Lab 2: Differential Power Analysis

1. Introduction

This tutorial will introduce you to the basics of the DPA (Differential Power Analysis)—a technique that exploits the dependency of the processed data on the power trace of the device to extract some secret information that would not be otherwise available. During the session you will learn how to process the power trace of the implementation of the AES encryption algorithm using an algebraic system (in our case MATLAB), create the power hypothesis, and extract the secret information and also how to measure the power consumption of the embedded system (smart card) in order to obtain the power traces.

2. Background

Differential Power Analysis (DPA) is a powerful method for breaking cryptographic systems. The method does not attack the algorithm itself, but the physical implementation of the algorithm. Therefore, even systems using modern strong ciphers like AES are vulnerable to such attacks, if proper countermeasures are not applied.

The DPA method uses the fact that every electronic system has power consumption. If you measure the power consumption of digital system, you will probably see the power trace like in Fig. 1 with its peaks on rising and falling edges of clock. If the digital system runs an encryption and if you run this encryption several times using various input data, you may notice slight variations in power traces, as shown in Fig. 1. These variations are caused by many factors (varying temperature, etc.), but one of them are varying processed (inner) data. DPA utilizes the fact that power consumption depends on processed data (e.g., number of ones and zeros in processed byte) to break the cryptographic system, i.e. deduce its secret key.

Before we start, please, download all necessary materials from the web. You will find the compressed archive at the course webpage in Canvas in Files/LABS/LAB2/DPA.zip. The compressed archive contains two folders. In the folder Analysis you will find files used in Sect. 5. You do not need to perform any measurement with an oscilloscope, as these measurement have been done or you. In the folder Analysis you will find two sets of measurements, one set for a known key 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff, and one set for an unknow key. These sets will be used in Sect. 5. You are also provided with sample codes in MATLAB that you can use in your program/script.

This lab is derived from the tutorial material “Differential Power Analysis for beginners”, Trudevice Training School 2014, and book chapter DOI 10.1007/978-3-319-44318-8_4.

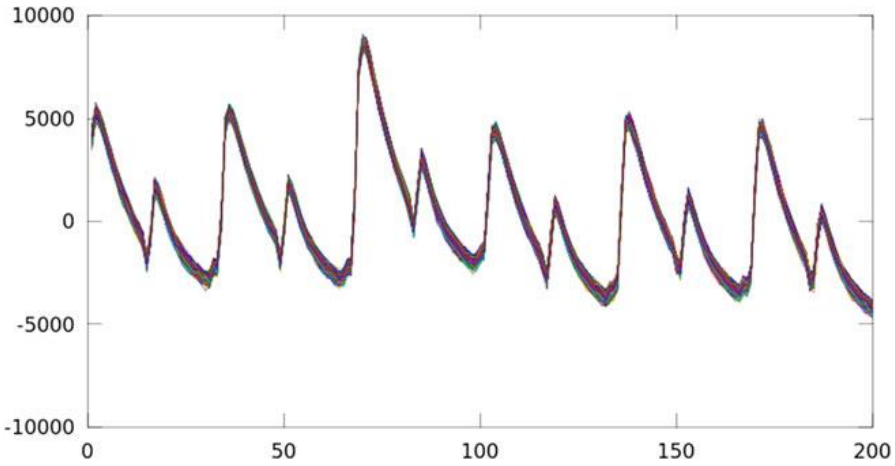

Figure 1: The figure shows 500 power traces in the same time interval of 200 samples. Each power trace is run for unique input data, power traces are overlapped [1].

3. Getting Started

In this lab, we will need to use a system suitable for numerical calculations. MATLAB from Mathworks is a system best-tailored for our needs (matrix operations with large matrices).

