Bachelor thesis
Computing Science

Radboud University

Comparing Correlation Coefficient and Difference of Means in a Differential Power Analysis Attack

Author:
Maaike van Leuken
s4641493

First supervisor/assessor:
prof. dr. L. Batina
lejla@cs.ru.nl

Second assessor:
MSc N. Samwel
nsamwel@science.ru.nl

July 6, 2019

Abstract

The Advanced Encryption Standard is the most important symmetric-key algorithm for bulk data at the moment. This thesis focuses on revealing the secret key of the Advanced Encryption Standard algorithm on a micro-controller, by using differential power analysis. Differential power analysis requires power measurements, a power model and a statistical model. First, we make a test vector leakage assessment to see whether the system leaks power information. Then, we show that Correlation Power Analysis (power model: Hamming weight, statistical model: correlation coefficient) is more efficient than standard DPA (power model: least significant bit, statistical model: difference of means). The first attack requires around at least 400 traces, whereas the second one needs at least 2150 traces.

Contents

Chapter 1

Introduction

In 1998, Joan Daemen and Vincent Rijmen proposed the block cipher Rijndael, which in 2001 was established as the Advanced Encryption Standard (AES) by the National Institute of Standards and Technology (NIST). AES is the most widely used symmetric-key algorithm for encrypting and decrypting large amounts of data. Over the years, AES has proven itself as a reliable algorithm for bulk encryption and decryption. AES resists differential and linear cryptanalysis [2] due to the usage of rounds and nonlinear transformations. So far, the only attacks feasible with current technology are side-channel attacks. Side-channel attacks (SCAs) are based on exploiting the implementation of an algorithm, by looking at the power consumption, temperature, radiation of electromagnetic waves and so on. The CMOS circuits used to implement the logic gates in the microcontroller leak power consumption information. CMOS stands for complementary metal-oxide-semiconductor and its circuits use transistors (MOSFETs) [11]. "Electrons flow across the silicon substrate when charge is applied to (or removed from) a transistor's gate, consuming power and producing electromagnetic radiation."[5] We can measure this power consumption by connecting an oscilloscope to the microcontroller. This thesis will focus on using differential power analysis (DPA) to recover the key used in the AES encryption on a ARM Cortex-M4 microcontroller. Exploiting the vulnerabilities of the implementation of encryption standards has been done before, Kocher et al. already did this in 1999 on DES [5] and Mangard et al. researched DPA on AES thoroughly in 2007 [7]. However, papers about this topic do not always provide a concrete implementation of their attacks.

When implementing a DPA attack, we need to choose two models: a power model and a statistical analysis model. In this thesis, we draw a comparison between two different combinations of models, namely the Hamming weight model with the correlation coefficient (also called correlation power analysis, CPA) versus the least significant bit with the difference of means as distinguisher. We compare these two DPA attacks based on the minimum

amount of traces they require for resolving the correct key. We contribute to the following matters:

- An application of a test vector leakage assessment to show that the set of traces used leaks power consumption information;
- A detailed description on how to perform both the DPA attacks and their Python code;
- A comparison between the two attacks, showing that CPA requires less traces than a difference of means based attack.

The fact that CPA is more efficient in the amount of traces can be used in further research and attacks and should be kept in mind when assessing the security of a system. Also, since we provide the code of the implementations of the attacks, further research can be picked up where we left off.

In real-life settings, the subject of a differential power analysis attack is often a smart card. A smart card has a microcontroller running a cryptographic algorithm such as AES. Differential power analysis attacks are sometimes not feasible because they require a large amount of power traces and a way to acquire those power traces, such as an oscilloscope. DPA attacks also require the collection of the plaintexts associated to the power traces. However, we can get access to the smart card's embedded microcontroller, allowing us to measure the power consumption [7]. Then, DPA attacks can be conducted very easily and quickly. Power analysis attacks are easy to implement and are hard to detect, due to the non-invasiveness of the attack. "Because DPA automatically locates correlated regions in a device's power consumption, the attack can be automated and little or no information about the target implementation is required."[5]. This means that DPA attacks are not specific to a certain microcontroller.

We start this thesis with background information on AES and DPA. In Chapter 3 we discuss the technical details on the attacks to show that our claim holds. Chapter 4 presents related work, containing both countermeasures to or improvements to DPA attacks. The conclusions are be discussed in Chapter 5.

Chapter 2

Preliminaries

To substantiate the claims made in Chapter 1, we first need some background knowledge on the Advanced Encryption Standard and differential power analysis attacks.

2.1 The Advanced Encryption Standard

AES is a symmetric-key algorithm, which means the same key is used for both encryption and decryption. There are multiple versions of AES, which differ in the key length. In our research we used AES-128. This is a version of AES with a key length of 128 bits, which is 16 bytes. The plaintext and ciphertext length are also 128 bits. AES-128 consists of 10 rounds. Before the rounds are performed, the secret key is XORed with the plaintext. The first nine rounds consist of four stages: SubBytes, ShiftRows, MixColumns and AddRoundKey. In the tenth round the MixColumns operation is not performed as can be seen in figure 2.1. For a detailed description see [3].

We are particularly interested in the SubBytes step. In this step, a S-box lookup table is used to substitute each byte for another one. The SubBytes step is the only nonlinear transformation in AES, used to minimize the correlation between the input and the output of the function. This protects AES from linear cryptanalysis. We will use this information in section 2.2.

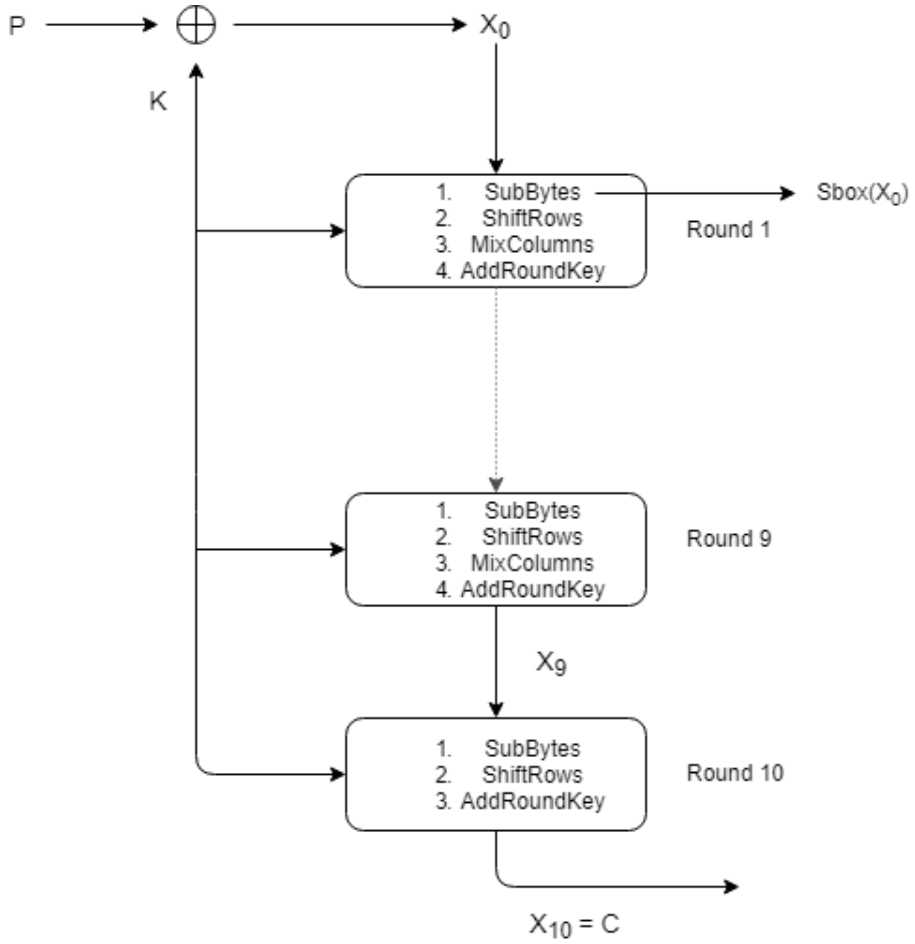


Figure 2.1: Schematic of the AES rounds based on [3]. The plaintext P is XORed with the key K. AES uses a key schedule to define different keys for each of the rounds. However, the key that we XOR with the plaintext before the first round is just the secret key. The output of the first SubBytes operation is $Sbox(plaintext \oplus key)$. The output of the tenth round is the ciphertext C.

