Advanced Topics In Cybersecurity 2

Task 1

Federico Dittaro, 12345773

Declaration

I declare that I am the sole author of this submission. I take full responsibility for all content. I confirm that I have either not used AI based tools at all, or, that I have only used them for the purposes of checking grammar and spelling.

1 Original script behavior

The original script performs a DPA attack on AES. The demo is based on a differential attack, using a Hamming weight leakage model, and correlation as a distinguisher. This attack to succeed requires 150 traces. Below the picture represents the succeed of the attack with 150 traces.

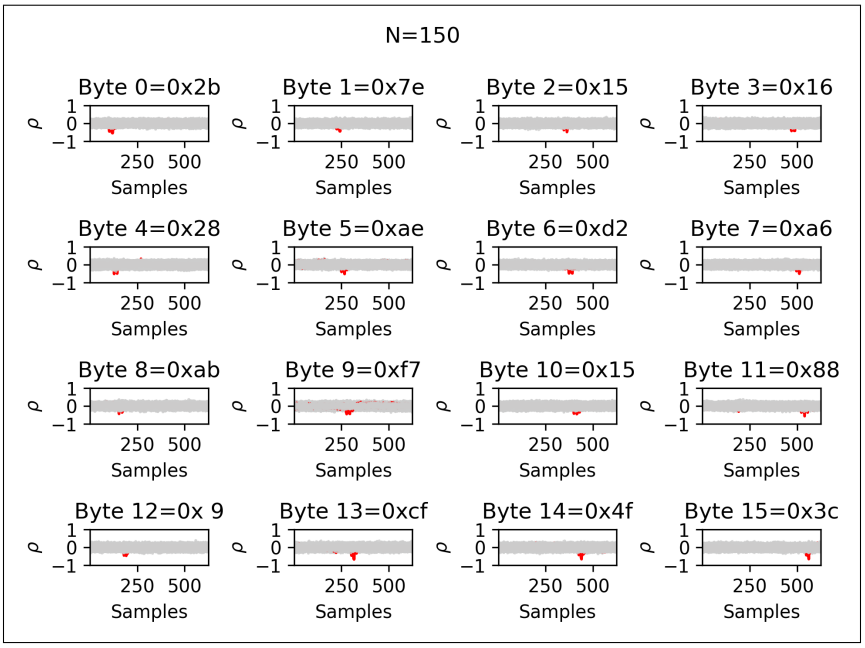

Figure 1: Result of the attack using 150 traces

2 Single Bit leakage model

In this attack instead of considering the whole byte (as in the original code), we consider just a single bit. To implement this idea I have created a new function called *BIT_LEAKAGE* which is reported in the box below:

```
def BIT_LEAKAGE(self, X):
        y = np.zeros(len(X)).astype(int)
        for i in range(len(X)):
            y[i] = (X[i] & 2**1) >> 1
        return y
```

Listing 1: BIT_LEAKAGE implementation

In this specific case we are considering the bit number 1 but for the attack the code has been run for eight times (one for each bit). The question now requires to run the script with both correlation and difference of means distinguishers. This will be explained in the sections 2.1 and 2.2

2.1 Correlation distinguisher

To run the script using the correlation as the distinguisher we don't need to modify a lot of the original code. In fact, it is sufficient to replace in the **Initialize** function the line

HWguess[byteno,kg]=AESAttack().HW(Y)

with the line

HWguess[byteno,kg]=AESAttack().BIT_LEAKAGE(Y)

the rest of the script remains unchanged.

2.1.1 Results

Since that now we are dealing with a single bit instead of the whole byte, 150 traces are not sufficient anymore. To succeed the attack we need 1000 traces. By running the script eight times (one for each bit), the only bit that succeed to retrieve all the keys is the bit number 5. Below it is reported the related figure.

