EE049017: Hardware Systems Security

Homework Assignment 2

Faculty of Electronic Engineering, Technion.

Submitted by:

#	Name	Id	email
Student 1	Kfir Girstein	316541218	kfirgirstein@campus.technion
Student 2	Alon Berkenstadt	205710205	alon.b@campus.technion.ac.il

Contents

The answers notebooks can be obtained by cloning the HW repo Hardware_Security_HW_049017 (https://github.com/kfirgirstein/Hardware_Security_HW_049017/tree/main/HW or viewed in your browser by using nbviewer. and a notebook nbviewer (https://nbviewer.jupyter.org/github/kfirgirstein/Hardware_Security_HW_04901

Part One - Finding the Key from a Synthetic Power Model

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import sys
```

This part aims to generate for a particular key (as shown in the attached AES_GenPowerProfile.c file)

```
{0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x15, 0x88,
0x09, 0xcf, 0x4f, 0x3c}
```

A collection of power traces that contain for each message with 32 bytes

- The Ciphertext can be found in the CIPHERFILE.dat file on the site
- In the state at the entrance to round 9, you can calculate using inverse pomegranates in the DoM_actual_wrapper.py file (requires adjustments)
- It is required to create two tables, one for calculating the power based on HW and the other while relying on HD
  - HW - how many "1 "s are in the state to end each Round
  - HD - how many bits were changed between the state at the end of each stage, compared to the value of the state in the previous stage (the initial state (not in the table) is the input)

At the end of this stage you have "in your hands" two matrices with the help of which we can predict the key with which we have encrypted all the messages (place the same key)

```
{ 0xd0, 0x14, 0xf9, 0xa8, 0xc9, 0xee, 0x25, 0x89, 0xe1, 0x3f, 0x0c, 0xc8,
0xb6, 0x63, 0x0c, 0xa6}
```

Here is an extension of the key for round 9, we would like to get the first key of this step

In [2]:
```
AVICHART_PATH = "./Resources/AVICHART.dat"
SAMPLEFILE_PATH = "./Resources/SAMPLEFILE.dat"
SAMPLEFILE_HD_PATH = "./Resources/SAMPLEFILE_HD.dat"
SAMPLEFREQFILE_PATH = "./Resources/SAMPLEFREQFILE.dat"
```

In [3]:
```
HD_table = []
HW_table = []
C_P_table = []
with open(AVICHART_PATH,"r") as fp:
    for line in fp:
        temp = line.split()
        C_P_table += [[temp[0],temp[1]]]
        HW_table += [temp[2:14]]
        HD_table += [temp[14:]]
display(len(C_P_table))
```

```
5000
```

In [4]:
```
display(pd.DataFrame(np.column_stack((C_P_table, HW_table)), columns=["PT","CT"] + [f"R{i}" for i in range(1,13)]))
```

	PT	CT	R1	R2	R3
0	67c6697351ff4aec29cdbaabf2fbe346	9c8a50cdfcfd9160f5d421bf8fd57295	5	3	4
1	66320db73158a35a255d051758e95ed4	4b63abca487f7e1c95c234d9664f7681	4	4	5
2	70e93ea141e1fc673e017e97eadc6b96	970eb501c8268b0c3ae26fda8d91277a	3	5	4
3	021afe43fbfaaa3afb29d1e6053c7c94	f46f2d4ecc3423f1641d50c0470dd72d	1	3	5
4	05eff700e9a13ae5ca0bcbd0484764bd	0a8e754595ea0142cab17e20c060e8bb	2	4	3
...	...	...	...	...	...
4995	b2ab553190c00931918eb487fe270135	e83f249ff9ff4dda57a98ac42299cf1a	4	4	6
4996	1f63f8af2302e0b590943c8fbc3ec449	42ea5ebdf7ad3fcc41e0fd542a5fb760	5	3	0
4997	9fcbaec3cd8e785d23b4ecdff2b02899	cf3812ddff18d8ab97ea1f8c06d08f3b	6	4	6
4998	6d866c3a15e498389884178b353f24bb	ca0f78ffb158d13313c5a8d08f06a4fa	5	3	3
4999	c1d652d6baea0e536f25dea465025f08	38e5628bf5313509bc4d51c3f860c07d	3	5	6

5000 rows × 14 columns

In the table above, in each round and for each trace, we use the Hamming weight model to estimate the power that is calculated for that round.

In [5]:
```
display(pd.DataFrame(np.column_stack((C_P_table, HD_table)), column
s=["PT","CT"] + [f"R{i}" for i in range(1,12)]))
```

	PT	CT	R1	R2	R3
0	67c6697351ff4aec29cdbaabf2fbe346	9c8a50cdfcfd9160f5d421bf8fd57295	4	5	6
1	66320db73158a35a255d051758e95ed4	4b63abca487f7e1c95c234d9664f7681	4	5	5
2	70e93ea141e1fc673e017e97eadc6b96	970eb501c8268b0c3ae26fda8d91277a	4	5	4
3	021afe43fbfaaa3afb29d1e6053c7c94	f46f2d4ecc3423f1641d50c0470dd72d	4	4	7
4	05eff700e9a13ae5ca0bcbd0484764bd	0a8e754595ea0142cab17e20c060e8bb	4	5	4
...	...	...	...	...	...
4995	b2ab553190c00931918eb487fe270135	e83f249ff9ff4dda57a98ac42299cf1a	4	6	2
4996	1f63f8af2302e0b590943c8fbc3ec449	42ea5ebdf7ad3fcc41e0fd542a5fb760	4	3	5
4997	9fcbaec3cd8e785d23b4ecdff2b02899	cf3812ddff18d8ab97ea1f8c06d08f3b	4	2	5
4998	6d866c3a15e498389884178b353f24bb	ca0f78ffb158d13313c5a8d08f06a4fa	4	4	3
4999	c1d652d6baea0e536f25dea465025f08	38e5628bf5313509bc4d51c3f860c07d	4	5	5

5000 rows × 13 columns

In the table above, in each round and for each trace, we use the Hamming distance model to estimate the power that is calculated for that round.

After creating the tables, we will begin the "prediction" process of the key, using the DOM method learned in class. The method works as follows (suppose here we want to guess the value of the first byte of the key, using the right bit (LSB) of the byte)

In [6]:
```
samples = []
with open(SAMPLEFILE_PATH,"r") as fp:
    for line in fp:
        samples += [np.array([int(i) for i in line.split()])]
display(len(samples))
```

```
256
```

In [7]:
```
samples_hd = []
with open(SAMPLEFILE_HD_PATH,"r") as fp:
    for line in fp:
        samples_hd += [np.array([int(i) for i in line.split()])]
display(len(samples_hd))
```

```
256
```

In [8]:
```
sample_freqs = []
with open(SAMPLEFREQFILE_PATH,"r") as fp:
    for line in fp:
        sample_freqs += [int(line.split()[0])]
display(len(sample_freqs))
```

