01409 - Lightweight Cryptography E15

Student name and id: Mikkel Ole Rømer (s113408)

Project 2: Differential Power Analysis of AES

Date: 26-11-2015

1 Introduction

The following article will present a key-recovery attack on the AES 128 cryptographic environment. In specific, this paper will analyse, implement, and execute a differential power analysis, **DPA**, attack. Moreover, though-out this attack, this report will focus upon key-recovery of one single key-byte. However, the attack could be expanded to cover full key-recovery. Finally, this article assumes preliminary knowledge about cryptography, AES in specific, since this article will provide little or no knowledge upon this algorithm. The solution presented is implemented using python 2.7. Additionally, the code presented within this report is available at[1].

1.1 Differential Power Analysis (DPA)

Differential Power Analysis, **DPA**, is a physical attack which can be used on eg. lightweight micro-controllers, such as the ATmega16. The attack is physical in the sense, that the power consumption of the CPU is measured while performing a known data-transformation. By doing this, the inspector retrieves a time series explaining power consumption at time t. This information is use-full, since it reveals sensible information regarding the data-manipulation taking place inside the controller.

Power-consumption and bit flipping has linear correlation[2]. At least, when disregarding other controller components running in the environment. This fact allows us to do clever estimation upon the key used for data-encryption, by introducing the notion of Hamming Weight[2]. Hamming Weight, **HW**, is a measure of the number of 1'bits in a binary representation. For example, the byte representation of 3 is 00000011, thus $HM(3) = 2$. Power consumption doing data manipulations is highest whenever bits are flipped. That is, going from $1 \rightarrow 0 \vee 0 \rightarrow 1$. It is assumed, for simplicity, that power consumption is bidirectionally equal[3].

This knowledge allows the attacker Charlie, C, to predict the expected power utilisation doing the known data-transformation, with respect to the inputted byte-value. This is done by calculating the HW for every single key-byte, k,In this case 2^8 different bytes. Finally, C would be able to compare the expectations with the observed power levels, by normalising the data and calculating the correlation.

It is thus, possible to divide the attack into three steps:

1. Obtain N power traces, for N random input values, $a_1, a_2, ..., a_N$.

2. Predict the Hamming Weight Matrix, H, knowing that $H = HW(S(a_i \oplus k_j))$. Notice that the only unknown variable here is k_j. Thus this estimation must be performed for all possible 2^8-values.

3. Compute the correlation of the Hamming-Weight data-set and the actual power consumption, using Pearsons Correlation Coefficient.

2 Implementation

This section seeks to describe and introduce, the code proposed by this report.

2.1 Step 1. The Data

As mentioned previously, the first step of this analysis is to obtain the required data. This is illustrated in code-snippet 1.

Code-snippet 1: Loading Power Consumption Data (T) and corresponding inputs (Inputs)

```
T = np.loadtxt(input_t_path, delimiter=',')
Inputs = np.loadtxt(input_data_path, delimiter=',', dtype=(int))
```

Moreover, the power samples T could be plotted, to get an indicator of what this data-set provides. This is illustrated in figure 1. It is clear; that there are some significant trends in this set, notice the huge spikes which seems to be present in all observations. Moreover, notice how the magnitude of the spikes seems to be shifting in size but not in form. That is, they have the same shape, but different power requirements. These observations are created from completely random input bytes. Thus cautiously inferring that inputs alters magnitude, power usage, but not trend is proposed.

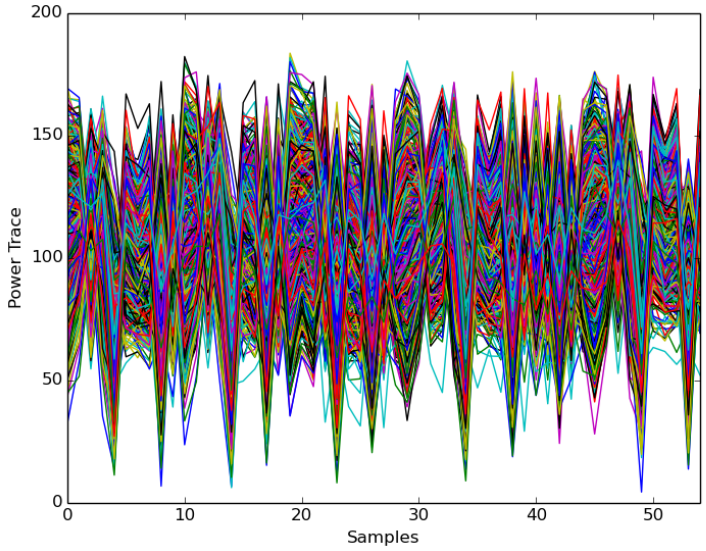

Figure 1: The 600 power consumptions observations plotted against measurements.