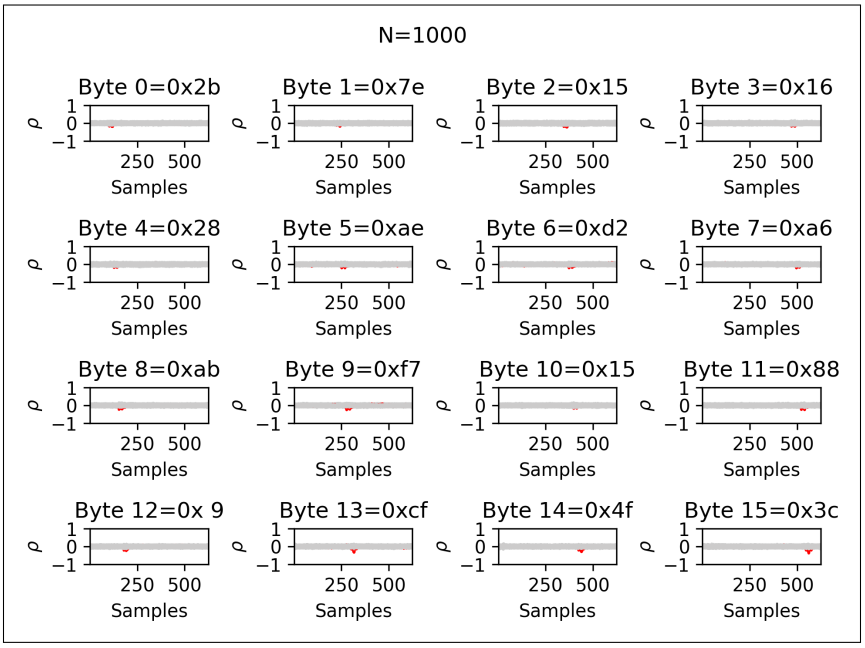

Figure 2: Result of the attack using 1000 traces and bit 5

2.2 Difference of means distinguisher

In the difference of means distinguisher the power traces are divided into two groups based on a bit of the key being 0 or 1. Then for each group, the mean power consumption at each time point is computed and are identified the time points where the difference trace is significant, suggesting a strong correlation between the power consumption and the hypothesized secret data.
In order to use the difference of means distinguisher instead of the correlation one, the code now should use again the *BIT_LEAKAGE* function as in the previous case and the following two new functions:

- **differenceOfMeans**: implements the Difference of Means distinguisher. For each time point i, the function divides the traces into two groups: *group1* corresponds to the power consumption values at time i for all traces where dom_indicator = 0 while *group2* corresponds to the power consumption values at time i for all traces where dom_indicator = 1. Then it computes the absolute difference between the two means and returns the *diff* array, where each element represents the absolute difference of means for a specific time point. The box below shows the related code:

```
    def differenceOfMeans(self, dom_indicator, traces):
        diff = np.zeros(trs.number_of_samples)
        for i in range(trs.number_of_samples):
            group1 = traces[dom_indicator==0, i]
            group2 = traces[dom_indicator==1, i]
            diff[i] = abs(np.mean(group1) - np.mean(group2))
        return diff
```

Listing 2: differenceOfMeans implementation

- **diffMeansAttack**: implements the actual attack using the Difference of Means (DoM) distinguisher. It analyzes power consumption traces to recover a single byte of a cryptographic key by evaluating all 256 possible key guesses. It calls on every iteration the distinguisher and keeps track of the maximum value of the difference of means for each guess. The box below shows the related code:

```
    def diffMeansAttack(self, trs, ax, byteno, Nm):
        ax.clear()
        maxkg = 0
        maxdiff_k = 0
        for kg in range(256):
            dom_indicator = HWguess[byteno, kg]
            diff = AESAttack().differenceOfMeans(dom_indicator[0:Nm
               ], trs.traces[0:Nm, :])
            maxdiff = np.max(diff)
            if maxdiff > maxdiff_k:
                maxkg = kg
                maxdiff_k = maxdiff
            if kg == key[byteno]:
                ax.plot(diff, 'r-', alpha=1)
            else:
                ax.plot(diff, color=(0.8, 0.8, 0.8), alpha=0.8)
        ax.set_xlim([1, trs.number_of_samples])
        ax.set_ylim([0, np.max(diff) * 1.1])
        ax.title.set_text('Byte {0}=0x{1:2x}'.format(byteno, maxkg)
           )
        ax.set_xlabel('Samples')
        ax.set_ylabel('DOM')
        return maxkg
```

Listing 3: diffMeansAttack implementation

To run the code the **Initialize** function remains unchanged from the previous case while in the **main** it is necessary to replace the line

AESAttack().corrAttack(trs, ax[byteno], byteno, Nm*50)

with the line

AESAttack().diffMeansAttack(trs, ax[byteno], byteno, Nm*50)

the rest of the script remains unchanged.