```
256
```

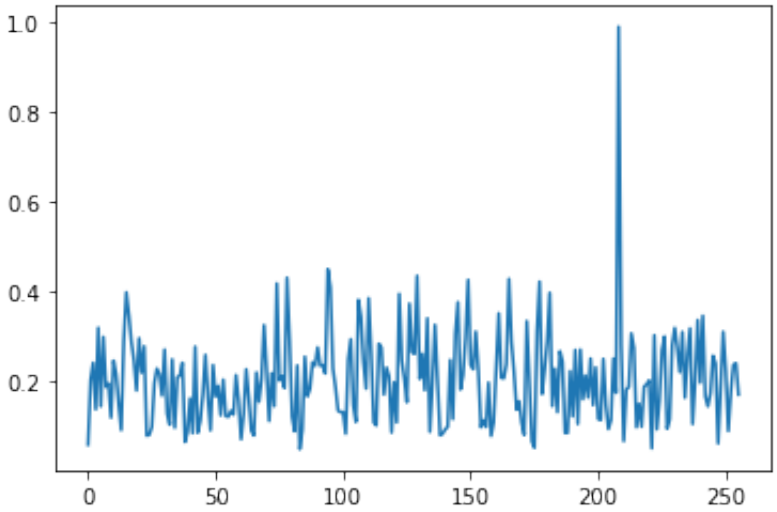
- The Ciphertext can be found for all traces in the table (third column, CT). Hundreds or thousands of footprints should be taken
- For each byte in Ciphertext we will go over all the possible values of the selected byte of the developer
  - Using the inverse functions (also given in the file) calculate what was the value of the state which was used as input to the phase
  - Based on the value of select bit, suppose we have chosen the LSB of the byte, we will sum their power value, at their absolute value, or to bin0 or bin1. For all 12 sampling points
  - At the end of the transition on all the encrypted messages, we will get two vectors which should normalize them according to the number of messages used in each bin
  - The normal vectors we subtract from each other and the absolute value of the differences, at the various points, is used to select the key, using one of the following three cryorions
    - The key used to create the largest difference for the selected byte
    - The key used to create the largest difference for one of the sampling points (vector point)
    - The key used to create the large average difference

In [9]:
```
InvSbox = (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA
3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x4
3, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x9
5, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA
2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5
C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x4
6, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x5
8, 0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xB
D, 0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xC
F, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x3
7, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x6
2, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC
0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x1
0, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7
A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xB
B, 0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x1
4, 0x63, 0x55, 0x21, 0x0C, 0x7D,
)

# TARGET_BYTE = 4
BIT_MASK = 2**5
HW_max_list = []
HD_max_list = []
keys_guesses = [i for i in range(256)]
```

Find key by Hamming Weight

In [10]:
```
key = -1
_max = 0
for key_guess in keys_guesses:
    p_0 = np.zeros(12) #The differences are averages of LSB = 0
    n_0 = 0
    p_1 = np.zeros(12) #The differences are averages of LSB = 1
    n_1 = 0
    for idx in range(256):
        select_bit = 0 if (InvSbox[key_guess ^ idx] & BIT_MASK == 0
) else 1
        if select_bit == 0:
            p_0 += samples[idx]
            n_0 += sample_freqs[idx]
        else:
            p_1 += samples[idx]
            n_1 += sample_freqs[idx]

    if (n_0 != 0 and n_1 != 0):
        dom = np.abs(p_0/n_0 - p_1/n_1)
        dom_max = np.max(dom)
        HW_max_list.append(dom_max)

        if (dom_max > _max):
            _max = dom_max
            key = hex(key_guess)

    else:
        print(str(idx) + " Failed")
```

In [11]:
```
plt.plot(keys_guesses,HW_max_list)
plt.show()
print(f"Result key is {key} or {int(key,16)}")
```

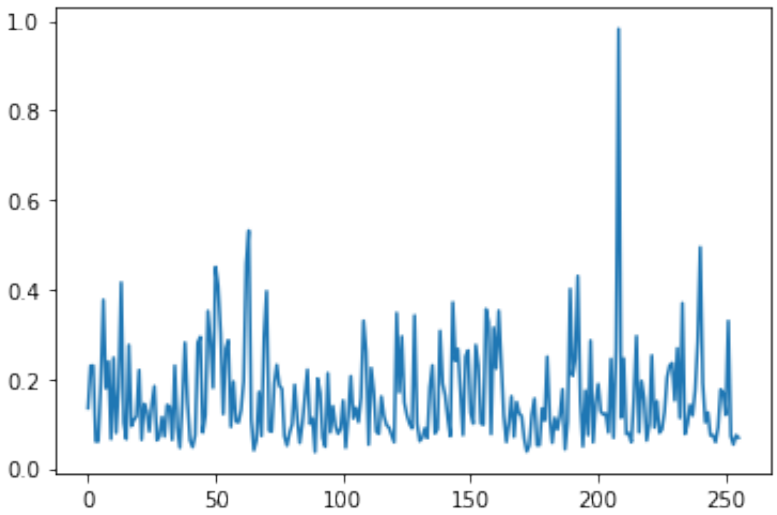



```
Result key is 0xd0 or 208
```

Find key by Hamming Distance

In [12]:
```
key = -1
_max = 0
for key_guess in keys_guesses:
    p_0 = np.zeros(11)
    n_0 = 0
    p_1 = np.zeros(11)
    n_1 = 0
    for idx in range(256):
        select_bit = 0 if ((InvSbox[key_guess ^ idx] & BIT_MASK) ^
(idx & BIT_MASK) == 0) else 1
        if select_bit == 0:
            p_0 += samples_hd[idx]
            n_0 += sample_freqs[idx]
        else:
            p_1 += samples_hd[idx]
            n_1 += sample_freqs[idx]

    if (n_0 != 0 and n_1 != 0):
        dom = np.abs(p_0/n_0 - p_1/n_1)
        dom_max = np.max(dom)
        HD_max_list.append(dom_max)

        if (dom_max > _max):
            _max = dom_max
            key = hex(key_guess)

    else:
        print(str(idx) + " Failed")
```

In [13]:
```
plt.plot(keys_guesses,HD_max_list)
plt.show()
print(f"Result key is {key} or {int(key,16)}")
```




```
Result key is 0xd0 or 208
```

Conclusions

From our results, a peak can be seen in the two graphs, one for each model. The above results present the expected value. To be more precise, we were able to get the key for round number 9 (as shown at the beginning of the question).

```
{ 0xd0, 0x14, 0xf9, 0xa8, 0xc9, 0xee, 0x25, 0x89, 0xe1, 0x3f, 0x0c, 0xc8,
0xb6, 0x63, 0x0c, 0xa6}
```

That is, with the help of power calculations of the AES algorithm, and with the help of the DoM methodology, we were able to predict the first byte of the key for the 9th round (**0xd0**).

- Please note that we did not consider our code in the mix columns operation and if so we managed with all the steps to recover the key. In our opinion, since this operation is not performed in the last step therefore there is enough "correlation" between the observations and the model.

For example the following code, will calculate the whole key for round 9

`Get_all_key.sh` :

```
gcc ./AES_GenPowerProfile.c -o AES_GenPowerProfile -lncurses
for i in $(seq 0 15); do
    # OUT=(`./AES_GenPowerProfile $i`)
    OUT=`./AES_GenPowerProfile $i`
    if [ $i == 0 ]
    then
        echo "True Round Key:    $OUT"
        printf "Extracted Key HW:  "
    fi
    # echo "Exp#$i - extracting byte #$i of the key";
    # echo "${OUT[${i}]}"
    EX=`python ./part1.py HammingWeight | cut -c3-`
    printf "%02s " $EX
done
printf "\n"
for i in $(seq 0 15); do
    ./AES_GenPowerProfile $i > /dev/null
    if [ $i == 0 ]
    then
        printf "Extracted Key HD:  "
    fi
    EX=`python ./part1.py HammingDistance | cut -c3-`
    printf "%02s " $EX
done
printf "\n"
```

Note that `part1.py` is a script which is customized according to the notebook

After running the script the following results were obtained:

