Practical Session: Differential Power Analysis for Beginners

Jiří Buček, Martin Novotný, and Filip Štěpánek

Abstract 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, 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.

The first part of the tutorial *Differential Power Analysis – Key Recovery* is aimed at explaining the creation of the power hypothesis and the use of algebraic systems.

The second part of the tutorial *DPA – measurement with an oscilloscope* covers the practical part of the exercise - the measurement of the power consumption using the PicoScope.

1 Introduction

Differential Power Analysis (DPA) is a powerful method for breaking the cryptographic system. The method does not attack the cipher, but the physical implementation of the cryptographic system. 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 a power consumption. If you measure the power consumption of digital system, you will probably see the power trace like in Figure 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

Jiří Buček, Martin Novotný, Filip Štěpánek
Czech Technical University in Prague, Faculty of Information Technology, Czech Republic
e-mail: {jiri.bucek, martin.novotny, filip.stepanek}@fit.cvut.cz

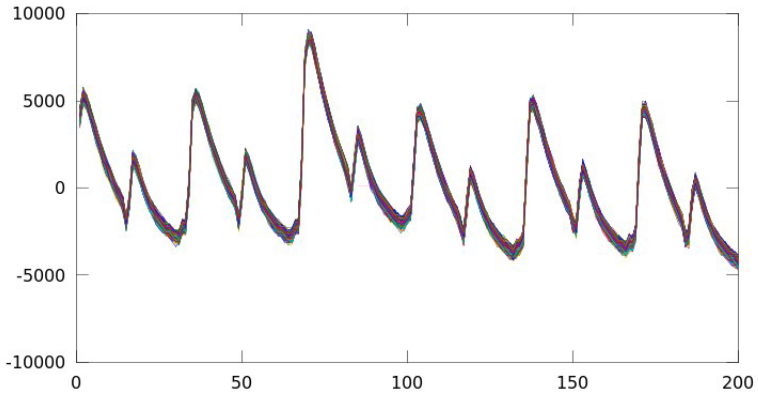


Fig. 1 The figure shows 500 power traces in the same time interval of 200 samples. Each power trace is run for unique input data, power traces are overlapped. Variations in power traces are caused by variations in processed data.

using various input data, you may notice slight variations in power traces, as shown in Figure 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.

To demonstrate the power of power analysis we prepared this tutorial for you. Before we start, please, download all necessary materials from the web. You will find the compressed archive at the address http://users.fit.cvut.cz/~novotnym/DPA.zip (the file has about 250 MB). Uncompressed materials can be found also at address http://users.fit.cvut.cz/~novotnym/DPA, which might be useful if you have problems to download the whole archive. The compressed archive contains two folders. In folder *Analysis* you will find files used in Section 2. Files used in Section 3 are available in folder *Measurement*.

You do not need to perform any measurement with an oscilloscope, as we have done these measurements for you. In 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 Section 2. You are also provided with sample codes in *Matlab* that you can use in your program/script.

However, if you are equipped with an osciloscope (e.g. PicoScope), you can make your own measurement. Several advices you will find in Section 3.

2 Differential Power Analysis – Key Recovery

At this point you are either given or were able to measure the power consumption (traces) of the SmartCard yourself. 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.

2.1 Method

We are not going to explain the method here. If you are not familiar with the method, you may find its explanation e.g. in the book [1], page 119, or you will find presentation *dpa_Lisbon.pdf* in downloaded materials.

To summarize the method, you shall go through 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* $\rightarrow$ *intermediate value* $\rightarrow$ *consumption*, best correlating to actually measured consumption at some moment. We repeat the analysis for other parts of key, until we determine the whole key.

2.2 Schedule of your work

We reccommend you to proceed according to the following steps:

1. Plot one trace in the program you are using (Matlab/Octave, Mathematica, etc.). 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. The method is summarized in Subsection 2.1.

2.3 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.

2.3.1 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`

2.3.2 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).