2.2.1 Results

By running the script eight times (one for each bit), none of them is capable of retrieving the keys. This results shows that the correlation distinguisher outperforms the difference of means distinguisher. This happens because the correlation one utilizes the full granularity of multivalued hypotheses and it is more robust against noise and outliers since it accounts for both means and variances. The only point in favor of the DoM distinguisher is that it is computationally faster but can only work in limited and controlled scenarios. As an example, the picture below shows the output of the attack using the Byte number 5.

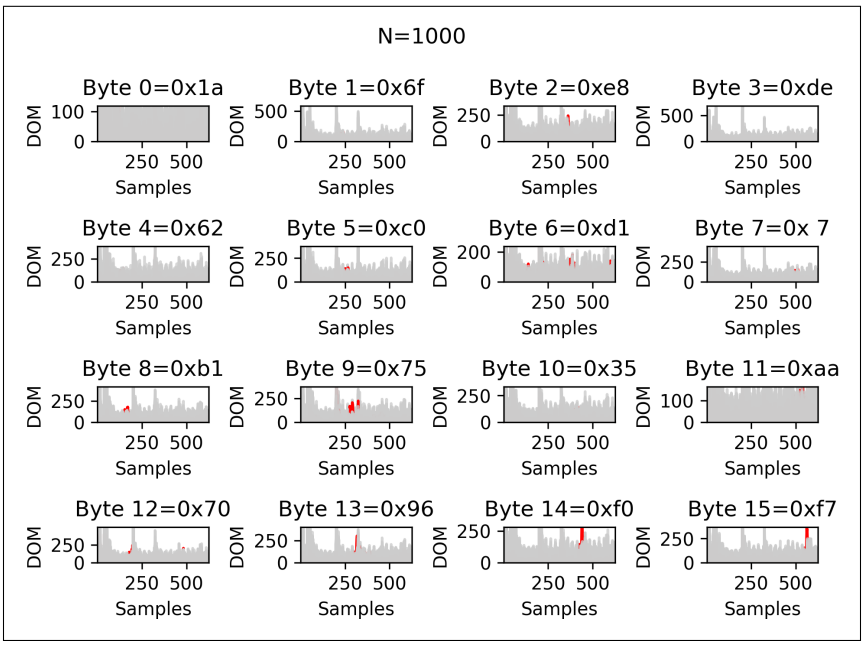

Figure 3: Result of the attack using 1000 traces and bit 5

3 Attacking the input of the SubBytes operation

The next question ask to rewrite the script so that we are now attacking the input of the SubBytes operation by using the preferred distinguisher. To attack this intermediate result it is sufficient to remove in the **Initialize** function the line

Y=AESAttack().Sbox(Y)

while the rest of the script remains unchanged. Based on the previous results the best attack is the one that uses Hamming weight as the leakage model, and correlation as distinguisher. If we run the attack until 1000 traces we won't get the whole keys correct but just some of them. The picture 4 shows the best result (just 5 bytes out of 16 are correct).

The combination of Hamming Weight and the correlation as distinguisher is not as good as the case reported in the chapter 1. This is because without applying the S-box, the intermediate values do not correctly represent the state that the traces are aligned to. So the correlation diminishes, and the key recovery becomes unreliable.