```
True Round Key:    `d0 14 f9 a8 c9 ee 25 89 e1 3f 0c c8 b6 63 0c a6`

Extracted Key HW:  `d0 14 f9 a8 c9 ee 25 89 e1 3f 0c c8 b6 63 0c a6`

Extracted Key HD:  `d0 14 f9 a8 c9 ee 25 89 e1 3f 0c c8 b6 63 0c a6`
```

In the same way, and with the help of the AES algorithm we can continue the investigation and even recover the whole key

Extra

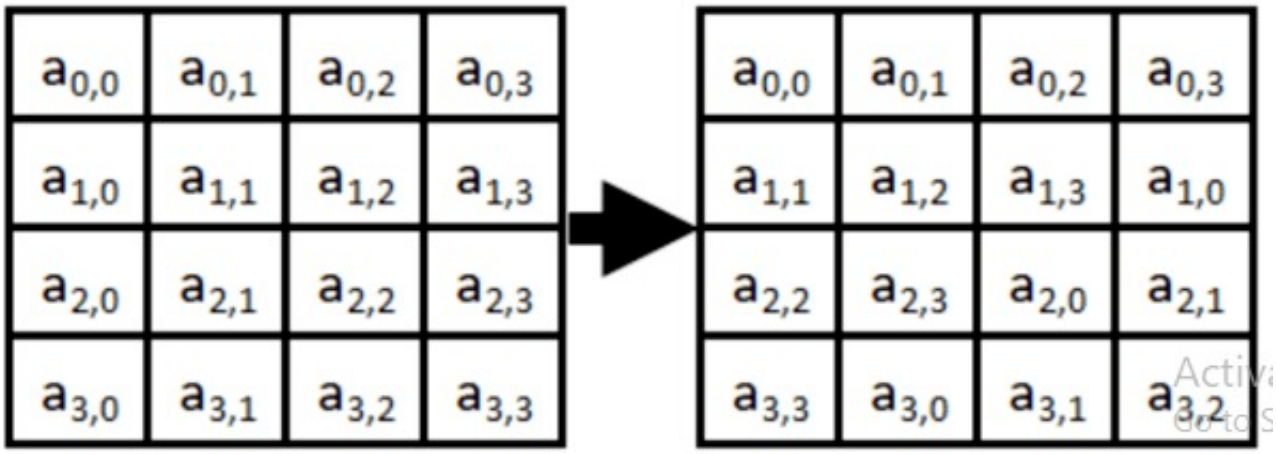
- As part of the study we tried to test whether Shift Rows activity affects key recovery. You can see the results here:

  True Round Key: `d0 14 f9 a8 c9 ee 25 89 e1 3f 0c c8 b6 63 0c a6`

  Extracted Key HW: `d0 b1 91 6a c9 7a 5a  a e1 ee 90 4e 81 5e 7e 6e`

  Extracted Key HD: `d0 5c fa 16 c9 1d f1 6b e1 22 7b cf b6 5e 1f ac`

  It can be seen that the bytes that are replaced in the Shift Rows were indeed affected and therefore we were unable to restore them. In contrast, the bytes in the first row can be seen as we were able to restore (for the most part). Therefore, we can conclude that canceling the Shift Rows operation does indeed affect our results



Finding the Key Using TRACES Sampled from Real Machines DPA

In [1]:
```
import csv
import numpy as np
import os
import pandas as pd
import matplotlib.pyplot as plt
```

For this part we use real traces which were taken from FPGA which performed encryption using AES The database for all samples at the time of the last round can be found in the file
"_DATA1_keyset_9_attack.csv"

In [2]:
```
KEYSET_FILE_PATH  = "./Resources/_DATA1_keyset_9_attack.csv"
DOM_OUT_PATH = "./Resources/DoM_Ex_Sample.dat"
```

In [3]:
```
wstart = 10
wstop = 1999
wlen = wstop-wstart
InvSbox = (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA
3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x4
3, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x9
5, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA
2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5
C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x4
6, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x5
8, 0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xB
D, 0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xC
F, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x3
7, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x6
2, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC
0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x1
0, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7
A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xB
B, 0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x1
4, 0x63, 0x55, 0x21, 0x0C, 0x7D,
)
```

Section 1

Perform DoM based on real data

In [4]:
```
dom_arr = np.zeros((256,wlen),dtype='float')
sarr_a = np.array([0]*wlen)
sarr_b = np.array([0]*wlen)
ntraces_a = 0
ntraces_b = 0
number_of_traces = 2000
```

In [5]:
```
csv_reader_data = []
with open(KEYSET_FILE_PATH) as csv_file:
        temp_csv_reader = csv.reader(csv_file, delimiter=',')
        for row in temp_csv_reader:
            csv_reader_data.append(row)
N = (len(csv_reader_data[0])-2)
print(f"Total samples:{len(csv_reader_data)} and N={N}")
```

```
Total samples:8939 and N=2001
```

In [6]:
```
#display(pd.DataFrame(csv_reader_data, columns=["PT","CT"] + [f"T{i
}" for i in range(1,N+1)]))
```

In [10]:
```
print_every = 10
if os.path.exists(DOM_OUT_PATH):
    with open(DOM_OUT_PATH,"r") as fp:
        for KeyGuess,line in enumerate(fp):
            if (KeyGuess % print_every == 0):
                print("Processing: "+ hex(KeyGuess))
            dom_arr[KeyGuess] = line.split()
    print("DoM computation Complete")
else:
    with open(DOM_OUT_PATH,"w") as fp:
        for KeyGuess in range(0,256):
            if (KeyGuess % print_every == 0):
                print("Processing: "+ hex(KeyGuess))
            dofmean=np.zeros(wlen,dtype='float')
            csv_reader = csv_reader_data
            for row in csv_reader_data:
                ct = int(row[1],16)
                ct_temp=ct>>120

                # write the DoM computation code here
                # Update the variable sarr_a and sarr_b depending
                # upon the lsb of the hypothetical leakage value
                ################ My Code ########################

                BIT_MASK = 2**0
                r9_temp = InvSbox[ct_temp ^ KeyGuess]
                select_bit = 0 if (r9_temp & BIT_MASK == 0) else 1
```

```
                ### Sum traces power by bit value
                if select_bit == 0:
                    sarr_a += np.array([int(row[j],16) for j in ran
ge(wstart,wstop)])
                    ntraces_a += 1
                else:
                    sarr_b += np.array([int(row[j],16) for j in ran
ge(wstart,wstop)])
                    ntraces_b += 1