2.2 Differential Power Analysis

DPA attacks exploit the correlation between the power consumption and the data that is being processed. These attacks use a general strategy consisting of five steps [7]. In these steps certain design choices are made, such as which power model and statistical model are used. These steps can be used for any DPA attack and require little knowledge on the technical details of the microcontroller. The attacks we describe are thus not specific to a certain microcontroller.

1. *Choosing an intermediate result of AES.*
   As an intermediate value, we can choose any intermediate result as long as it can be described as a function of a plaintext byte and a key byte. We will use the output of the first SubBytes operation as our intermediate result. Before the first round, the secret key is XORed with the plaintext. After that, the SubBytes function is applied. Like we have seen in section 2.1, the intermediate value is a function

$$S(plaintext \oplus key) \tag{2.1}$$

2. *Collecting traces.*
   We will measure the power consumption of $D = 10,000$ encryptions and collect the plaintexts (and ciphertexts) associated with each of the encryption runs. These plaintexts are randomly generated by the software implementation of AES that runs on the microcontroller. We measure the power consumption by connecting an oscilloscope to the microcontroller. Each power trace has $T = 110,000$ samples, where each sample is a data point in time with an associated power consumption.

3. *Calculating the hypothetical intermediate values.*
   Since we chose the output from the first SubBytes operation as our intermediate result, we can compute the hypothetical intermediate values as in 2.1. If we would do this for all possible keys for a 16-byte key, this would be a brute-force attack and these are infeasible with current technology (2^{128} computations). Hence we try to recover each key byte separately, thus doing steps 3 through 5 for each of 16 bytes. We compute all possible keys for a byte, yielding $K = 256$ keys from integer values 0 to 255 (security strength then transforms from 2^{128} to $16 \cdot 2^8 = 2^{12}$). The hypothetical intermediate value matrix $\mathbf{V}$ is a D by K matrix with $S(plaintext \oplus key)$ for every plaintext byte and every key byte possibility.

4. *Calculating the hypothetical power consumption values.*
   Power models are used to map the hypothetical intermediate values to hypothetical power consumption values. We compute a D by K matrix $\mathbf{H}$ by running the power model over $\mathbf{V}$. These models are described in sections 3.2.1 and 3.2.2.

5. *Finding the correlation between the hypothetical power consumption values and the traces.*
   The columns h_i (column i of $\mathbf{H}$) and t_j (column j of $\mathbf{T}$) are compared with a statistical model. The comparison is $r_{i,j}$ and we store this value in the matrix $\mathbf{R}$ of size $K \times T$. We use two different statistical models in this paper, which will both be explained in 3.3. The entry $r_{i,j}$ in

R with the highest value reveals the correct key i for a byte and the sample j at which the SubBytes operation was performed.

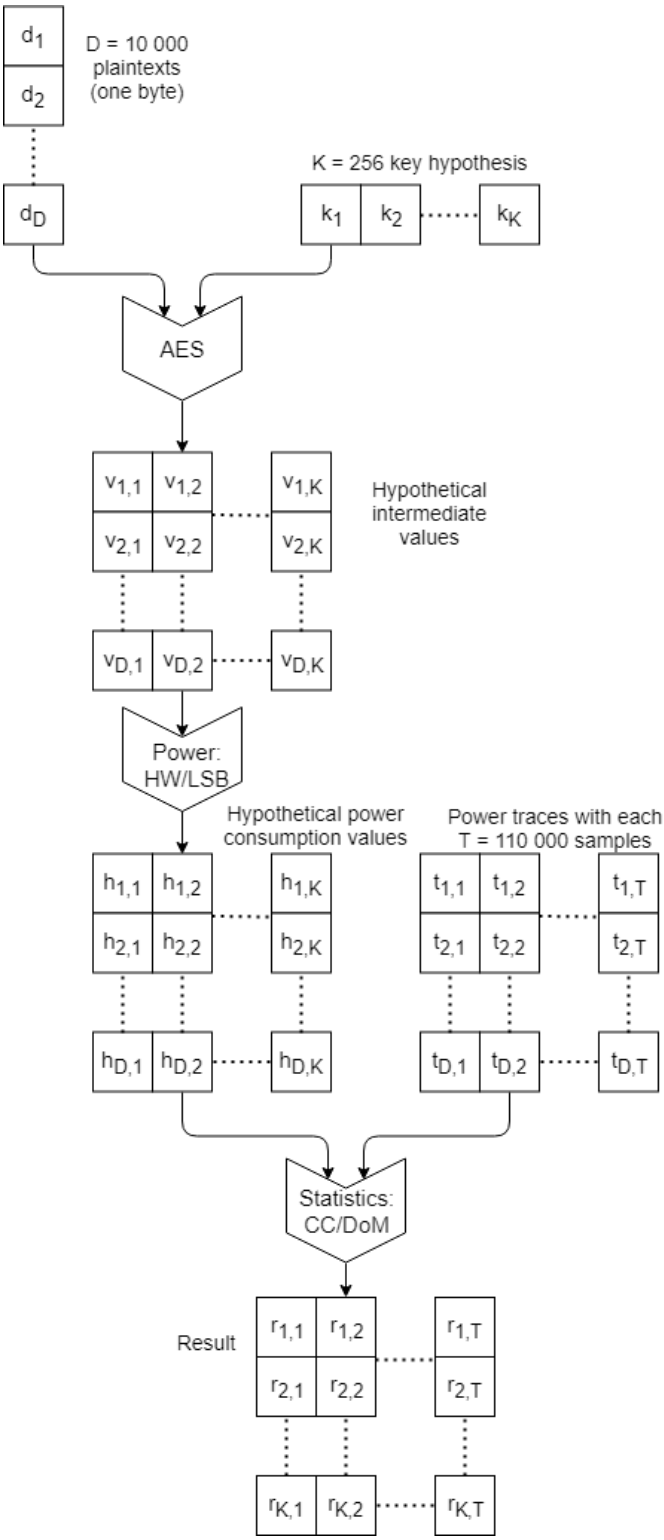


Figure 2.2: A visualisation of the steps described above. This figure is based on the block diagram shown on page 122 of [7].

Chapter 3

Research

In this Chapter, the details on both attacks are described, as well as the application of the test vector leakage assessment.
We measure the power traces with the setup displayed in figure 3.1 and store them in a matrix $\mathbf{T} \in M_{D \times T}$.