3.1. Downloading and Installing MATLAB

MATLAB® is a high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis, and numeric computation. Matlab installation is very simple and straightforward. The current stable version is 2017a. Please download the installation program and the associated license/activation keys from the *KTH program distribution* and follow the instructions on the website:

https://intra.kth.se/it-services/programvara/download/matlab

3.2. Useful MATLAB Commands

Here you find several useful commands. We are working in certain working directory where all working files and scripts (files .m) are placed. Almost all MATLAB objects are matrices. Column or row vector are special cases, however, generally we are working with n-dimensional arrays. Almost all numbers are of type double.

- *example 1 - Matrixes*

  ```
  % matrix creation
  a = [1,2,3;4 ,5 ,6;7 ,8 ,9]
  % we have defined the variable a, the result has been printed
  b = rand(100 ,100) ;
  ```

```
% semicolon (;) suppresses printing the result (important for huge data)
% showing part of a matrix b:
b(1:10,5:7)
% matrix multiplication (addition / subtract ion / division ) works :
c = [2,0,0; 0,2,0; 0,0,2]
a*c
% for entry-by-entry multiplication, we use .*
a.* c
```

- *example 2 - Vectors*

```
% Vectors are special cases of matrices
v = [1, 3, 5, 7] % row vector
v(1,:) % equivalent to v
v(1,3:4) % part of v
% transposition
v' % creates column vector
% special matrices
zeros (3,3)
ones (3,3)
eye (3,3)
rand (3,3)
% indexing by a vector
iv = [3 ,4 ,1 ,2]
v (iv )
% by indexing we can create originally not existing components
v
v( [ 1 , 1 ] , 1 : 3 )
v([ 1 , 1 , 1 ] , : )
v(ones (1 ,5) , : )
v'( : , ones (1 ,5) ) % works only in Octave
```

- *example 3 – Graph plot*

```
% graph plot
e = rand(1, 100) ;
plot ( e )
f = rand(1, 100) ;
hold on % adding the second trace into the graph
```

```
plot ( f )
% if x is a matrix:
plot (x) % plot s the s e t of traces by columns of x
plot (x ') % plots the  set of traces by rows of x ( using transposition )
```

- *example 4 – Cycles*

```
% how to write cycles
for i =1:10
  for j =1:20
    x(i)=bitxor(v(i) ,w(j) ) ;
  end
end
```

- *example 5 – Manipulating files*

```
% manipulating file
% open file for reading :
MyFile = fopen ( 'myFile.bin ' , ' r ' ) ;
% skip in Myfile from current position ( ' cof ' ) by Offset :
fseek ( MyFile , Of fset , ' cof ' ) ;
% read Number of uint8s to the vector Values (from the current position ) :
Values = fread (MyFile , Number , ' uint8 ' ) ;
% close MyFile:
fclose ( MyFile ) ;
% Manipulating text files
% open files for reading:
TextFile = fopen ( 'myTextFile.txt ' , ' r ' ) ;
% reading line from TextFi le:
Line = fgets ( TextFile ) ;
% reading 16 values from the Line according to the pat tern (like in C) :
[values , l ] = sscanf (Line, '%02x ' , 16) ;
```

- *example 6 – Printing data in hex-form*

```
% suppose the key i s stored here in the key array:
key=[1 ,2 ,3 ,4 ,5 ,6 ,7 ,8 ,9 ,1 ,2 ,3 ,4 ,5 ,6 ,7];
% to print it in hex−form run the following for−cycle:
for i =1:16
    fprintf ( 'Byte %d of the key is 0x%2.2X \ n ', i , key (i) ) ;
end ;
```

4. MATLAB as Tools for DPA

In this section, we introduce the basics of working at the Linux command line that are useful for this design project.

4.1. MATLAB—Using the Prepared Functions

The following code samples show, how to use the prepared functions (files) to speed up the key recovery process.

measurement.m	the code template for the key recovery process
myin.m	loads the content of the text files (plaintext.txt, ciphertext.txt) generated during the measurement
myload.m	loads the content of the binary files (traces.bin) generated during the measurement
mycorr.m	is used to calculate the correlation coefficient later during the recovery process

All the files are available in archive in the folder Analysis and should be placed into your MATLAB project directory. The following codes show in more detail how to load the appropriate data using the prepared functions and are all included in *measurement.m*.