                ################################################
            # Update the valriable marr_a and marr_b with the mean
of sarr_a
            # and sarr_b and compute the DoM value in the variable
dofmean
            marr_a = 1.0 * sarr_a / ntraces_a
            marr_b = 1.0 * sarr_b / ntraces_b
            dofmean = np.abs(marr_a-marr_b)

            dom_arr[KeyGuess] = dofmean
            fp.write(' '.join(map(str, dofmean)) + "\n")
            ntraces_a=0
            ntraces_b=0
            sarr_a = np.array([0]*wlen)
            sarr_b = np.array([0]*wlen)
            del dofmean
    print("DoM computation Complete")
```

```
Processing: 0x0
Processing: 0xa
Processing: 0x14
Processing: 0x1e
Processing: 0x28
Processing: 0x32
Processing: 0x3c
Processing: 0x46
Processing: 0x50
Processing: 0x5a
Processing: 0x64
Processing: 0x6e
Processing: 0x78
Processing: 0x82
Processing: 0x8c
Processing: 0x96
Processing: 0xa0
Processing: 0xaa
Processing: 0xb4
Processing: 0xbe
Processing: 0xc8
Processing: 0xd2
Processing: 0xdc
Processing: 0xe6
Processing: 0xf0
Processing: 0xfa
DoM computation Complete
```

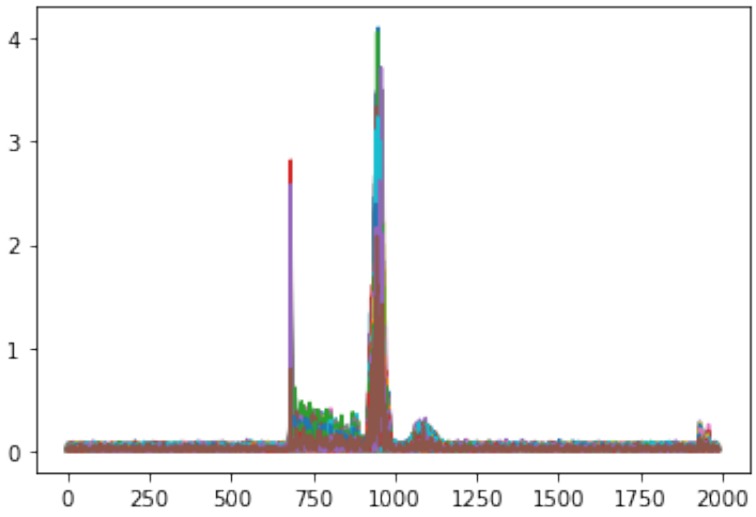
In other words we created a table in which for each column (ie an approximate value of a key) find the correlation coefficient of the row with each of the rows of the table from the file

In [11]:
```
df = pd.DataFrame(dom_arr, columns=[f"T{i}" for i in range(1,wlen+1
)]).transpose()
display(df)
```

	0	1	2	3	4	5	6	7	
T1	0.008203	0.031416	0.031150	0.006729	0.026068	0.030651	0.005018	0.002068	0.0
T2	0.024229	0.022270	0.007045	0.010774	0.012368	0.008872	0.004974	0.023640	0.0
T3	0.010433	0.004152	0.019493	0.031393	0.018453	0.023216	0.010724	0.006057	0.0
T4	0.018925	0.030998	0.023461	0.040337	0.011969	0.002295	0.014775	0.014135	0.0
T5	0.036331	0.014211	0.033703	0.013914	0.018647	0.015287	0.044271	0.020926	0.0
...	...	...	...	...	...	...	...	...	
T1985	0.017962	0.016210	0.008430	0.010222	0.046686	0.012641	0.021807	0.039347	0.0
T1986	0.011926	0.037411	0.002673	0.005774	0.039436	0.018284	0.031276	0.019076	0.0
T1987	0.017373	0.033190	0.013643	0.016629	0.000826	0.026325	0.042299	0.027543	0.0
T1988	0.007473	0.030493	0.027612	0.035141	0.003909	0.016697	0.046329	0.004417	0.0
T1989	0.014365	0.045118	0.023715	0.020126	0.027955	0.056721	0.054740	0.022458	0.0

1989 rows × 256 columns

In [12]:
```
plt.plot(dom_arr.transpose())
plt.show()
```

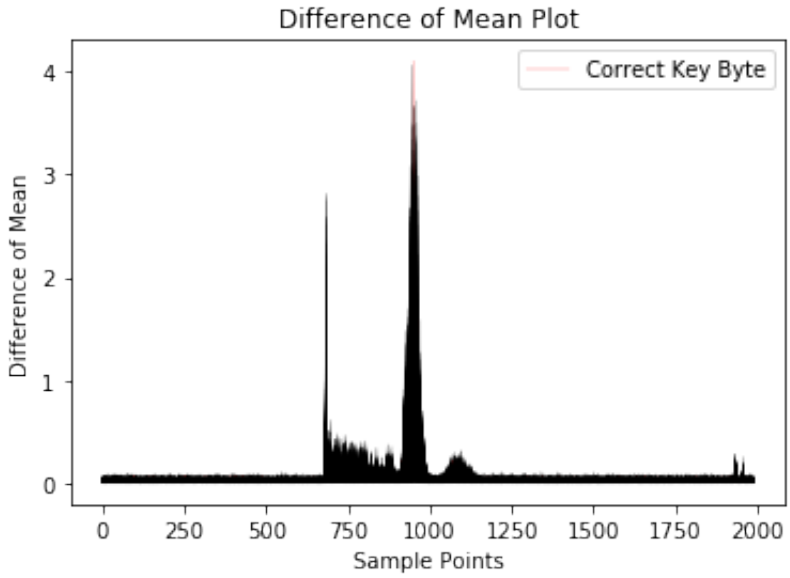


In [13]:
```
maxval=0
correct_key = float("nan")
for i in range(256):
    row=dom_arr[i]
    if(maxval<max(row)):
        maxval=max(row)
        correct_key=i
        correct_row=row

print ("correct_key_byte = " + hex(correct_key))
```

```
correct_key_byte = 0xbe
```

In [14]:
```
fig, ax1 = plt.subplots()
for i in range(256):
    row=dom_arr[i]
    tp=range(len(row))
    if (i == correct_key):
        plt.plot(range(len(correct_row)),correct_row , 'r', linewid
th=0.2,label='Correct Key Byte')
    else:
        plt.plot(tp, row, 'k', linewidth=0.2)

ax1.legend()
plt.locator_params(axis='y', nbins=5)
plt.title('Difference of Mean Plot')
plt.xlabel('Sample Points')
plt.ylabel('Difference of Mean')
plt.savefig("DOM_AllKeyByte.png",dpi=1200,bbox_inches='tight')
plt.show()
```