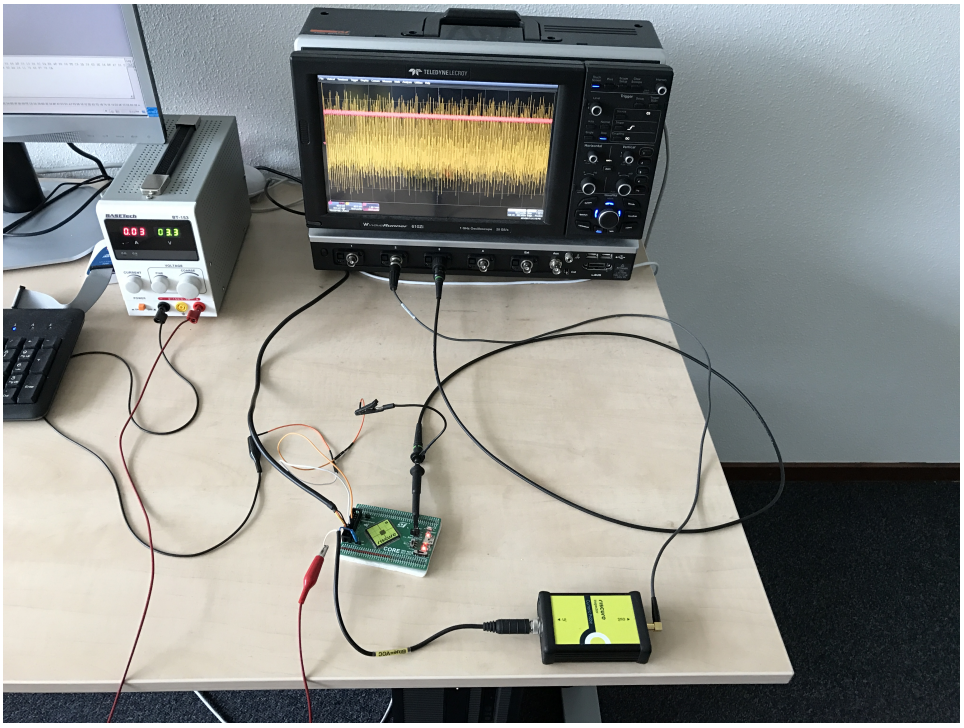


Figure 3.1: The setup used to collect the power traces. The ARM Cortex-M4 microcontroller is connected to a power supply and to an oscilloscope via a Riscure current probe. This current probe is used to measure the power consumption with high accuracy.

The power consumption is measured relatively, the values are between -128 and 127. As we will see later on in 3.3, we do not need the absolute power values. We just need the relation between consecutive samples.

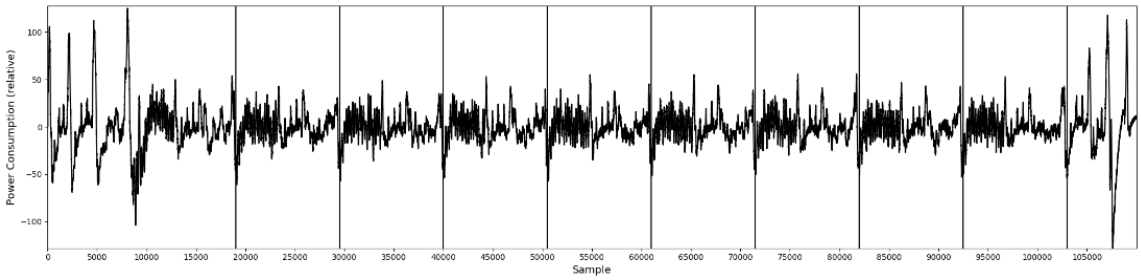


Figure 3.2: The power trace of a single AES encryption. The ten vertical lines represent the end of a round. We only concern ourselves with the first round, which roughly ends at sample $j = 20,000$.

In figure 3.2 the power trace of the first AES encryption can be seen. This power trace is associated to a certain plaintext, as is true for all $10,000$ collected traces. The trace contains a pattern, which allows us to distinguish ten rounds. This already gives us an indication on where the first SubBytes operation is, sample-wise. We know that the first round ends roughly at sample $j = 20,000$, so we do not necessarily have to include the samples beyond that point in our attack. We will first have a look at whether a DPA attack indeed is possible, by making a test vector leakage assessment. Then we perform the two attacks and look at their results.

3.1 Test Vector Leakage Assessment

A test vector leakage assessment (TVLA) determines whether a set of traces verifies the presence of leakage. It is a simple, quick test assessing whether we can perform the DPA attack or not.[10] When doing the TVLA, Welch's t-test is computed for every point in time, thus for every sample. We need two traces sets: one with all random plaintexts (such as the one we use for the attacks) and one with a certain fixed plaintext. Welch's t-test can be computes as follows:

$$t = \frac{\bar{X}_1 - \bar{X}_2}{\sqrt{\frac{s_1^2}{N_1} + \frac{s_2^2}{N_2}}} \tag{3.1}$$

Where $\bar{X}_i$ is the mean, s_i the standard deviation and N_i the sample size of set i. The result is shown in figure 3.3. If the t-test value is higher than 4.5, there is sufficient leakage to perform a DPA attack. This means that in our setup, there is enough leakage present, so we can distinguish the two sets of traces from each other.

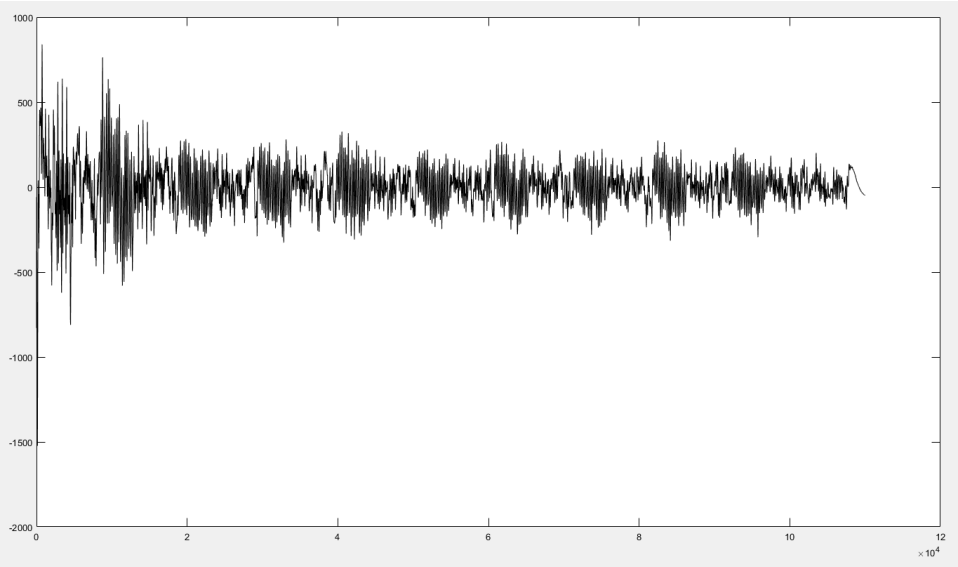


Figure 3.3: The test vector leakage assessment for the setup shown in figure 3.1.

3.2 Power Models

We will first have a look at the different power models. Power models are used to map the hypothetical intermediate values to the hypothetical power consumption values (as in step 4 in 2.2).

3.2.1 Hamming Weight

For the first attack, we use the Hamming Weight model. The Hamming Weight (HW) model maps a byte to an integer, which is the number of logical ones in the bit representation of the intermediate value. There are nine possible values for the HW of one byte, 0 through 8. The first one corresponds to a byte consisting solely of zeros, the latter one to a byte containing only ones. For every value in $\mathbf{V}$, we look at the bit representation of that byte and count the logical ones resulting in a D by K matrix $\mathbf{H}$. Every entry in this matrix is calculated as $HW(Sbox(plaintext \oplus key))$ (see Appendix C.1, lines 89 - 95).

3.2.2 Least Significant Bit

The Least Significant Bit (LSB) model takes that bit of the byte that has the lowest corresponding integer value. Since we are using a big-endian ordering, the LSB is the last bit. The LSB can have either of two values, 0 or 1. Hence, the LSB is a binary power model. We compute the LSB of every entry in $\mathbf{V}$, resulting in the matrix $\mathbf{H}$, which has dimensions D by K like $\mathbf{V}$. Every entry in this matrix is calculated as $LSB(Sbox(plaintext \oplus key))$ (see Appendix C.2, lines 77 - 79).

3.3 Statistical Analysis Models

Now we can apply statistics to compare the hypothetical power consumption values with the actual power traces (as in step 5 in 2.2).

Note that the statistical methods indeed can be applied, since the plaintexts that are used are randomly generated such that the bits are normally distributed. After having used the power model to calculate $\mathbf{H}$, we calculate the result matrix $\mathbf{R}$, which is a K by T matrix. Every element in this matrix is a comparison between a column of $\mathbf{H}$ and a column of $\mathbf{T}$. The relative power consumption suffices, since we only concern ourselves with how good the columns match. We need a statistical model to find this match.