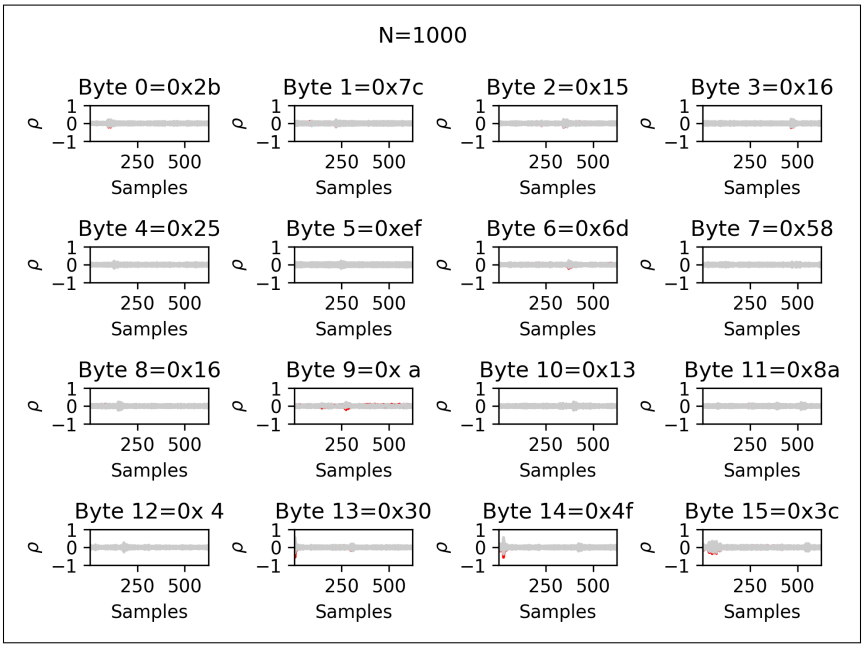

Figure 4: Result of the attack using 1000 traces

4 Number of needed traces

We can summarize the outcomes of the previous attacks:

- When attacking the output of the SubBytes operation with Hamming weight leakage model, and correlation as a distinguisher are required 150 traces;
- When attacking the output of the SubBytes operation with a single bit leakage model, and correlation as a distinguisher are required 1000 traces (with bit 5 it is possible to retrieve all the keys);

- When attacking the output of the SubBytes operation with a single bit leakage model, and difference of means as a distinguisher none of the keys for any bit is correct (with 1000 traces);
- When attacking the input of the SubBytes operation with Hamming weight leakage model, and correlation as a distinguisher just 5 keys out of 16 are correct (with 1000 traces).

To compute the number of needed traces to succeed also in the last two attacks we can use the approach based on the correlation coefficient and the signal-to-noise ratio (SNR). The correlation coefficient ρ evaluates the correlation between the predicted power consumption (based on a key hypothesis) and the measured power traces while the SNR is defined as the ratio of the average power of a signal (meaningful information) to the average power of background noise.
The formula to compute the number of traces needed to distinguish the correct key hypothesis is:

$$n = 3 + 8 \cdot \frac{z_{1-\alpha}^2}{\left(\ln \frac{1+\rho_{ck,ct}}{1-\rho_{ck,ct}}\right)^2},$$

where $z_{1-\alpha}$ is the quantile of the standard normal distribution for a confidence level $1-\alpha$, and $\rho_{ck,ct}$ is the correlation coefficient for the correct key at the correct time. For $\rho_{ck,ct} \leq 0.2$ and for small SNRs the following relation hold:

$$n \approx \frac{28}{\rho_{ck,ct}^2}$$

This latter formula can be computed to find an approximation of the number of needed traces to succeed in the last two attacks.

- **Attacking the output of SubBytes with a single bit leakage model and DoM as distinguisher**: since that the difference of means yields no success with 1000 traces, we can suppose that the value of $\rho_{ck,ct}$ is very small (close to zero) like 0.01, so an estimation of the total number of needed traces is

$$n \approx \frac{28}{0.01^2} = 280.000$$

- **Attacking the input of SubBytes with Hamming Weight leakage model and correlation as distinguisher**: in this case with 1000 traces we get 5 keys correct out of 16. This suggest that the value of $\rho_{ck,ct}$ is higher than before but it is still not sufficient to succeed in the attack. We can estimate this value around 0.05, so an estimation of the total number of needed traces is

$$n \approx \frac{28}{0.05^2} = 11.200$$

All the formulas has been taken from the Differential Attack Handout available on the Moodle page of the course.