```
/Users/thorodin/miniconda3/envs/cs236781-hw/lib/python3.7/site-pac
kages/ipykernel_launcher.py:15: UserWarning: Creating legend with
loc="best" can be slow with large amounts of data.
  from ipykernel import kernelapp as app
/Users/thorodin/miniconda3/envs/cs236781-hw/lib/python3.7/site-pac
kages/IPython/core/pylabtools.py:132: UserWarning: Creating legend
with loc="best" can be slow with large amounts of data.
  fig.canvas.print_figure(bytes_io, **kw)
```



Finding the Key Using TRACES Sampled from Real Machines CPA

In [1]:
```
import csv,os
import numpy as np
from scipy import stats
import matplotlib.pyplot as plt
import pandas as pd
```

For this part we use real traces which were taken from FPGA which performed encryption using AES The database for all samples at the time of the last round can be found in the file "_DATA1_keyset_9_attack.csv"

In [2]:
```
KEYSET_FILE_PATH  = "./Resources/_DATA1_keyset_9_attack.csv"
CPA_HW_OUT_PATH = "./Resources/CPA_HW_Ex_Sample.dat"
CPA_Plot_PATH = "./Resources/CPA_AllKeyByte_N"
CPA_BOOK_RES_PATH = "./Resources/CPA_BOOK_RES.dat"
COR_TABLE_PATH = "./Resources/correlation_table.dat"
```

In [3]:
```
wstart = 10
wstop = 1999
wlen = wstop-wstart
InvSbox = (
    0x52, 0x09, 0x6A, 0xD5, 0x30, 0x36, 0xA5, 0x38, 0xBF, 0x40, 0xA
3, 0x9E, 0x81, 0xF3, 0xD7, 0xFB,
    0x7C, 0xE3, 0x39, 0x82, 0x9B, 0x2F, 0xFF, 0x87, 0x34, 0x8E, 0x4
3, 0x44, 0xC4, 0xDE, 0xE9, 0xCB,
    0x54, 0x7B, 0x94, 0x32, 0xA6, 0xC2, 0x23, 0x3D, 0xEE, 0x4C, 0x9
5, 0x0B, 0x42, 0xFA, 0xC3, 0x4E,
    0x08, 0x2E, 0xA1, 0x66, 0x28, 0xD9, 0x24, 0xB2, 0x76, 0x5B, 0xA
2, 0x49, 0x6D, 0x8B, 0xD1, 0x25,
    0x72, 0xF8, 0xF6, 0x64, 0x86, 0x68, 0x98, 0x16, 0xD4, 0xA4, 0x5
C, 0xCC, 0x5D, 0x65, 0xB6, 0x92,
    0x6C, 0x70, 0x48, 0x50, 0xFD, 0xED, 0xB9, 0xDA, 0x5E, 0x15, 0x4
6, 0x57, 0xA7, 0x8D, 0x9D, 0x84,
    0x90, 0xD8, 0xAB, 0x00, 0x8C, 0xBC, 0xD3, 0x0A, 0xF7, 0xE4, 0x5
8, 0x05, 0xB8, 0xB3, 0x45, 0x06,
    0xD0, 0x2C, 0x1E, 0x8F, 0xCA, 0x3F, 0x0F, 0x02, 0xC1, 0xAF, 0xB
D, 0x03, 0x01, 0x13, 0x8A, 0x6B,
    0x3A, 0x91, 0x11, 0x41, 0x4F, 0x67, 0xDC, 0xEA, 0x97, 0xF2, 0xC
F, 0xCE, 0xF0, 0xB4, 0xE6, 0x73,
    0x96, 0xAC, 0x74, 0x22, 0xE7, 0xAD, 0x35, 0x85, 0xE2, 0xF9, 0x3
7, 0xE8, 0x1C, 0x75, 0xDF, 0x6E,
    0x47, 0xF1, 0x1A, 0x71, 0x1D, 0x29, 0xC5, 0x89, 0x6F, 0xB7, 0x6
2, 0x0E, 0xAA, 0x18, 0xBE, 0x1B,
    0xFC, 0x56, 0x3E, 0x4B, 0xC6, 0xD2, 0x79, 0x20, 0x9A, 0xDB, 0xC
0, 0xFE, 0x78, 0xCD, 0x5A, 0xF4,
    0x1F, 0xDD, 0xA8, 0x33, 0x88, 0x07, 0xC7, 0x31, 0xB1, 0x12, 0x1
0, 0x59, 0x27, 0x80, 0xEC, 0x5F,
    0x60, 0x51, 0x7F, 0xA9, 0x19, 0xB5, 0x4A, 0x0D, 0x2D, 0xE5, 0x7
A, 0x9F, 0x93, 0xC9, 0x9C, 0xEF,
    0xA0, 0xE0, 0x3B, 0x4D, 0xAE, 0x2A, 0xF5, 0xB0, 0xC8, 0xEB, 0xB
B, 0x3C, 0x83, 0x53, 0x99, 0x61,
    0x17, 0x2B, 0x04, 0x7E, 0xBA, 0x77, 0xD6, 0x26, 0xE1, 0x69, 0x1
4, 0x63, 0x55, 0x21, 0x0C, 0x7D,
)
```

Section 2

Perform a CPA in which you compare the model (HW) and the measurements for each Trace You must perform a CPA based on the right home key The direction of the trace contains a large number of measurements.

Make the correlation at the beginning of each trace, at the end and at another 20 points that you choose at random along the timeline (of course the same points for all traces).

Perform the experiment for 10,50,100,500,1000,2000, different messages, and indicate the degree of correlation

In [4]:
```
def _variance(a):
    N = len(a)
    res = np.sqrt((N * (np.inner(a, a))) - (np.sum(a)**2))
    return res

def _covariance(a, b):
    N = len(a)
    res = (N * (np.inner(a, b))) - (np.sum(a)) * (np.sum(b))
    return res

def _correlation(a,b):
    res = float("nan")
    var_a_b = (_variance(a) * _variance(b))
    if var_a_b != 0:
        res = _covariance(a, b) / (_variance(a) * _variance(b))
    return res

def HammingWeight(value):
    hammingWeight = 0
    while value > 0:
        hammingWeight += value % 2
        value = value // 2
    return hammingWeight
```

In [5]:
```
MASK = 2**8 - 1 # Least significant byte
number_of_samples = 7
samples = np.append(np.append(wstart, np.sort(np.random.randint(wst
art+1, wstop-1, number_of_samples-2))), wstop)
correlation_arr = np.zeros((256,number_of_samples),dtype='float')
number_of_traces_list = [10,100,500,1000,2000,4000,8000]
print(f"number of traces: {number_of_traces_list}")
```

```
number of traces: [10, 100, 500, 1000, 2000, 4000, 8000]
```

In [6]:
```
print("Chosen samples: " + str(samples))
print("")
```

```
Chosen samples: [  10  110  392  627 1160 1876 1999]
```

In [7]:
```
csv_reader_data = []
with open(KEYSET_FILE_PATH) as csv_file:
        temp_csv_reader = csv.reader(csv_file, delimiter=',')
        for row in temp_csv_reader:
            csv_reader_data.append(row[:-1])
N = (len(csv_reader_data[0])-2)
print(f"Total samples:{len(csv_reader_data)} and N={N}")
```

