SCA_correlation_DPA

Note:
np array with shape (x,y), so x rows y columns
4444
5555
6666
7777

```
arr[0,:] => gives the 0th row => 4444
arr[:,0] => gives the 0th column => 4567
```

First we implement the Pearson coefficient function.

$$r_{xy} = \frac{n\sum x_i y_i - \sum x_i \sum y_i}{\sqrt{n\sum x_i^2 - \left(\sum x_i\right)^2}\sqrt{n\sum y_i^2 - \left(\sum y_i\right)^2}},$$

```
def pearson_corr(x, y):
    """
    x: raw traces, as an np.array of shape (nb_traces, nb_samples)
    y: model, as an np.array of shape (nb_traces, nb_samples)
    return: the pearson coefficient of each sample as a np.array of
shape (1, nb_samples)
    """
    #TODO:
    x = np.asarray(x)
    n = x.shape[0]
    nb_samples = x.shape[1]
    corr_coeffs = np.zeros(nb_samples)

    for j in range(nb_samples):
        x_j = x[:,j]
        y_j = y[:,j]
        x_sum = np.sum(x_j)
        y_sum = np.sum(y_j)
        x2_sum = np.sum(x_j**2)
        y2_sum = np.sum(y_j**2)
```

```
        numerator = n * np.matmul(np.transpose(x_j), y_j) - x_sum *
y_sum
        denominator = np.sqrt((n * x2_sum - x_sum ** 2)

* (n * y2_sum - y_sum ** 2))
        corr_coeffs[j] = numerator / denominator

    return corr_coeffs
```

In this exercise the goal is to use an a-priori selected leakage model (the Hamming weight of the processed data) and we assume that the power consumption of the target device is linearly correlated with it.

Pearson coefficient helps us to measure how well the leakages can be predicted by a linear function of the model values. The underlying idea is that the subkey hypothesis that predicts the leakages best should be the correct one.

We are going to find the Hamming weight of the intermediate value, then we are going to extend it to become an array since we are dealing with traces, and Hamming weight is an integer. So to find the Pearson correlation we need both
of them to be arrays.

Then as before, we are going to do this for every key candidate and then order the best ones.

In the exercise we are asked to perform this attack for the value before the S-box and after the S-box which are implemented in the following code:

```
def HW(v):
    c = 0
    while v:
        c += 1
        v &= v - 1

    return c
def cpa_byte_out_sbox(index, pts, traces):
    """
    index: index of the byte on which to perform the attack.
    pts: the plaintext of each encryption performed.
    traces: the power measurements performed for each encryption.
```

```
    return: an np.array with the key bytes, from highest probable to
less probable
    when performing the attack targeting the input of the sbox
    """
    # TODO
    nb_traces = traces.shape[0]
    nb_samples = traces.shape[1]
    coeffs = []
    for key_guess in range(256):
        weights = []
        for i in range(nb_traces):
            x_i = pts[i,:][index] # get the byte of the plaintext
            v_i_star = sbox[x_i ^ key_guess] # apply the prediction
(remove s-box for the in_sbox version)
            hw = HW(v_i_star)
            weights.append(np.full(nb_samples,hw))
        p_corr = pearson_corr(weights,traces)
        coeffs.append(np.max(abs(p_corr)))

    return np.array(sorted(range(len(coeffs)), key=lambda i: coeffs[i],
reverse=True))
```

Why this works

We have traces with the following dimensions: (nb_traces, nb_samples)
For each key guess we calculate the intermediate value, then we take its hamming weight and extend it to an array to get the dimensions (nb_samples,)

HW matrix for a key guess then has the following form:

```
[444...4 ; 222...2 ; .... ; 111...1 ] => (nb_traces, nb_samples)
```

Then Pearson correlation coefficient takes the first time sample of each trace, then takes the first HW of each row. In other words, correlates the columns.

Then each time sample ends up having a Pearson correlation coefficient associated to it. If the key guess is correct we expect that correlation coefficient is high, since our model for the leakage is assumed to hold.

Then you do this for each key guess, and then you rank them.

Attacking intermediate value before the S-box and after the S-box

When attacking the intermediate value before the S-box, the attacks performs worse than attacking after the S-box.

XOR operation => linear operation
S-Box => non-linear operation

Before the S-box => linear model on a linear operation
After the S-box => linear model on non-linear operation

Due to non-linearity the model overlaps less, since there could be different x y values that lead to the same HW on the XOR result.

for dpa attacks sboxes are problem
non-linearity allows larger attack surface