2.2 Step 2. Hamming-Weights

The second step includes calculating an $N \times 2^8$ Hamming-Weight Matrix. To do this, an implementation of the Hamming-Weight calculation is needed. This implementation is visible from code-snippet 2. The actual Matrix generation is visible from code-snippet 4. The code executes, by computing the Hamming-Weight of the S-Box value for the input, i, XOR'ed with the key k. As mentioned earlier. Since the key is unknown, this is done for each possible key-byte $1 \leq k \geq 2^8$.

Code-snippet 2: Lambda implementation of the Hamming-Weight

```
hamming_weight = lambda x: bin(x).count('1')
```

Code-snippet 3: Lambda implementation of the bit-wise $\oplus$ operation

```
xor = lambda x,y: (x ^ y) % 256
```

Code-snippet 4: Calculating the Hamming-Weight Matrix

```
def get_hamming_weight_table(inputs):
  A = []
    for ith_input in inputs:
      A.append(
        [hamming_weight(sbox[xor(ith_input, k)]) for k in xrange(1,257)]
      )
  return np.array(A)
```

The output is a data matrix, arranged such that each index pair (i, k) represents the expected Hamming Weight in the iTh input using key k. For example, the very first index pair $(0, 0)$ is the HW when inputting $0x49$, *the random input of the first row*, and encrypting using $k = 0x01$. The HW in this example is 3. In figure 2 the first column is plotted, illustrating expected Hamming Weight against the n'th sample, using key $k = 0x01$. Notice, that the key k is constant column-wise whilst the input is constant row-wise.

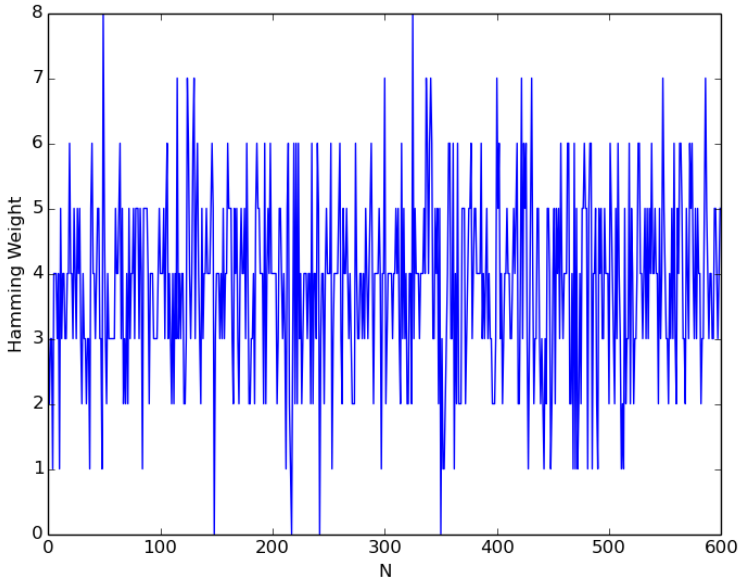

Figure 2: The 600 inputs plotted against the Hamming Weight, using key-byte $0x01$.

2.3 Step 3. Correlation

The final step concerns correlating every the columns of the Hamming Weight matrix, with all actual 55 power measures. In other words, each column vector of HW, should be correlated to each column vector of T. The output is an 256×55 matrix, since the operation is correlating 55 measures for 256 key-candidates. This provides Charlie with an indication of how well each key k, fits the measured power data. Since this investigation is dealing with two very different scales, the data must be normalised. Thus making the Pearsons correlation measurement a good choice[3][4]. For this calculation an expression for finding the mean of a vector is needed, available form code-snippet 5. Finally, the actual correlation should be calculated, using code 6 and 7.

Code-snippet 5: Lambda implementation of the average function; finding the mean of a vector

```
average = lambda x: sum(x) / len(x)
```

Code-snippet 6: The Pearson Correlation algorithm

```
def correlation(x, y):
  n = len(x)
  avg_x = average(x)
  avg_y = average(y)
  dx = dy = dp = dxpow = dypow = 0
  for i in xrange(n):
    dx = x[i] - avg_x
    dy = y[i] - avg_y
    dp += dx * dy
    dxpow += dx * dx
    dypow += dy * dy
  return dp / math.sqrt(dxpow * dypow)
```

Code-snippet 7: Correlating the Hamming Weight with the 55 measurements of the 600 inputs

```
Corr = []
for i in xrange(0, 256):
  Corr.append([correlation(HW[:,i],T[:,t]) for t in xrange(0,55)])
```

In figure 3, the correlation plot of the found correlation-matrix, *Corr*, is illustrated. It is observed; that one of the key-candidates is explaining much more accurate power consumption, than the others. Thus making this a likely choice of key. In figure 4, the data is filtered, and the likely candidate is shown alone.