```
Total samples:8939 and N=2000
```

In [8]:
```
display(pd.DataFrame(csv_reader_data, columns=["PT","CT"] + [f"T{i}
" for i in range(1,N+1)]))
```

	PT	CT	T1
0	21A7D2BD873F22ECB0F4840E2BD69177	D9959FDF641778A83D3D45DC3E282F5D	116
1	C0A06A4EE3A233EE582C28FFC83ABA0D	E412BCFE4AA01B0A2D382149AD3F2906	116
2	07D3580C7695EC206ACB7476E361190C	6BB092EF6950C6B0117FE63E7FAA5C6D	114
3	EEF08757D5D3417F73E3655A17222476	A79DE9A553D24744FF1FD28D4DF43651	114
4	397707577D85012BB34AE3182EBD0BF6	D2133DDCE3CC67928A084F20A2102E77	114
...	...	...	...
8934	3B3684904B9F0ECA079AA1E8BEF1E447	999808DE26928EC8FE7A056F8321CE61	115
8935	D1500312A4781FBAF220D605BCD8848D	143563446FFB99FC145E90370C7732E6	115
8936	8C88E806D1A495630494FA0BE0E14E31	15A46DFB62A25F463ECA8A4E8F9291E4	116
8937	ABAA5DBDB09170C216CA9D2BC8080E2F	FA7C3ABF50C13DB1B9E3E6BDE56BAF84	115
8938	15135C3508DD752CBBA00D0F7D14D911	990DCC7D385EAF77112F4AF324AF46EE	116

8939 rows × 2002 columns

In this file for 8000 different traces, samples of power were performed on a real system It is not known from the system data whether the algorithm applied is serial (ie works on one house at a time) or parallel, ie works on all houses simultaneously, but for the sake of simplicity we assume that the execution is serial Also in this part we will assume that we want to decipher the right house of the key. Therefore, we will do the following:

For each possible key value, and for CT, calculate the power estimate that was generated at the end of round-9 using HW:

In [9]:
```
hamming_prediction_table = np.zeros((256,len(csv_reader_data)))
if os.path.exists(CPA_HW_OUT_PATH):
    with open(CPA_HW_OUT_PATH,"r") as fp:
        for KeyGuess,line in enumerate(fp):
            hamming_prediction_table[KeyGuess] = line.split()
    print("HW computation Complete")
else:
    with open(CPA_HW_OUT_PATH,"w") as fp:
        for kb in range(0,256,1):
            csv_reader = csv_reader_data
            for idx, row in enumerate(csv_reader):
                ct = int(row[1],16)
                ct_temp=ct & MASK
                BIT_MASK = 2**0
                r9_temp = InvSbox[ct_temp ^ kb]
                hamming_prediction_table[kb][idx] = HammingWeight(r
9_temp)
            fp.write(' '.join(map(str, hamming_prediction_table[kb]
)) + "\n")
    print("HW computation Complete")
```

```
HW computation Complete
```

In [10]:
```
display(pd.DataFrame(hamming_prediction_table))
```

	0	1	2	3	4	5	6	7	8	9	...	8929	8930	8931	8932	8933	8934
0	4.0	4.0	5.0	3.0	1.0	2.0	4.0	5.0	3.0	3.0	...	6.0	3.0	3.0	2.0	3.0	4.0
1	5.0	3.0	4.0	4.0	4.0	5.0	3.0	2.0	6.0	2.0	...	5.0	3.0	5.0	2.0	6.0	2.0
2	2.0	2.0	2.0	2.0	6.0	2.0	3.0	3.0	5.0	3.0	...	5.0	3.0	7.0	3.0	3.0	0.0
3	5.0	4.0	3.0	2.0	4.0	4.0	5.0	5.0	4.0	4.0	...	4.0	5.0	5.0	4.0	5.0	5.0
4	3.0	4.0	4.0	6.0	5.0	1.0	3.0	5.0	3.0	5.0	...	7.0	7.0	4.0	5.0	4.0	5.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
251	4.0	2.0	4.0	3.0	4.0	6.0	7.0	5.0	4.0	3.0	...	3.0	6.0	2.0	3.0	3.0	5.0
252	5.0	2.0	4.0	2.0	5.0	4.0	4.0	5.0	5.0	6.0	...	5.0	4.0	2.0	5.0	2.0	5.0
253	4.0	4.0	4.0	4.0	6.0	2.0	5.0	4.0	4.0	3.0	...	3.0	3.0	4.0	3.0	3.0	3.0
254	4.0	4.0	2.0	4.0	5.0	1.0	5.0	4.0	5.0	5.0	...	3.0	6.0	5.0	3.0	4.0	5.0
255	3.0	4.0	4.0	6.0	5.0	4.0	6.0	6.0	4.0	3.0	...	4.0	3.0	0.0	1.0	3.0	7.0

256 rows × 8939 columns

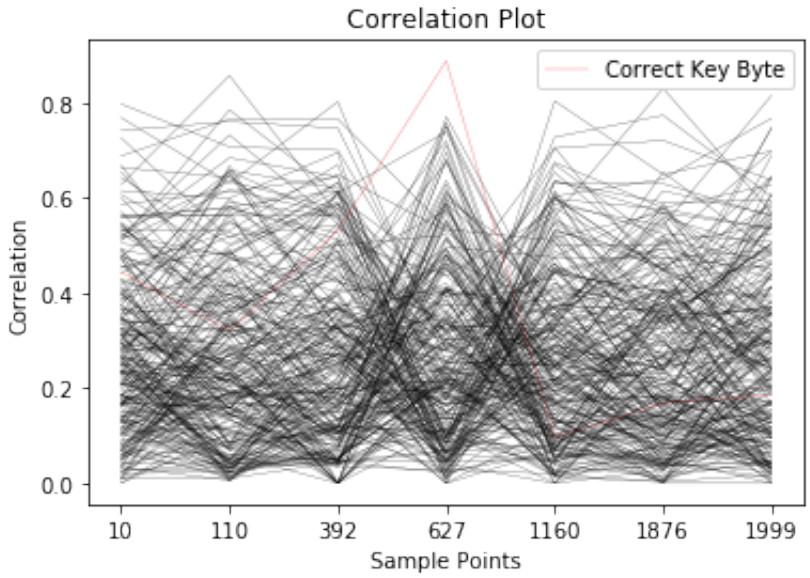
For each column in the table you created (ie an approximate value of a key) find the correlation coefficient of the row with each of the rows of the table from the file

We created a new table in which each row indicates the use of an increasing number of traces and in each box we indicate the correlation value between the approximate key and the measurements

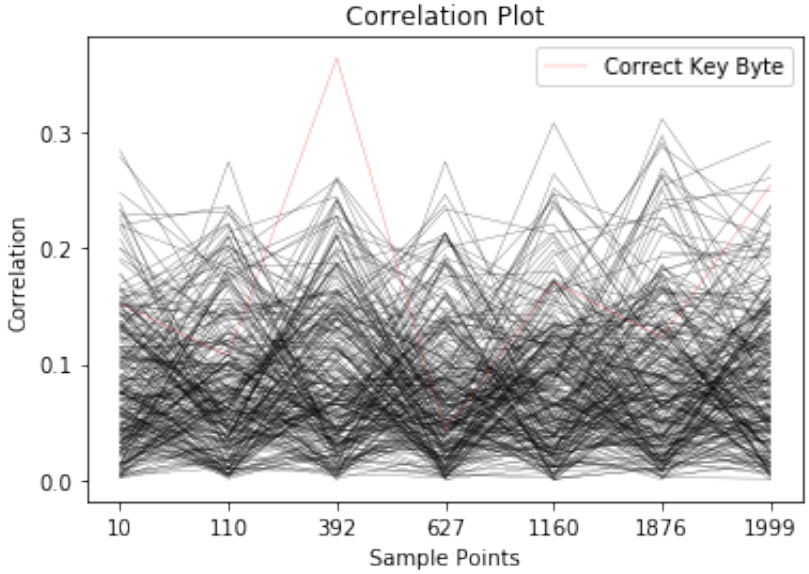
In [12]:
```
def polt_CPA(correlation_arr,number_of_traces):
    fig, ax1 = plt.subplots()
    maxval=0
    correct_key = float("nan")
    for i in range(256):
        row = correlation_arr[i]
        tp = range(len(row))
        if(maxval<max(row)):
            maxval=max(row)
            correct_key=i
            correct_row=row