3.3.1 Correlation Coefficient

When attacking a software implementation of AES, the combination of the HW model and the correlation coefficient is generally used [7]. The correlation coefficient is used to calculate the linear relationship between the columns of the two matrices $\mathbf{H}$ and $\mathbf{T}$. The correlation coefficient for i and j is computed by equation 3.3 (see Appendix C.1, lines 107 - 114).

$$R = \frac{Cov(\mathbf{H}, \mathbf{T})}{\sigma_{\mathbf{H}} \cdot \sigma_{\mathbf{T}}} = \frac{Cov(\mathbf{H}, \mathbf{T})}{\sqrt{Var(\mathbf{H}) \cdot Var(\mathbf{T})}} \tag{3.2}$$

$$r_{i,j} = \frac{\sum_{d=1}^{D}(h_{d,i} - \bar{h_i}) \cdot (t_{d,j} - \bar{t_j})}{\sqrt{\sum_{d=1}^{D}(h_{d,i} - \bar{h_i})^2 \cdot \sum_{d=1}^{D}(t_{d,j} - \bar{t_j})^2}} \tag{3.3}$$

The matrix $\mathbf{R}$ has for every $i \in K$ and for every $j \in T$ an entry containing the correlation between key i and sample j. As seen before, we only need the first $20,000$ samples, which saves us quite some computations. We find the key i and the sample j by finding the highest absolute value in $\mathbf{R}$. The indices of that value reveal the correct key and the 'time' (the sample) at which the SubBytes round of that byte was computed (see Appendix C.1, lines 119 - 131). For the first byte, the correlation for four key possibilities are plotted in figure A.1. In plots A.1a, A.1b and A.1d we do not see such high peaks as in plot A.1c. In fact, the first peak in A.1c is the highest correlation in $\mathbf{R}$. This reveals that the first byte of the key is 0xCA. We then repeat this for the other 15 bytes and reveal the key: CAFEBABEDEADBEEF01020304050607 in hexadecimal notation. The results can be be found in more detail in section B.1. Since we have gathered the plaintexts and ciphertext for each trace, we can check this key by encrypting the plaintext with AES and check the result with the ciphertext.

In figure 3.4 the column $j = 4,935$ of the result matrix $\mathbf{R}$ is shown for different amounts of traces for the first byte. The red line represents the correct key for the first byte, namely 0xCA. The correlation coefficient graph for other key bytes in general also have peaks, just much smaller ones than the correct key as can be seen in figure A.1. This is due to the fact that columns in $\mathbf{H}$ can be dependent of each other. When we have less than 400 traces, we see that other keys have a higher correlation than the correct key byte. So we need at least 400 traces to get the correct key byte with certainty.

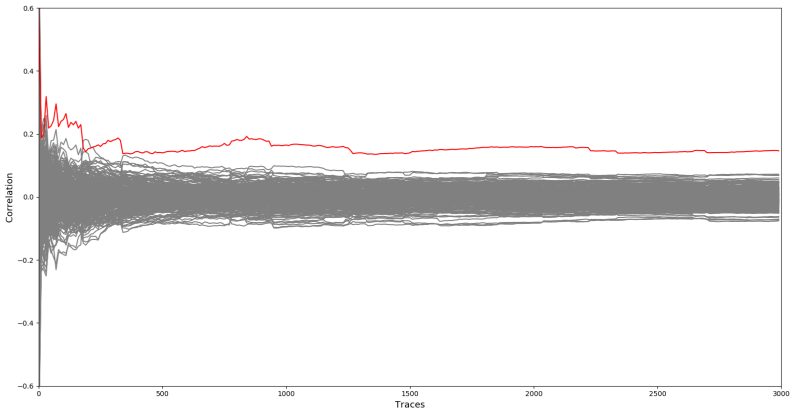


Figure 3.4: The column $j = 4,935$ of the result matrix $\mathbf{R}$ for the CPA attack is shown for different amounts of traces for the first byte. The red line represents the correct key for the first byte, namely 0xCA.

3.3.2 Difference of Means

In section 3.2.1 we saw that the HW model has 9 possible values for the hypothetical intermediate value. The difference-of-means method (DoM) requires a binary power model, so we cannot longer use the HW to compute the power consumption values. This is why we use the LSB model.

To compute the DoM, we need to divide the matrix $\mathbf{T}$ in two matrices, $\boldsymbol{T}_0$ and $\boldsymbol{T}_1$. For every key possibility i, we look at the column $\mathbf{H}_i$. This is a column of D elements, which are all 0 or 1 ($LSB(Sbox(plaintext \oplus key))$). $\boldsymbol{T}_{0,i}$ contains all rows $d \in D$ of $\mathbf{T}$, for which $\mathbf{H}_{d,i} = 0$. $\boldsymbol{T}_{1,i}$ contains all rows $d \in D$ of $\mathbf{T}$, for which $\mathbf{H}_{d,i} = 1$. Now we can calculate the result $\mathbf{R}$ as follows:

$$\boldsymbol{R} = \boldsymbol{M}_1 - \boldsymbol{M}_0 \tag{3.4}$$

where $\boldsymbol{M}_1$ is the mean of $\boldsymbol{T}_1$ and $\boldsymbol{M}_0$ is the mean of $\boldsymbol{T}_0$. The means can be calculated for every key possibility (thus for a different split) and for every sample as:

$$m_{1_{i,j}} = \frac{1}{\#\boldsymbol{T}_{1,i}} \sum_{d=1}^{D} \boldsymbol{H}_{d,i} \cdot \boldsymbol{T}_{d,j} \tag{3.5}$$

$$m_{0_{i,j}} = \frac{1}{\#\boldsymbol{T}_{0,i}} \sum_{d=1}^{D} (1 - \boldsymbol{H}_{d,i}) \cdot \boldsymbol{T}_{d,j} \tag{3.6}$$

where $\#\boldsymbol{T}_{0,i}$ and $\#\boldsymbol{T}_{1,i}$ are the amount of traces in $\boldsymbol{T}_0$ and $\boldsymbol{T}_1$ for key possibility i, respectively, both containing about $\frac{D}{2} \approx 5,000$ traces (see Appendix C.2, lines 101 - 116).

For the first byte, the correlation for four key possibilities are plotted in figure A.2. In plots A.2a, A.2b and A.2d we again see much smaller peaks than in plot A.2c. This reveals that the first byte of the key is 0xCA, which was the same conclusion as in section 3.3.1. We perform the attack for the other 15 bytes and reveal the key: CAFEBABEDEADBEEF01020304050607 in hexadecimal notation. For the full results, see B.2.

In figure 3.5 we see that if we have less than 2,150 traces that other keys have a higher difference of means than the correct key byte. The difference of means for other key bytes in general also have peaks, as can be seen in figure A.2. For less than 2,150 traces, the correct key cannot be found with certainty.

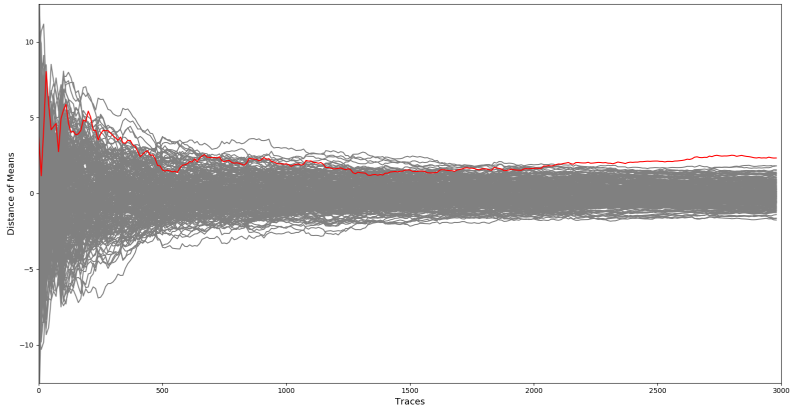


Figure 3.5: The column $j = 8,424$ of the result matrix $\mathbf{R}$ for the difference of means attack is shown for different amounts of traces for the first byte. The red line represents the correct key for the first byte, namely 0xCA.