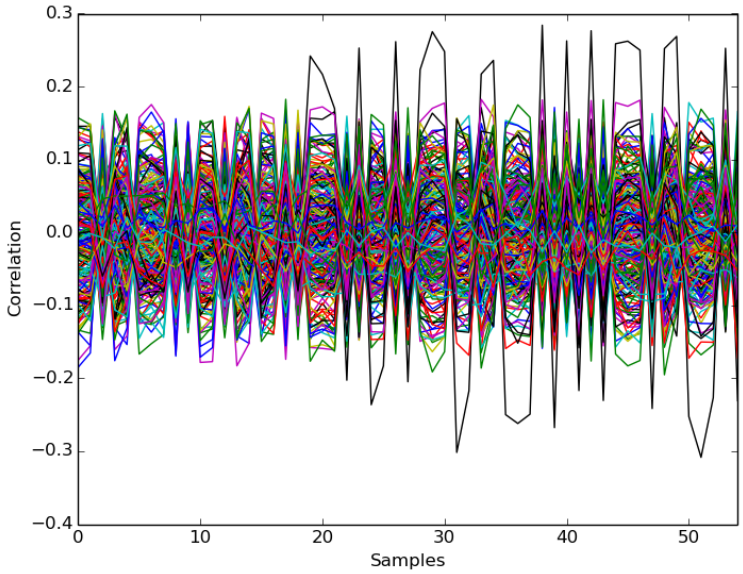

Figure 3: The correlation plot of the Hamming Weight and the power consumption, there seems to be a good key candidate.

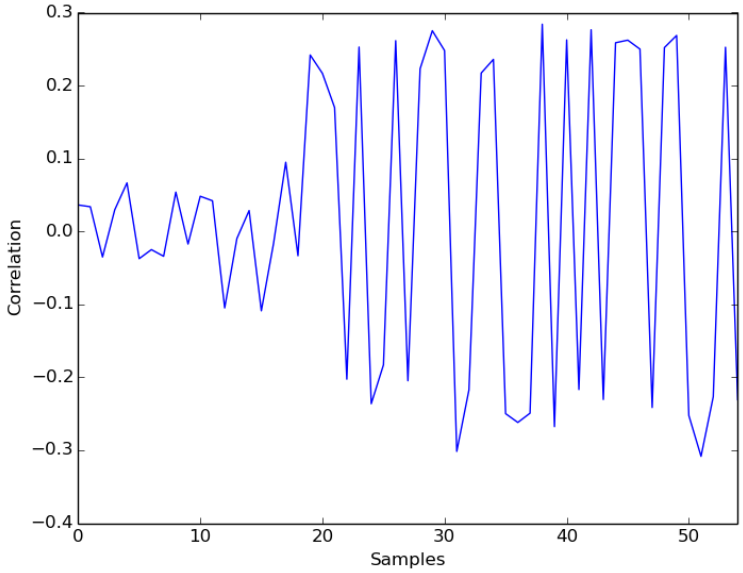

Figure 4: The best guess of a possible key candidate.

As seen from figure 3, one key candidate is explaining the power consumption more precise than the others. Thus, it is convenient to investigate this candidate further. Since each row of the correlation matrix explains all possible key candidates; going from 1 to 256. And the correlation, illustrated in figure 4, is found in row 139, the corresponding key is $140 = 0x8C$.

3 Conclusion

Through-out this paper it has been explained, and illustrated, how to obtain the secret key of a micro-controller. This was done actively by feeding different inputs to the encryption algorithm, and measuring the power consumption. The power consumption was then linked together with a Hamming Weight estimate on each possible key-byte. Finally, a key candidate was found by cleverly selecting the key explaining most of the observed power consumptions. The correlation was calculated using the Pearson correlation.

4 References

1. **Python Source Code**, 25-11-2015
   https://github.com/MikkelsCykel/LightweightCrypto/blob/master/Project2/Code/DPA.py

2. **Aditya Bagchi, Indrakshi Ray**, Information System Security, 9th international conference, p255-p268, ICISS 2013, ISBN: 978-3-642-45203-1.

3. **Eric Brier, Christophe Clavier, Francis Olivier**, Correlation Power Analysis with a Leakage Model. https://www.iacr.org/archive/ches2004/31560016/31560016.pdf

4. **Lejla Batina, Benedikt Gierlichs, Kerstin Lemke-Rust**, Comparative Evaluation of Rank Correlation, ESAT/SCD-COSIC and IBBT,
   https://securewww.esat.kuleuven.be/cosic/publications/article-1135.pdf Based DPA on an AES Prototype Chip