    print ("correct_key_byte=" + hex(correct_key))

    for i in range(256):
        row=correlation_arr[i]
        tp=range(len(row))
        if (i==correct_key):
            plt.plot(range(len(correct_row)),correct_row , 'r', lin
ewidth=0.2,label='Correct Key Byte')
        else:
            plt.plot(tp, row, 'k', linewidth=0.2)

    plt.xticks(tp, samples)
    ax1.legend()
    plt.locator_params(axis='y', nbins=5)
    plt.title('Correlation Plot')
    plt.xlabel('Sample Points')
    plt.ylabel('Correlation')
    plt.savefig(CPA_Plot_PATH + str(number_of_traces) + ".png",dpi=
1200,bbox_inches='tight')
    plt.show()
```

In [13]:
```
correlation_table = np.zeros((256,len(number_of_traces_list)))
if os.path.exists(COR_TABLE_PATH):
    with open(COR_TABLE_PATH,"r") as fp:
        for i,line in enumerate(fp):
            correlation_table[i] = line.split()
else:
    with open(COR_TABLE_PATH,"w") as fp:
        for i,number_of_traces in enumerate(number_of_traces_list):
            print("Number of traces: " + str(number_of_traces))
            for kb in range(0,256,1):
                    dofmean=np.zeros(wlen,dtype='float')
                    H = np.zeros(number_of_traces)
                    W = np.zeros([number_of_traces, number_of_sampl
es])
                    correlation = np.zeros(number_of_samples)
                    csv_reader = csv_reader_data
                    for idx, row in enumerate(csv_reader):
                        if idx==number_of_traces: break
                        H[idx] = hamming_prediction_table[kb][idx]
                        W[idx][:] = [int(c) for c in np.array(row)[
samples].tolist()]

                    for jdx in range(number_of_samples):
                        correlation[jdx] = _correlation(W[:,jdx], H
)

                    correlation_arr[kb] = np.abs(correlation)
                    correlation_table[kb][i] = np.max(correlation_a
rr[kb])
            polt_CPA(correlation_arr,number_of_traces)
            fp.write(' '.join(map(str, correlation_table[kb])) + "\
n")
```

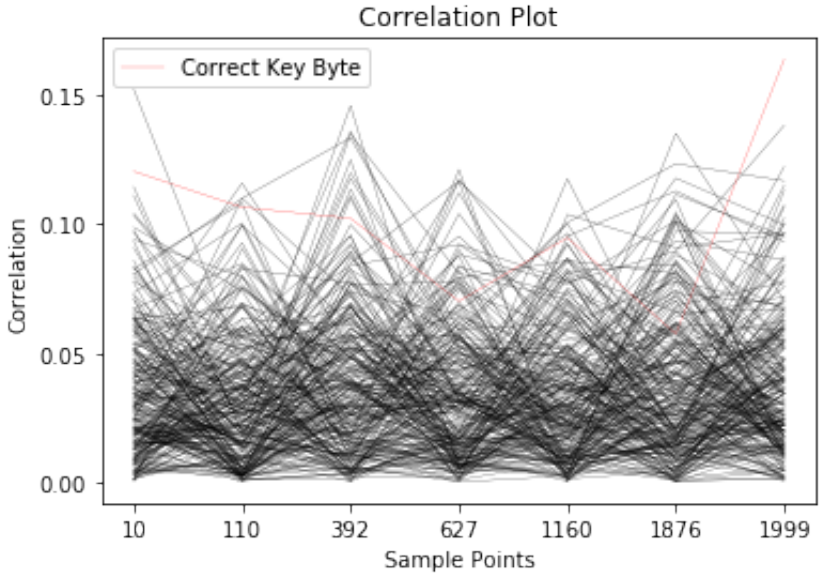
```
Number of traces: 10
correct_key_byte=0xbb
```



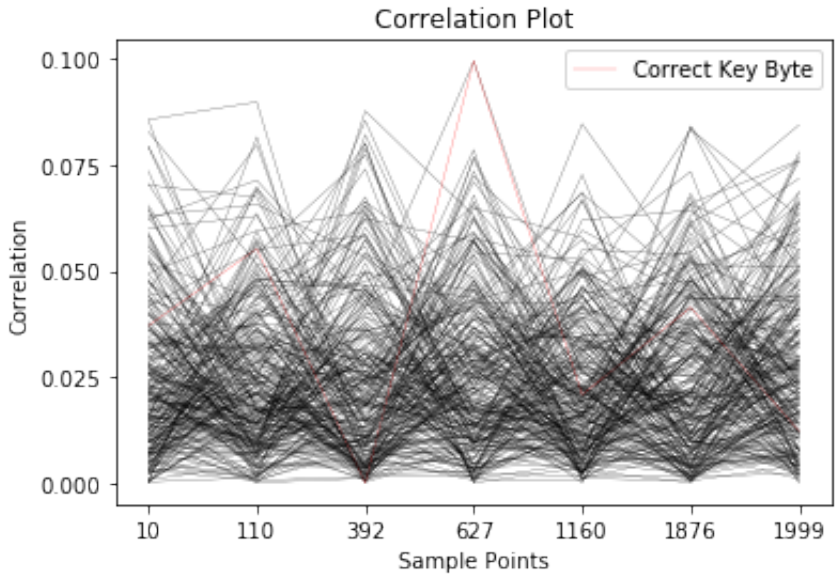
```
Number of traces: 100
correct_key_byte=0xb1
```



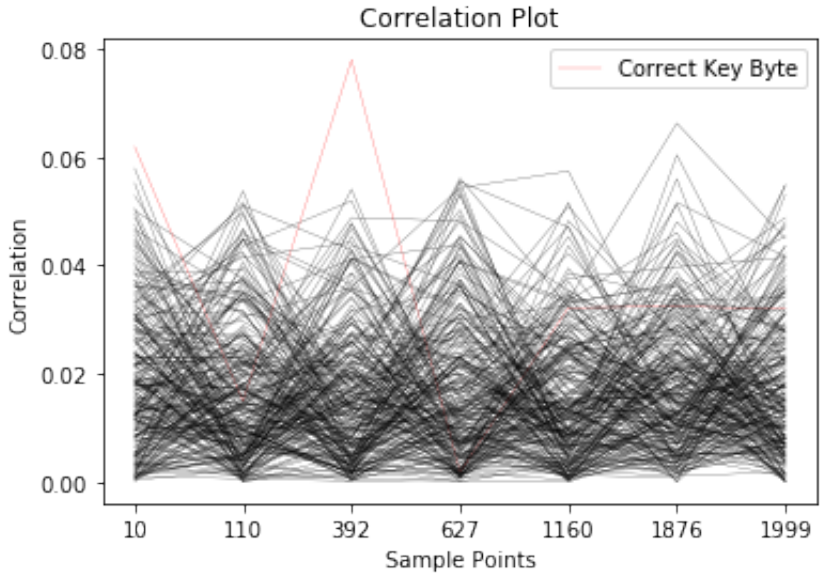