- MATLAB code example 1 - loading the data for the known key case

```
%%%%%%%%%%%%%%%%%%%%
% LOADING the DATA %
%%%%%%%%%%%%%%%%%%%%

% modify following variables so they correspond
% your measurement setup
numberOfTraces = 200;
traceSize = 370000;
offset = 40000;
segmentLength = 50000;

% columns and rows variables are used as inputs
% to the function loading the plaintext/ciphertext
columns = 16;
rows = numberOfTraces;
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%
% Calling the functions %
%%%%%%%%%%%%%%%%%%%%%%%%%

% function myload processes the binary file containing the measured traces and
% stores the data in the output matrix so the traces (or their reduced parts)
% can be used for the key recovery process.
% Inputs:
%   'file' - name of the file containing the measured traces
%   traceSize - number of samples in each trace
%   offset - used to define different beginning of the power trace
%   segmentLength - used to define different/reduced length of the power trace
%   numberOfTraces - number of traces to be loaded
%

traces = myload('traces.bin', traceSize, offset, segmentLength, numberOfTraces);

% function myin is used to load the plaintext and ciphertext
% to the corresponding matrices.
% Inputs:
%   'file' - name of the file containing the plaintext or ciphertext
%   columns - number of columns (e.g., size of the AES data block)
%   rows - number of rows (e.g., number of measurements)
plaintext = myin('plaintext.txt', columns, rows);
ciphertext = myin('ciphertext.txt', columns, rows);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% EXERCISE 1 -- Plotting the power trace(s): %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Plot one trace (or plot the mean value of traces) and check that it is complete
% and then select the appropriate part of the traces (e.g., containing the first round).

% --> create the plots here <--
```

- MATLAB code example 2 - loading the data

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
% EXERCISE 2 -- Key recovery: %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Create the power hypothesis for each byte of the key and then
correlate
% the hypothesis with the power traces to extract the key.
% Task consists of the following parts:
%   - create the power hypothesis
%   - extract the key using the results of the mycorr function

% variables declaration
byteStart = 1;
byteEnd = 16;
keyCandidateStart = 0;
keyCandidateStop = 255;

% for every byte in the key do:
for BYTE=byteStart:byteEnd

    % Create the power hypothesis matrix (dimensions:
    % rows = numberOfTraces, columns = 256).
    % The number 256 represents all possible bytes (e.g.,
0x00..0xFF).
    powerHypothesis = zeros(numberOfTraces,256);
    for K = keyCandidateStart:keyCandidateStop
        % --> create the power hypothesis here <--
    end;

    % function mycorr returns the correlation coefficients matrix
calculated
    % from the power consumption hypothesis matrix powerHypothesis
and the
    % measured power traces. The resulting correlation coeficients
stored in
    % the matrix CC are later used to extract the correct key.
    CC = mycorr(powerHypothesis, traces);