Chapter 4

Related Work

Differential power analysis attacks can also be performed on other cryptosystems, such as RSA [12] and ECC [8][1].

Lomné et al. came up with a method for preprocessing the order of the traces that are attacked. The idea is that bigger peaks in the power consumption curve (PCC, we call it a trace) require less traces for the DPA attack to be successful. The traces set is order in decreasing order of data disclosure. "A classical DPA or CPA discloses the full round-key 16 in a bit less than 400 PCCs in average, whereas our preprocessing techniques allow to reach an average success rate of 1 with about 250 PCCs."[6] The paper focuses on DPA on DES, but it should also be applicable to AES, since the principles of DPA are independent of the cryptographic algorithm.

In [4] a countermeasure to all differential power analysis attacks is presented by using an integrated voltage regulator (IVR). "An inductive IVR is shown to transform the current signatures generated by an encryption engine. Furthermore, an all-digital circuit block, referred to as the loop-randomizer, is introduced to randomize the IVR transformations."[4] Kar et al. propose a circuit that eliminates power information leakage with a low overhead, using a IVR that is already present on the microcontroller.

In [9], Popp et al. divided the countermeasures against DPA into 3 categories:

1. Protocol. Changing the key frequently makes DPA impossible, since many traces are required. However, changing the key frequently enough is impractical or even impossible sometimes.

2. Hiding. The dependency between the power consumption and the processed data or operation used is minimized. The countermeasure described in [4] falls into this category.

3. Masking, "masking allows making the power consumption independent of the intermediate values, even if the device has a data-dependent power consumption."[7]

Chapter 5

Conclusions

In section 3.1, we have seen that the implementation of AES that we used showed leakage of power consumption information. If there was no leakage, the differential power analysis attack would not have worked. The absence of leakage of power consumption information could be due to one of the countermeasures described in Chapter 4.

For both attacks, we succeeded in revealing the secret key of AES. A noticeable difference between the two attacks is that the CPA attack finds the correct key at sample $j = 4,935$, whereas the difference-of-means attack found it at $j = 8,424$. The peak at $j = 8,424$ is actually a ghost peak, but the difference of means is for that place the highest. A ghost peak occurs when the same intermediate value is used in a different place in the algorithm. In this case, the highest value is not found at the SubBytes operation, but somewhere else. In section 3.3 we have seen that the correlation power analysis attack required at least 400 traces, whereas the difference of means attack required at least 2,150. We can conclude that the correlation coefficient as distinguisher is indeed more efficient in the amount of traces than the difference of means. The explanation for this result can be deduced from the formulas for the two statistical methods. The correlation coefficient (equation (3.2)) takes both the differences as the variances of the variables into account. The difference of means (equation (3.4)) only takes the differences into account. The correlation coefficient considers more aspects of statistics, which leads to a better comparison between the columns of the matrices. Besides that, there is another explanation for the fact that the difference of means attack is less efficient and finds the correct key at the wrong sample. Namely the fact that the difference of means attack uses the least significant bit as power model. This model only considers the last bit of a byte and ignores all other bits. The Hamming weight model which is used for the CPA attack does include all bits, which makes CPA more efficient and more accurate.

Bibliography

[1] Sven Bauer. Attacking exponent blinding in RSA without CRT. In *Constructive Side-Channel Analysis and Secure Design*, pages 82–88. Springer Berlin Heidelberg, 2012.

[2] Joan Daemen and Vincent Rijmen. *Design of Rijndael : AES - The Advanced Encryption Standard*. Springer Berlin Heidelberg, 2002.

[3] Michael T. Goodrich and Roberto Tamassia. *Introduction to Computer Security*. Pearson, 2011.

[4] Monodeep Kar, Arvind Singh, Sanu K. Mathew, Anand Rajan, Vivek De, and Saibal Mukhopadhyay. Reducing power side-channel information leakage of AES engines using fully integrated inductive voltage regulator. *IEEE Journal of Solid-State Circuits*, 53(8):2399–2414, August 2018.

[5] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. *Advances in Cryptology — CRYPTO' 99 Lecture Notes in Computer Science*, page 388–397, 1999.

[6] Victor Lomné, Amine Dehbaoui, Philippe Maurine, Lionel Torres, and Michel Robert. Differential power analysis enhancement with statistical preprocessing. In *2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)*. IEEE, March 2010.

[7] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: revealing the secrets of smart cards*. Springer, 2007.

[8] Cédric Murdica, Sylvain Guilley, Jean-Luc Danger, Philippe Hoogvorst, and David Naccache. Same values power analysis using special points on elliptic curves. In *Constructive Side-Channel Analysis and Secure Design*, pages 183–198. Springer Berlin Heidelberg, 2012.

[9] Thomas Popp, Stefan Mangard, and Elisabeth Oswald. Power analysis attacks and countermeasures. *IEEE Design & Test of Computers*, 24(6):535–543, November 2007.

[10] Tobias Schneider and Amir Moradi. Leakage assessment methodology. *Journal of Cryptographic Engineering*, 6(2):85–99, Jun 2016.

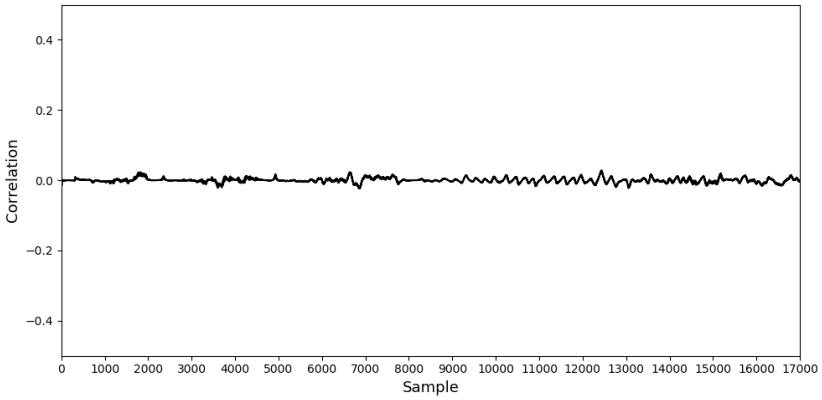
[11] John P. Uyemura. *Physics and Modelling of MOSFETs*, pages 1–60. Springer US, Boston, MA, 2001.

[12] Yiwei Zhang, Xinjian Zheng, and Bo Peng. A side-channel attack countermeasure based on segmented modular exponent randomizing in RSA cryptosystem. In *2008 11th IEEE Singapore International Conference on Communication Systems*. IEEE, November 2008.