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. File *traces-unknown_key.bin* stores power traces recorded during encryptions of above plaintexts. 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.

2.4 Tools

We will use a system suitable for numerical calculations. *Matlab* seems to be a system best-tailored for our needs (matrix operations with large matrices). We also can use freeware alternative *Octave*, that is compatible with *Matlab* in its basic functions.

Mathematica is also one of alternatives. Mathematica can `Import` data in Matlab format (*.mat*).

You may also use other computer algebraic systems - your possible experience is welcome!

2.4.1 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 code snippets show in more detail, how to load the appropriate data using the prepared functions and are all included in the template *measurement.m*.

In case you are new to Matlab, you can see some basic examples in the Paragraph 2.4.2.

Matlab code example – loading the data

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

% modify following variables so they correspond
% your measurement setup
numberOfTraces = 200;
traceSize = 350000;

% modify the following variables to speed-up the measurement
% (this can be done later after analysing the power trace)
offset = 0;
```

```
13 segmentLength = 350000;
14 % for the beginning the segmentLength = traceSize
15
16 % columns and rows variables are used as inputs
17 % to the function loading the plaintext/ciphertext
18 columns = 16;
19 rows = numberOfTraces;
20
21 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
22 % Calling the functions %
23 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%
24
25 % function myload processes the binary file containing the
26 % measured traces and stores the data in the output matrix so
27 % the traces (or their reduced parts) can be used for the key
28 % recovery process.
29 % Inputs:
30 %    'file' - name of the file containing the measured traces
31 %    traceSize - number of samples in each trace
32 %    offset - used to define different beginning of the power
       trace
33 %    segmentLength - used to define different/reduced length of
       the power trace
34 %    numberOfTraces - number of traces to be loaded
35 %
36 % To reduce the size of the trace (e.g., to speed-up the
37 % computation process) modify the offset and segmentLength
38 % inputs so the loaded parts of the traces correspond to the
39 % trace segment you are using for the recovery.
40 traces = myload('traces.bin', traceSize, offset, segmentLength,
       numberOfTraces);
41
42 % function myin is used to load the plaintext and ciphertext
43 % to the corresponding matrices.
44 % Inputs:
45 %    'file' - name of the file containing the plaintext or
       ciphertext
46 %    columns - number of columns (e.g., size of the AES data block
       )
47 %    rows - number of rows (e.g., number of measurements)
48 plaintext = myin('plaintext.txt', columns, rows);
49 ciphertext = myin('ciphertext.txt', columns, rows);
50
51 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
52 % EXERCISE 1 -- Plotting the power trace(s): %
53 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
54 % Plot one trace (or plot the mean value of traces) and check
55 % that it is complete and then select the appropriate part of
56 % the traces (e.g., containing the first round).
57
58 % --> create the plots here <--
```

Matlab code example – using the correlation coefficients

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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 (0x00..0xFF).
    powerHypothesis = zeros(numberOfTraces,256);
    for K = keyCandidateStart:keyCandidateStop
        % --> create the power hypothesis here <--
    end;

    % function mycorr returns the correlation coeficients 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 proper operations here      <--
    % --> to find the correct byte of the key <--

end;
```

2.4.2 Matlab for beginners

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 (this is a comment)
```

```
% 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/subtraction/division) works:
c = [2,0,0;0,2,0;0,0,2]
a * c
% for entry-by-entry multiplication, we use .*
a .* c
```

Vectors are special cases of matrices

```
% 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
```

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)             % plots the set of traces by columns of x
```

```
plot(x')            % plots the set of traces by rows of x (using
    transposition)
```

Cycles

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

Manipulating files

```
% Manipulating files

% open file for reading:
MyFile = fopen ('myFile.bin', 'r');
% skip in Myfile from current position ('cof') by Offset:
fseek(MyFile, Offset, '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 file for reading:
TextFile = fopen('myTextFile.txt','r');
% reading line from TextFile:
Line = fgets(TextFile );
% reading 16 values from the Line according to the pattern (like
    in C):
[values, l] = sscanf(Line, '%02x ', 16);
```