    % --> do some operations here to find the correct byte of the key
<--

end;
```

4.2. More MATLAB Tips

- For xoring of values you may use the bitxor function. This function also performs bit-wise xor of vectors and matrices (of the same size).
- The average value (mean value) is calculated by the function mean. If a is a matrix, then mean(a) is a (row) vector of mean values of columns, while mean(a,2) is a (column) vector of mean values of rows (which is probably what we want).
- If you like to extend (copy) the column vector into matrix, use indexing (e.g., you like to extend vector b into matrix having 100 columns):
    1. By indexing: b_mat = b(:,ones(1,100));
    2. By replication: b_mat = repmat(b,1,100);
- You can use arrays SubBytes and byte_Hamming_weight (see the file tab.mat). Remember that the first index of an array is equal to 1, therefore you probably need to increment index by 1, e.g.,: a = SubBytes(x + 1); b = byte_Hamming_weight(a + 1); This works also for matrices (!) - if x is a matrix, then SubBytes applies to all its elements (the result is a matrix again).

5. Differential Power Analysis - Key Recovery

In this section, you are given the power consumption (traces) of a SmartCard. For each power trace you have a pair of the plaintext and the encrypted ciphertext. Therefore you have all the information you need, except of the secret key. It is the goal of the differential power analysis to extract the secret key using the mentioned traces, plaintext, ciphertext, and the knowledge of the encryption algorithm by creating the hypothesis of the power consumption and correlating it to the measured traces.

5.1. Method

The explanation of the method presented below can be found in the presentation dpa_Lisbon.pdf which is incuded in DPA.zip file or in the book [2], p. 119.

The method goes through the following steps:

1. Choose an intermediate value that depends on data and key
2. Measure the power traces while encrypting the data

3. Build a matrix of hypothetical intermediate values inside the cipher for all possible keys and traces

4. Using a power model, compute the matrix of hypothetical power consumption for all keys and traces

5. Statistically evaluate which key hypothesis best matches the measured power in each individual time.

The right key (part of the right key) is determined by key hypothesis → intermediate value → consumption, best correlating to actually measured consumption at some moment. Repeat the analysis for other parts of key, until you determine the whole key.

5.2. Schedule

You're recommended to proceed according to the following steps:

1. Plot one trace in the program you are using MATLAB. Check that it is complete.

2. Plot several traces (e.g., 1st, 10th, 50th). Check the alignment of traces (they overlay correctly, triggering works).

3. Select the appropriate part of the traces (e.g., containing the first round). Read in the appropriate number of traces.

4. Depending on your measurements, you may have to perform a correction of mean values (if your measurements "wander" in voltage over time). You can do so by subtracting from each trace its mean value.

5. Recover the secret key using the DPA with correlation coefficients using the method described in Sect. 5.1.

5.3. Training Sets

In folder Analysis you will find two sets of measurements. One set is for known key 00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff, the other set is for unknown key.

- Training Set for Known Key 00 11 22 33 44 55 66 77 88 99 Aa Bb Cc Dd Ee Ff

  To implement and debug your program/script, we provide testing traces of 200 AES encryptions (AES 128, with 10 rounds). We encrypted 200 plaintexts (file *plaintext-00112233445566778899aabbccddeeff.txt*), obtaining 200 ciphertexts (file *ciphertext-00112233445566778899aabbccddeeff.txt*. During encryptions we measured power traces (*traces-00112233445566778899aabbccddeeff.bin*). Each

trace has a length of 370 000 samples (in this case). Each sample is represented by 8 bit unsigned value (i.e., the length of the file is 370 000 bytes*200 traces = 74 MB).

If your program/script is correct, then you should reveal the key 00 11 22 33 44 55 66 77 88 99 AA BB CC DD EE FF.

- Training Set for Unknown Key

  If you are successful with the set above, you may try to recover the unknown key. We made 150 AES encryptions (AES 128, with 10 rounds).

  1. Files plaintext-unknown_key.txt and ciphertext-unknown_key.txt contain plaintexts and corresponding ciphertexts, which were produced by an AES encryption with an unknown key.
  2. File traces-unknown_key.bin stores power traces recorded during encryptions of above plaintexts.
  3. File traceLenght-unknown_key.txt contains information on trace length, i.e., 550000 samples in this case.

  You can easily check whether you found correct key. Just take any plaintext from the file plaintext-unknown_key.txt, encrypt it with the key you determined by the analysis, and compare the resulting ciphertext with a corresponding ciphertext from the file ciphertext-unknown_key.txt. If the ciphertexts match, you found the correct key.

6. Results

Please present your recovered key results to the lab assistant.

References:

[1] Introduction to Differential Power Analysis with Correlation Coefficients, J. Bucek, M. Novotny, CTU Prague, TRUEDEVICE'2014.

[2] “Power Analysis Attacks: Revealing the Secrets of Smart Cards” by Stefan Mangard, Elisabeth Oswald and Thomas Popp, Springer, 2007, ISBN: 978-0-387, available at KTH library