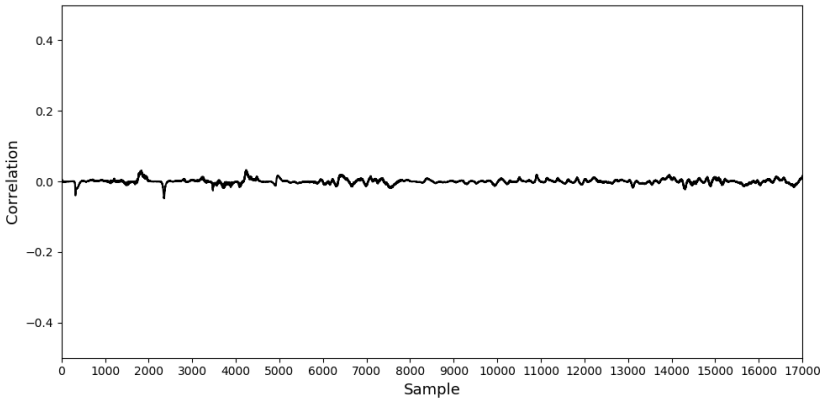
Appendix A

Differential Curves for Key Hypotheses

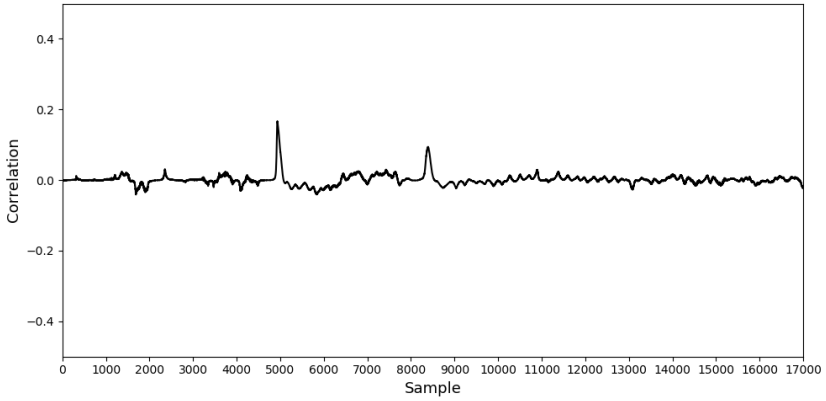
A.1 Hamming Weight with Correlation Coefficient



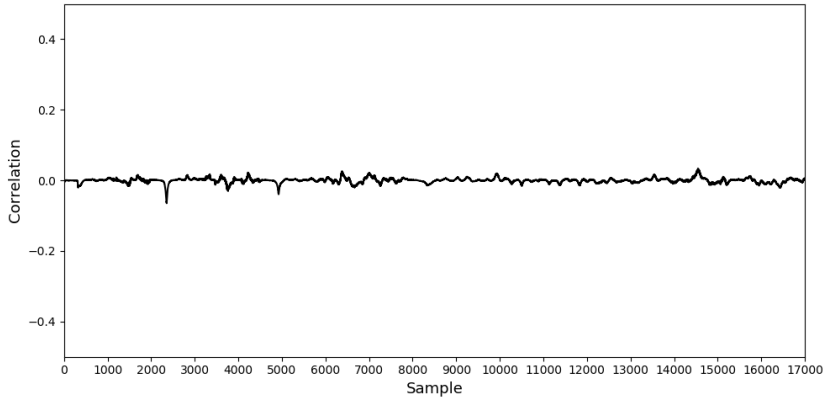
(a) The correlation per time point for key hypothesis 0xBE.



(b) The correlation per time point for key hypothesis 0xBF.



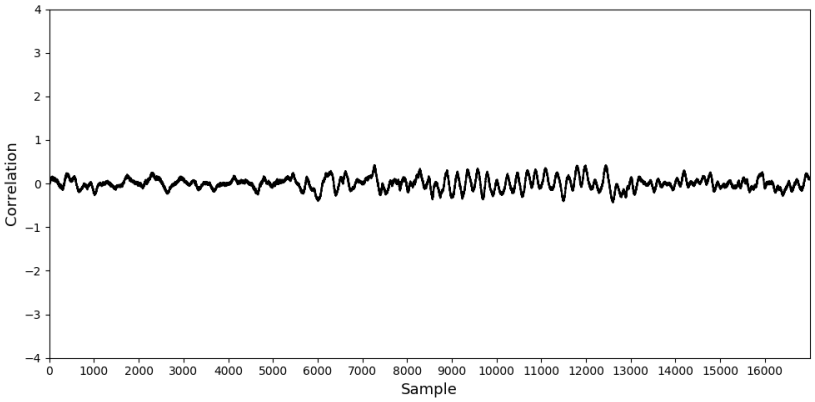
(c) The correlation per time point for key hypothesis 0xca. Two peaks can be clearly distinguished. The first occurs when the SubBytes operation in the first round is executed. The second one is a 'ghost' peak, it occurs when the same byte is also in another operation.



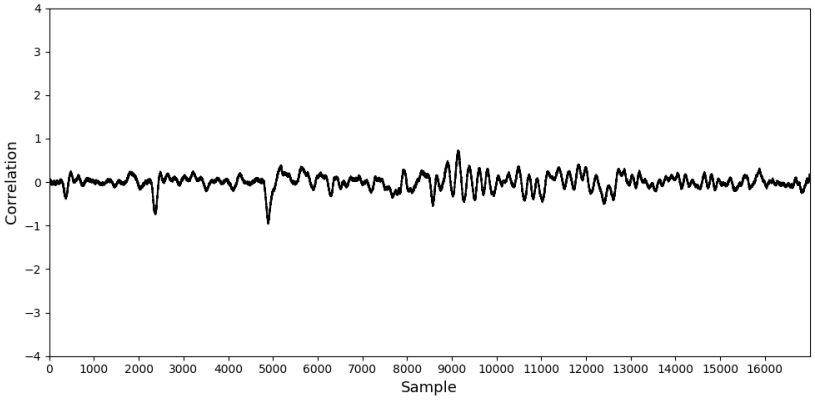
(d) The correlation per time point for key hypothesis 0xcb.

Figure A.1: The graphs corresponding to rows 200, 201, 202 and 203 (0xbe, 0xbf, 0xca, 0xcb). In figure A.1c peaks can be distinguished. The correlation value at sample $j = 4935$ is in fact the highest of the entire matrix. All other values, and the values in the other figures, are significantly lower. We can thus conclude that $i = 202 =$ 0xca is the correct key for byte 0.

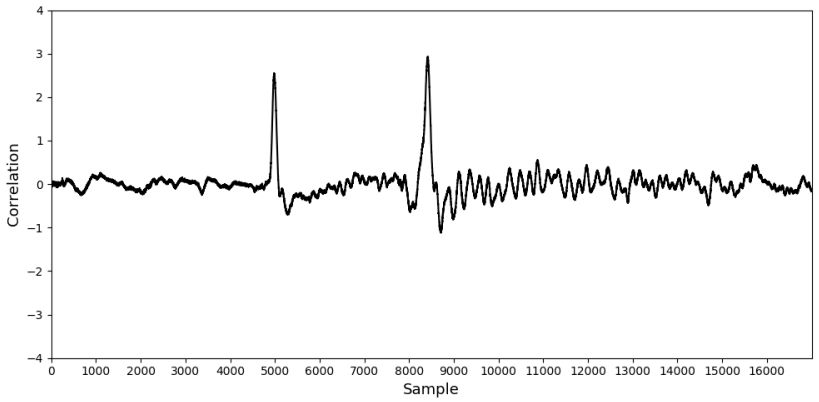
A.2 Least Significant Bit with Difference of Means



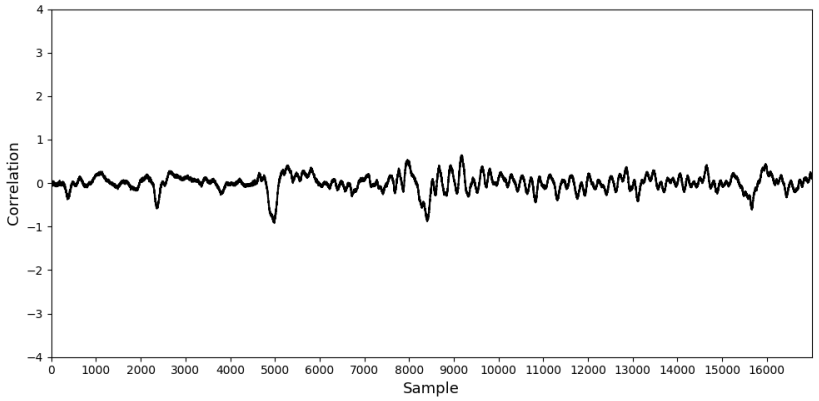
(a) The correlation per time point for key hypothesis 0xBE.