```
Number of traces: 500
correct_key_byte=0xc7
```



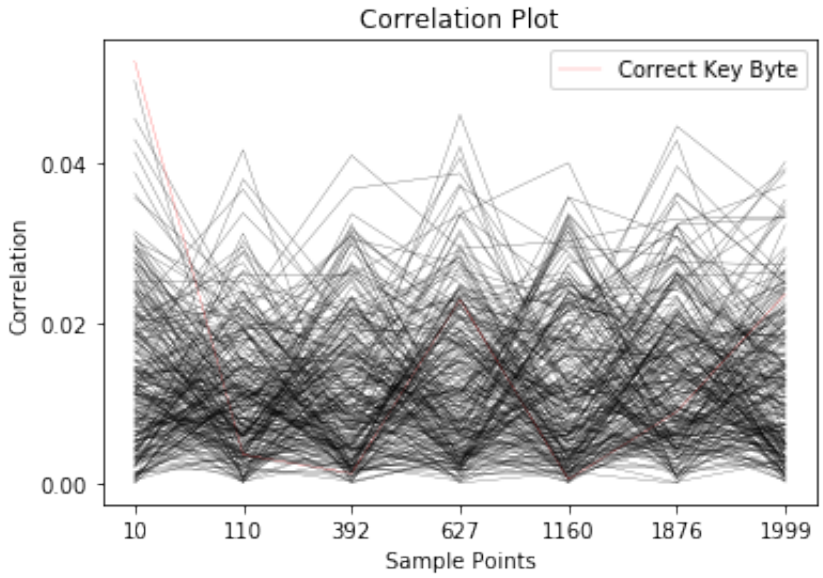

```
Number of traces: 1000
correct_key_byte=0xc3
```



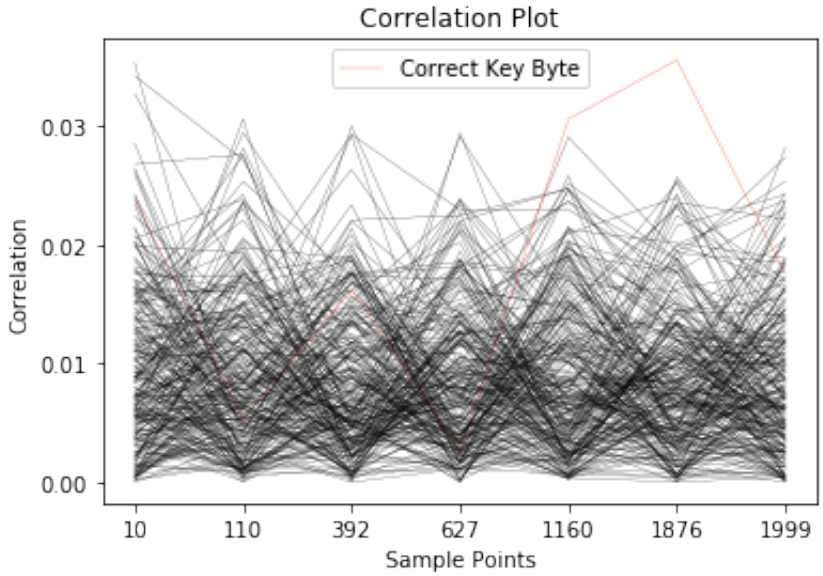

```
Number of traces: 2000
correct_key_byte=0x5a
```




```
Number of traces: 4000
correct_key_byte=0x8f
```




```
Number of traces: 8000
correct_key_byte=0x6b
```



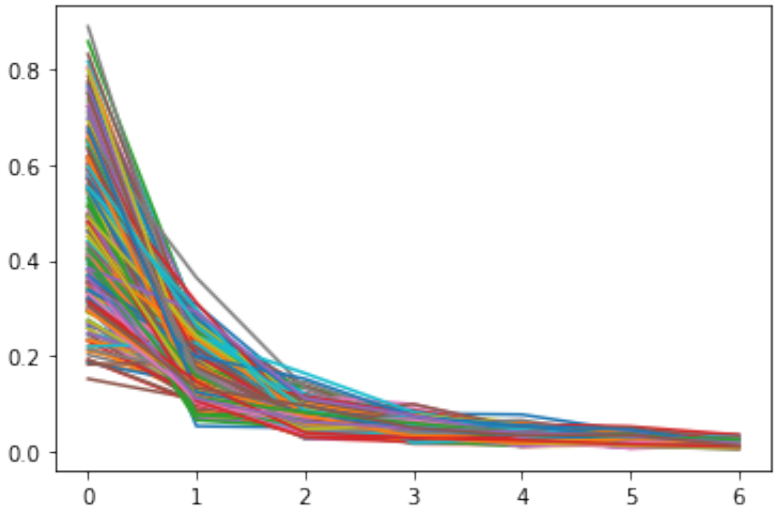
In [18]:
```
display(pd.DataFrame(correlation_table.transpose()))
display(np.argmax(correlation_table,0))
```

	0	1	2	3	4	5	6	7	
0	0.298246	0.803685	0.588439	0.349386	0.349206	0.152484	0.738549	0.202694	0.23598
1	0.096560	0.207705	0.186143	0.251014	0.220645	0.109976	0.150955	0.148772	0.14731
2	0.092187	0.099791	0.053718	0.075286	0.049449	0.064508	0.082694	0.041460	0.04254
3	0.060342	0.057457	0.061920	0.062947	0.051082	0.053631	0.067111	0.042901	0.07930
4	0.043712	0.045702	0.028568	0.037224	0.027645	0.042170	0.051915	0.018207	0.04352
5	0.029854	0.026807	0.018186	0.018971	0.023993	0.023332	0.035453	0.019945	0.01705
6	0.017537	0.020653	0.014786	0.013797	0.014128	0.018723	0.029073	0.020187	0.01375

7 rows × 256 columns

```
array([187, 177, 199, 195,  90, 143, 107])
```

In [19]:
```
plt.plot(correlation_table.transpose())
plt.show()
```



Note

For the sake of comparison, we implemented the algorithm 10.3 according to the book Hardware security - Design, Threats, and Safeguards. The results of the run can be seen here

In [20]:
```
meanTrace = []
result = np.zeros((256,len(csv_reader_data[0])-2))
meanH = np.mean(hamming_prediction_table,1)
for i in range(len(csv_reader_data)):
    lst = [int(c) for c in csv_reader_data[i][2:]]
    meanTrace.append(np.average(lst))
```

In [ ]:
```
print_every = 10
for i in range(0,256,1):
    if i % print_every == 0:
        print(f"Proccesing {hex(i)}")
    for j in range(len(csv_reader_data[0])-2):
        a = 0
        b = 0
        c = 0
        for k in range(len(csv_reader_data)):
            temph = (hamming_prediction_table[i][k] - meanH[i])
            tempt = (int(csv_reader_data[k][j+2]) - meanTrace[j])
            a += temph*tempt
            b += temph*temph
            c += tempt*tempt
        result[i][j] = (a / np.sqrt(b*c))
```

In [ ]:
```
display(pd.DataFrame(result))
```

In [ ]:
```
plt.plot(result)
plt.show()
```