Printing data in hex-form

```
% suppose the key is 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));
```

```
7 end;
```

2.4.3 Matlab/Octave 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).

2.4.4 Mathematica – Tips

Mathematica can import files in Matlab format (matrix format, *.mat*) using function `Import`. Function `Import` may be highly memory demanding, as it always imports the whole file.

```
1 $HistoryLength = 0; (*saving the memory*)
2 SetDirectory[NotebookDirectory[]]
3 NUMBER = 500;
4 t = Import["traces-part.mat"][[1]][[1 ;; NUMBER]];
```

```
1 {MemoryInUse[], MaxMemoryUsed[]} (*checking the occupied memory*)
2 ListLinePlot[t[[1 ;; 100, 1 ;; 100]]]
```

Use the function `Mean` for elimination of a systematic error of measurement (different DC component between traces). `Mean` applied to a matrix returns a vector of means of columns. However, we need means of rows, i.e. we have to use `Mean[Transpose[...]]`.

Fig. 2 Connect the card reader to your computer and insert the SmartCard into the reader.

3 DPA – measurement with an oscilloscope

First we will set up a basic measurement of the smart card consumption. We will use JSmartCard Explorer to communicate with the card, and PicoScope 6 GUI to establish basic parameters of the measurement. Then, we will switch to a separate program that will control both the card and the oscilloscope and will perform a series of measurements needed for the DPA attack.

3.1 Preparation of the measurement

1. Connect the card reader to your computer and insert the SmartCard into the reader, as shown in Figure 2.
2. Run *JSmartCard Explorer* from Primiano Tucci [2]. (In Java. Compiled JAR file you find either on web [3] or in file *JSmartCardExplorer.jar* in downloaded archive).
   - Press the *Connect* button to connect to the SmartCard. (Status should be green.)
   - Fill-in the fields *Class* (80), *INS* (60), *P1* (00), *P2* (00), *Data IN*, and *Le* (10) as shown in Figure 3 (all in HEX).
   - Press *Send*. The card should run AES encryption over the entered data and return the ciphertext, as shown in Figure 3.
3. To measure the card power consumption, we will be using Picoscope 5204 (and 5203) and two oscilloscope probes. Connect the probes to channels A (blue, trigger), and B (red, trace measurement).
4. Remove the card from the reader and insert it into the measuring adaptor (green PCB), then insert the measuring adaptor into the reader. See Figure 4
5. Connect the Picoscope probes to the measuring adaptor. Unlike to Figure 5, set the trigger probe (channel A, blue) to the X10 position and the measurement probe (channel B, red) to the X1 position.
6. Connect the Picoscope to a free USB port of your computer (if not already connected).
7. For the measurement you need the following software.

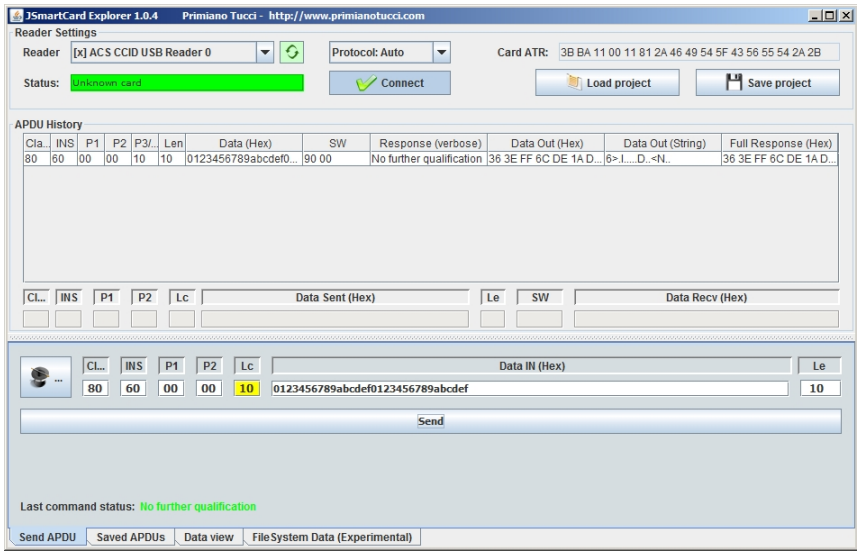


Fig. 3 Fill-in proper fields and press *Send*.