(b) The correlation per time point for key hypothesis 0xBF.



(c) The correlation per time point for key hypothesis 0xca. Two peaks can be clearly distinguished. The first occurs when the SubBytes operation in the first round is executed. The second one is a 'ghost' peak, it occurs when the same byte is also used in another operation later in the algorithm.



(d) The correlation per time point for key hypothesis 0xcb.

Figure A.2: The graphs corresponding to rows 200, 201, 202 and 203 (0xbe, 0xbf, 0xca, 0xcb). In figure A.2c, high peaks can be distinguished. The correlation value at sample $j = 8,424$ for key possibility 0xCA is the highest of the entire matrix. We can thus conclude that $i = 202 =$ 0xca is the correct key for byte 0.

Appendix B

Results of the Programs

B.1 Results for the Correlation Coefficient with Hamming Weight

byte number	key byte	sample point	correlation
0	0xca	4935	0.17
1	0xfe	5747	0.15
2	0xba	6503	0.38
3	0xbe	7270	0.41
4	0xde	9530	0.09
5	0xad	5934	0.22
6	0xbe	6691	0.41
7	0xef	7445	0.41
8	0x0	3565	0.13
9	0x1	6111	0.31
10	0x2	6866	0.40
11	0x3	7622	0.39
12	0x4	11813	0.16
13	0x5	6288	0.42
14	0x6	7041	0.37
15	0x7	7763	0.18

B.2 Results for the Difference of Means with Least Significant Bit

byte number	key byte	sample point	correlation
0	0xca	8424	2.93
1	0xfe	7983	3.07
2	0xba	8078	2.56
3	0xbe	8215	2.19
4	0xde	5157	2.09
5	0xad	5934	2.23
6	0xbe	8142	3.84
7	0xef	7447	2.21
8	0x0	5348	2.32
9	0x1	8006	2.48
10	0x2	6868	2.19
11	0x3	8214	1.96
12	0x4	5522	2.34
13	0x5	6286	2.17
14	0x6	8138	2.15
15	0x7	7820	2.33

Appendix C

Code

C.1 Code for HW and CC

```
"""
CPA attack on AES.
The correlation coefficient is used as distinguisher,
with the Hamming weight as power model.
"""

from __future__ import division
import itertools
import numpy as np

"""
TRACES

The class trs2npz parses a .trs file into a .npz file.
We only need to parse the traces set once.
Include import trs2npz.
"""

# trs2npz.main("traces")


# The byte substitution look-up table in hexadecimal notation for the SubBytes round.
Sbox = (
['63', '7C', '77', '7B', 'F2', '6B', '6F', 'C5', '30', '01', '67', '2B', 'FE', 'D7', 'AB', '76'],
['CA', '82', 'C9', '7D', 'FA', '59', '47', 'F0', 'AD', 'D4', 'A2', 'AF', '9C', 'A4', '72', 'C0'],
['B7', 'FD', '93', '26', '36', '3F', 'F7', 'CC', '34', 'A5', 'E5', 'F1', '71', 'D8', '31', '15'],
['04', 'C7', '23', 'C3', '18', '96', '05', '9A', '07', '12', '80', 'E2', 'EB', '27', 'B2', '75'],
['09', '83', '2C', '1A', '1B', '6E', '5A', 'A0', '52', '3B', 'D6', 'B3', '29', 'E3', '2F', '84'],
['53', 'D1', '00', 'ED', '20', 'FC', 'B1', '5B', '6A', 'CB', 'BE', '39', '4A', '4C', '58', 'CF'],
['D0', 'EF', 'AA', 'FB', '43', '4D', '33', '85', '45', 'F9', '02', '7F', '50', '3C', '9F', 'A8'],
['51', 'A3', '40', '8F', '92', '9D', '38', 'F5', 'BC', 'B6', 'DA', '21', '10', 'FF', 'F3', 'D2'],
['CD', '0C', '13', 'EC', '5F', '97', '44', '17', 'C4', 'A7', '7E', '3D', '64', '5D', '19', '73'],
['60', '81', '4F', 'DC', '22', '2A', '90', '88', '46', 'EE', 'B8', '14', 'DE', '5E', '0B', 'DB'],
['E0', '32', '3A', '0A', '49', '06', '24', '5C', 'C2', 'D3', 'AC', '62', '91', '95', 'E4', '79'],
['E7', 'C8', '37', '6D', '8D', 'D5', '4E', 'A9', '6C', '56', 'F4', 'EA', '65', '7A', 'AE', '08'],
['BA', '78', '25', '2E', '1C', 'A6', 'B4', 'C6', 'E8', 'DD', '74', '1F', '4B', 'BD', '8B', '8A'],
```

```
['70', '3E', 'B5', '66', '48', '03', 'F6', '0E', '61', '35', '57', 'B9', '86', 'C1', '1D', '9E'],
['E1', 'F8', '98', '11', '69', 'D9', '8E', '94', '9B', '1E', '87', 'E9', 'CE', '55', '28', 'DF'],
['8C', 'A1', '89', '0D', 'BF', 'E6', '42', '68', '41', '99', '2D', '0F', 'B0', '54', 'BB', '16'],
)


# Compute all key hypotheses. There are K = 256 possibilities for a key byte.
def init_keys():
    keys = list(itertools.product([0, 1], repeat=8))
    for poss in range(K):
        keys[poss] = hex(int(''.join(map(str, keys[poss])), 2))  # Rewrite to hexadecimal.
    return keys


"""
HYPOTHETICAL INTERMEDIATE VALUES

Step 3 of the DPA attack.
Result: D by K matrix V.
"""


# Calculate the hypothetical intermediate values, a D x K matrix V.
# For all plaintexts and key possibilities, compute Sbox(plaintext XOR key).
def hypo_inter_values(byte):
    v = []
    for i in range(D):
        a = data[i][byte]
        x = ["0x" + Sbox[(a ^ int(keys[k], 16)) // 16][(a ^ int(keys[k], 16)) % 16] for k in range(K)]
        v.append(x)
    return v


"""
HYPOTHETICAL POWER CONSUMPTION VALUES

Step 4 of the DPA attack, using the Hamming weight model.
Result: D by K matrix H.
"""


# Counts the amount of logical ones in a byte in binary.
def counter(b):
    count = 0
    for i in range(len(b)):
        if b[i] == "1":
            count += 1
    return count


# Calculate the hypothetical power consumption values, a D x K matrix H.
# For all plaintexts and key possibilities, compute HW(Sbox(plaintext XOR key)).
```

```
89  def hw_hypo_power():
90      h = []
91      for pt in range(D):
92          x = [counter("{0:b}".format(int(v[pt][k], 16))) for k in range(K)]
93          h.append(x)
94      h = np.array(h)
95      return h
96
97
98  """
99  RESULTS
100
101 Step 5 of the DPA attack.
102 Result: K by T matrix R.
103 """
104
105
106 # Compute the K x T result matrix R using the correlation coefficient.
107 def correlation():
108     diff_t = traces - (np.einsum("dt->t", traces, optimize='optimal') / np.double(D))
109     diff_h = h - (np.einsum("dk->k", h, optimize='optimal') / np.double(D))
110     cov = np.einsum("dk,dt->kt", diff_h, diff_t, optimize='optimal')  # Covariant.
111     std_h = np.einsum("dk,dk->k", diff_h, diff_h, optimize='optimal')  # Standard deviation of h.
112     std_t = np.einsum("dt,dt->t", diff_t, diff_t, optimize='optimal')  # Standard deviation of t.
113     temp = np.einsum("k,t->kt", std_h, std_t, optimize='optimal')
114     return cov / np.sqrt(temp)
115
116
117 # Find the highest correlation in R. The indices correspond to the correct key
118 # and the time at which the SubBytes round was processed.
119 def max_absolute(result):
120     highest_cor = 0
121     i = -1
122     j = -1
123     for row in range(256):
124         x = max(map(abs, result[row]))
125         if x > highest_cor:
126             highest_cor = x
127             i = row
128     for column in range(T_used):
129         if abs(result[i][column]) == highest_cor:
130             j = column
131     return i, j, highest_cor
132
133
134 def main():
135     help = ""
136     for byte in range(16):
137         help += key[byte][0]
138     print help
139
140
```

```
if __name__ == "__main__":
    D = 10000
    T_used = 20000
    T = 110000
    K = 256
    key = []
    dic = np.load('traces.npz', mmap_mode='r')
    data = dic['data']
    traces = dic['traces'][:, :T_used]
    keys = init_keys()
    for byte in range(16):  # For every byte, find the key, time and correlation.
        v = hypo_inter_values(byte)
        print "Hypothetical intermediate values computed."
        h = hw_hypo_power()
        print "Hypothetical power consumptions computed."
        result = correlation()
        print "Correlation computed."
        (i, j, highest_cor) = max_absolute(result)
        print "Byte " + str(byte) + " has been processed."
        key.append((keys[i], j, highest_cor))
    main()
```

