the AES (after *SubBytes*).

Data of the two previous files are recovered from the `load` function of `numpy`, using `traces = np.load(filename1)` and `plaintext = np.load(filename2)`. Verify that `len(traces)` returns 50 and `len(traces[0])` returns 9996.

You can visualize the first trace using (delete these three lines after use) :

```
import matplotlib.pyplot as plt
plt.plot(traces[0])
plt.show()
```

First round of the AES. The secret key $K = (K_0, \ldots, K_{15})$ is directly xored with the plaintext $P = (P_0, \ldots, P_{15})$ before SubBytes. During SubBytes, each byte of the result is sent into the S-box which maps 1 byte into 1 byte. This S-box is represented by an array `sbox` (the input i is an integer between 0 and 255, and the corresponding output is `sbox[i]`). The key K is recovered byte by byte (independently), so the ith byte K_i is recovered, by considering the corresponding byte P_i of the plaintext. The good hypothesis on K_i (among $2^8 = 256$ possibilities) provides the greatest correlation between the Hamming weight of the output of the S-box and the corresponding power trace.

The Hamming weight can be computed with a function, but in our case we know the size of the data (one byte, corresponding to the output of the S-box). Therefore, it is more appropriate to precompute an array HW of size 256, such that $HW[i]$ be the Hamming weight of i. Moreover, the Hamming weight of an integer n can be computed by `bin(n).count('1')`.

CPA attack. The attack is performed on each byte K_j of the key K (j is between 0 and 15, but you can begin by $j = 0$, and add a loop in a second time). Let $W'_{i,l}$ be the l-th point of the i-th power trace (consequently i is between 0 and $N-1$, whereas l is between 0 and $N'-1$). Compute $\overline{W'_l}$, the mean (on i) of these points, represented by an array of size N', which is independent from the hypothesis on the key. Remark : look the good numpy function for this.

For each hypothesis k on the byte K_j (among the 256 possibilities), we compute the Pearson coefficients between the power traces and the Hamming weight of the output of the S-box. More precisely, the max of these coefficients (called `maxcpa`) should be computed. Thus, `maxcpa` is an array of size 256 and the good hypothesis on K_j is the index of `maxcpa` with the greatest value (look at the good numpy function for this). More precisely, for each hypothesis k on K_j :

1. For each trace (indexed by i), we compute the Hamming weights $W_{i,k}$ of the output of the corresponding S-box, stored in an array of size N (these weights depend on the corresponding plaintext). We also compute the mean $\overline{W_k}$ of these weights.
2. We compute the following Pearson's coefficients :

$$r_{k,l} = \frac{\sum_{i=0}^{N-1}(W_{i,k} - \overline{W_k})(W'_{i,l} - \overline{W'_l})}{\sqrt{\sum_{i=0}^{N-1}(W_{i,k} - \overline{W_k})^2}\sqrt{\sum_{i=0}^{N-1}(W'_{i,l} - \overline{W'_l})^2}}.$$

   More precisely, for each trace (indexed by i) :
   (a) $hdiff_i \leftarrow W_{i,k} - \overline{W_k}$ (float)
   (b) $tdiff_i \leftarrow W'_{i,l} - \overline{W'_l}$ (array of size N')
   (c) $sum_1 \leftarrow sum_1 + hdiff_i * tdiff_i$ (array of size N')
   (d) $sum_2 \leftarrow sum_2 + tdiff_i * tdiff_i$ (array of size N')
   (e) $sum_3 \leftarrow sum_3 + hdiff_i * hdiff_i$ (float)

   The coefficient $r_{k,l}$ is represented as an array of 256 elements, where each element is an array of N' points using the multiplication/division/sqrt functions on arrays in numpy.

3. Compute an array `maxcpa` of size 256 where `maxcpa[k]` is the greatest value among the N' values $|r_{k,l}|$ (don't forget the absolute value).

The good hypothesis on K_j is the index of the greatest value of `maxcpa` (look the good numpy function for this).

Solution : the good key is `98a90ac6a2bc26493c3be04b113555d4`.