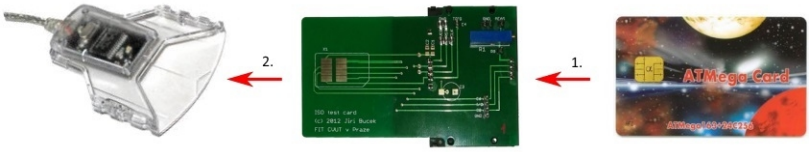


Fig. 4 Insert card into the measuring adaptor (green PCB), then insert the measuring adaptor into the reader.

- PicoScope 6 software with drivers. You can download it from [4]. You can find it also in downloaded materials as a file *PicoScope6_r6_8_11.exe*.
- Software Development Kit. The relevant files should be included in the Visual Studio project below. If not, you can download the SDK from the web [5] or you may find it in downloaded materials as a file *PS5000sdk_r10_5_0_32.zip*.
- Library for working with smart cards (*WINSCard.lib*). This should be included in the installation of Visual Studio. (It is a part of Microsoft Windows SDK.)

8. Run the PicoScope 6 program
9. You should make following settings:

- *Timebase*: 500us/Div, x1 (zoom), 1 MS (samples)
- *Channel A*: +-1V DC
- *Channel B*: +-1V DC
- *Trigger*: Auto (after tuning the settings, switch to Repeat)
- *Trigger Event*: Simple Edge, Rising

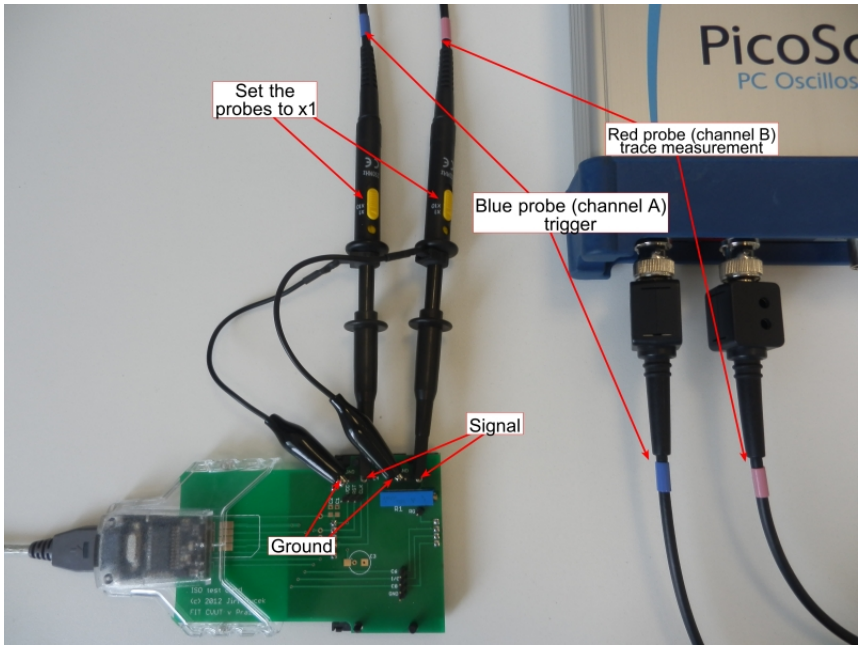


Fig. 5 Connect the Picoscope probes to the measuring adaptor. Set the trigger probe (channel A, blue) to the X10 position and the measurement probe (channel B, red) to the X1 position.