C.2 Code for LSB and DoM

```
"""
DPA attack on AES.
The difference of means is used as distinguisher,
with the least significant bit as power model.
"""

from __future__ import division
import itertools
import numpy as np

"""
TRACES

The class trs2npz parses a .trs file into a .npz file.
We only need to parse the traces set once.
Include import trs2npz.
"""

# The byte substitution look-up table in hexadecimal notation for the SubBytes round.
Sbox = (
['63', '7C', '77', '7B', 'F2', '6B', '6F', 'C5', '30', '01', '67', '2B', 'FE', 'D7', 'AB', '76'],
['CA', '82', 'C9', '7D', 'FA', '59', '47', 'F0', 'AD', 'D4', 'A2', 'AF', '9C', 'A4', '72', 'C0'],
['B7', 'FD', '93', '26', '36', '3F', 'F7', 'CC', '34', 'A5', 'E5', 'F1', '71', 'D8', '31', '15'],
['04', 'C7', '23', 'C3', '18', '96', '05', '9A', '07', '12', '80', 'E2', 'EB', '27', 'B2', '75'],
['09', '83', '2C', '1A', '1B', '6E', '5A', 'A0', '52', '3B', 'D6', 'B3', '29', 'E3', '2F', '84'],
['53', 'D1', '00', 'ED', '20', 'FC', 'B1', '5B', '6A', 'CB', 'BE', '39', '4A', '4C', '58', 'CF'],
['D0', 'EF', 'AA', 'FB', '43', '4D', '33', '85', '45', 'F9', '02', '7F', '50', '3C', '9F', 'A8'],
```

```
    ['51', 'A3', '40', '8F', '92', '9D', '38', 'F5', 'BC', 'B6', 'DA', '21', '10', 'FF', 'F3', 'D2'],
    ['CD', '0C', '13', 'EC', '5F', '97', '44', '17', 'C4', 'A7', '7E', '3D', '64', '5D', '19', '73'],
    ['60', '81', '4F', 'DC', '22', '2A', '90', '88', '46', 'EE', 'B8', '14', 'DE', '5E', '0B', 'DB'],
    ['E0', '32', '3A', '0A', '49', '06', '24', '5C', 'C2', 'D3', 'AC', '62', '91', '95', 'E4', '79'],
    ['E7', 'C8', '37', '6D', '8D', 'D5', '4E', 'A9', '6C', '56', 'F4', 'EA', '65', '7A', 'AE', '08'],
    ['BA', '78', '25', '2E', '1C', 'A6', 'B4', 'C6', 'E8', 'DD', '74', '1F', '4B', 'BD', '8B', '8A'],
    ['70', '3E', 'B5', '66', '48', '03', 'F6', '0E', '61', '35', '57', 'B9', '86', 'C1', '1D', '9E'],
    ['E1', 'F8', '98', '11', '69', 'D9', '8E', '94', '9B', '1E', '87', 'E9', 'CE', '55', '28', 'DF'],
    ['8C', 'A1', '89', '0D', 'BF', 'E6', '42', '68', '41', '99', '2D', '0F', 'B0', '54', 'BB', '16'],
    )


    # Compute all key hypotheses. There are K = 256 possibilities for a key byte.
    def init_keys():
        keys = list(itertools.product([0, 1], repeat=8))
        for poss in range(K):
            keys[poss] = hex(int(''.join(map(str, keys[poss])), 2))  # Rewrite to hexadecimal.
        return keys


    """
    HYPOTHETICAL INTERMEDIATE VALUES

    Step 3 of the DPA attack.
    Result: D by K matrix V.
    """


    # Calculate the hypothetical intermediate values, a D x K matrix V.
    # For all plaintexts and key possibilities, compute Sbox(plaintext XOR key).
    def hypo_inter_values(byte):
        v = []
        for i in range(D):
            a = data[i][byte]
            x = ["0x" + Sbox[(a ^ int(keys[k], 16)) // 16][(a ^ int(keys[k], 16)) % 16] for k in range(K)]
            v.append(x)
        return v


    """
    HYPOTHETICAL POWER CONSUMPTION VALUES

    Step 4 of the DPA attack, using the least significant bit as power model.
    Result: D by K matrix H.
    """


    # Calculate the hypothetical power consumption values, a D x K matrix H.
    # For all plaintexts and key possibilities, compute LSB(Sbox(plaintext XOR key)).
    def LSB_hypo_power():
        h = [[int("{0:08b}".format(int(v[d][k], 16))[7]) for k in range(K)] for d in range(D)]
        return h
```

```
80
81
82  # Computes the ith column of a given array.
83  def column(array, i):
84      return [row[i] for row in array]
85
86
87  # Subtracts 1 by the ith column of a given array.
88  def neg_column(array, i):
89      return [1 - row[i] for row in array]
90
91
92  """
93  RESULTS
94
95  Step 5 of the DPA attack.
96  Result: K by T matrix R.
97  """
98
99
100 # Compute the K x T result matrix R using the difference of means.
101 def dom():
102     R = []
103     T0 = []
104     T1 = []
105     for k in range(K):
106         for d in range(D):
107             if h[d][k] == 0:
108                 T0.append(traces[d])
109             else:
110                 T1.append(traces[d])
111         M1 = np.einsum("d,dt->t", column(h, k), traces, optimize='optimal') / np.double(len(T1))
112         M0 = np.einsum("d,dt->t", neg_column(h, k), traces, optimize='optimal') / np.double(len(T0))
113         R.append(M1 - M0)
114         T1 = []
115         T0 = []
116     return R
117
118
119 # Find the highest correlation in R. The indices correspond to the correct key
120 # and the time at which the SubBytes round was processed.
121 def max_absolute(result):
122     highest_cor = 0
123     a = -1
124     b = -1
125     for row in range(K):
126         x = max(map(abs, result[row]))
127         if x > highest_cor:
128             highest_cor = x
129             a = row
130     for column in range(T_used):
131         if abs(result[a][column]) == highest_cor:
```

```
132                 b = column
133         return a, b, highest_cor
134 
135 
136     def main():
137         help = ""
138         for byte in range(16):
139             help += key[byte][0]
140         print help
141 
142 
143     if __name__ == "__main__":
144         D = 10000
145         T_used = 20000
146         T = 110000
147         K = 256
148         key = []
149         dic = np.load('traces.npz', mmap_mode='r')
150         data = dic['data']
151         traces = dic['traces'][:, :T_used]
152         keys = init_keys()
153         for byte in range(16):
154             v = hypo_inter_values(byte)
155             print "Hypothetical intermediate values computed."
156             h = LSB_hypo_power()
157             print "Hypothetical power consumptions computed."
158             result = dom()
159             print "Difference of Means computed."
160             (i, j, highest_cor) = max_absolute(result)
161             print "Byte " + str(byte) + " has been processed."
162             key.append((keys[i], j, highest_cor))
163         main()
```