- *Trigger Channel*: A
- *Trigger Threshold*: 200 mV

Warning: These settings may need to be adjusted according to the particular card and other circumstances.

10. Set the *Single* measurement at the oscilloscope, and send data to the card using *JSmartCard Explorer* from Primiano Tucci. You should see a waveform like in Figure 6.
11. Display the Properties panel by right-clicking somewhere in the window and selecting View Properties (see Figure 7).
12. From the Properties panel remember the following values:

    - Sample interval,
    - Sample rate,
    - No. samples.

    We will need these values later, when setting the measurement program.

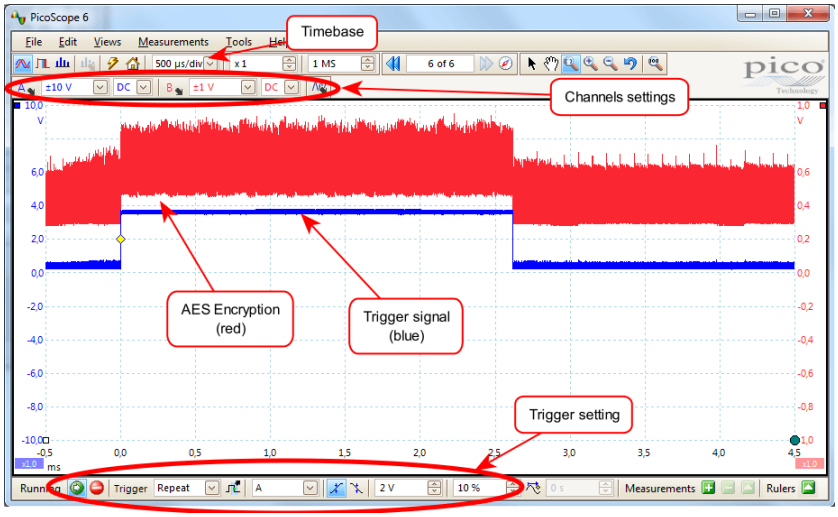


Fig. 6 Powertrace of one encryption.

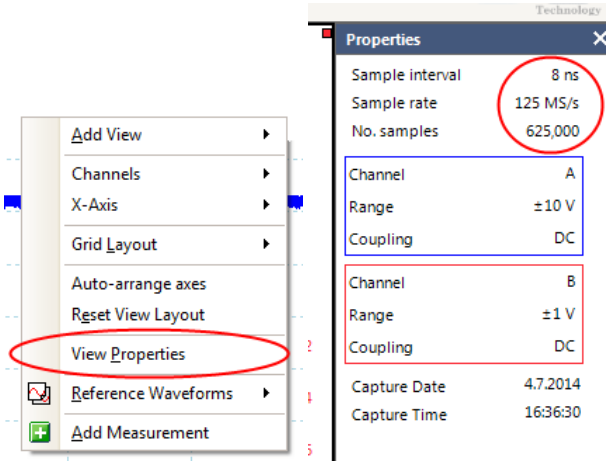
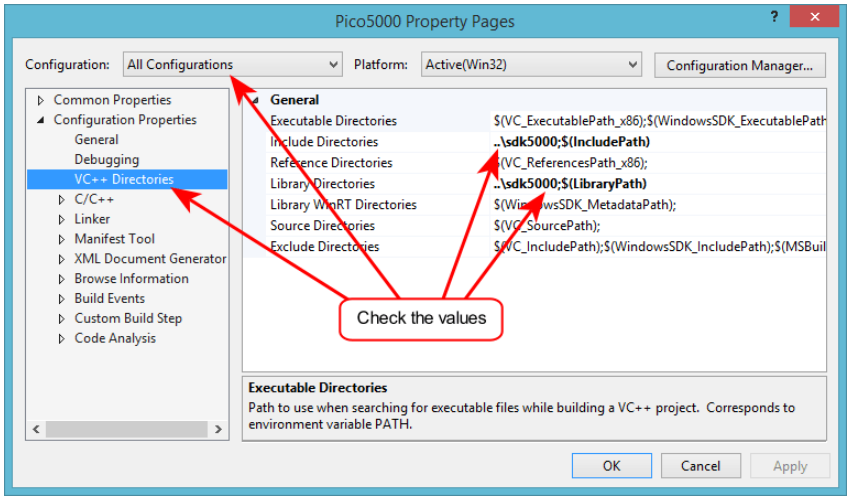


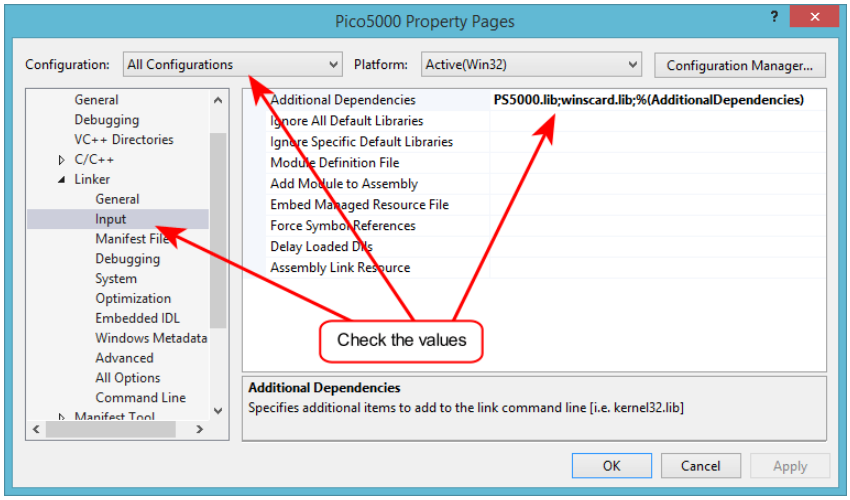
Fig. 7 Menu Properties of PicoScope program.

3.2 Compilation of program for measurement

At this stage, you should have verified that the SmartCard works correctly (responds to the command for AES encryption), and that the signals from the card look reasonable. Press *Disconnect* or close *JSmartCard Explorer*, and close *PicoScope 6 GUI*. We will use a separate program to control both the SmartCard reader and the oscilloscope.

For measurement it is necessary to adjust and compile C++ program stored in an archive *Pico5000.zip*. Zip file contains source files and Microsoft Visual Studio





project. After extracting the archive and opening the project in Microsoft Visual Studio you have to check the following settings in project properties:

1. Paths to include and library directories, see Figure 1.
2. Paths to additional dependencies, see Figure 2
3. In source file *main.cpp* set up the measuring channels, trigger voltage level, and number of measurements.

Compile the program (*Build* → *Build Solution*). Before running the program do not forget:

- to disconnect the card in JSmartCard Explorer and
- quit the PicoScope program,

otherwise the card and/or the PicoScope would be occupied, hence the measuring program will not be able to connect to it.

Measured data are in file *traces.bin*, plaintext and cipher text in files *plaintext.txt* and *ciphertext.txt* and length of one measurement is stored in file *traceLength.txt*.

Now you have measured data to be used for DPA.

References

1. Mangard, S. and Oswald, E., Popp, T.: Power Analysis Attacks: Revealing the Secrets of Smart Cards. Springer, US (2008)
2. Primiano Tucci: JSmartCardExplorer. https://www.primianotucci.com/os/smartcard-explorer. Cited 02 Mar 2016
3. Primiano Tucci: JSmartCardExplorer. http://downloads.sourceforge.net/jsmartcard/JSmartCardExplorer.jar. Cited 02 Mar 2016
4. Picotech: PicoScope 6 software with drivers. http://downloads.picotech.com/winxp/PicoScope6_r6_8_11.exe. Cited 02 Mar 2016
5. Picotech: Software Development Kit. http://dl.picotech.com/drivers/PS5000sdk_r10_5_0_32.zip. Cited 02 Mar 2016