Side Channel Attack Report

Differential Power Analysis & Correlation Power Analysis applied on AES Attack

Realised by : Yanis Alim
Anastasia Cotorobai

Reported to : Karine Heydemann
Quentin Meunier

Master 2 MSI-P9

19-01-2021

DPA Article Summary

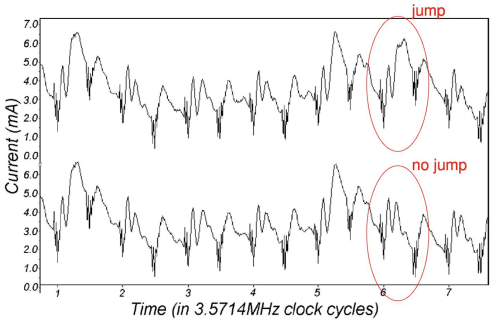
This is a summary of Paul Kocher, Joshua Jae, and Benjamin Jun's article about the Simple Power Analysis and Differential Power Analysis focused on DES (**Data Encryption Standard**) algorithm.

Simple Power Analysis (SPA)

The SPA is focused on the visual analysis of a trace, which can leak information about the algorithm, the operations being used. This information is visually detectable because of the power fluctuations during cryptographic operations.

Simple Power Analysis (SPA) on DES

As we said before, SPA can reveal sequences of instructions executed. For example, we can see the difference from a trace performing a “jump” and a trace with no “jump” instruction in the DES algorithm trace.



Given that the SPA reveals information about the instructions which are executed, it can be used to break cryptographic implementations in which the execution path depends on the data being processed.

For example: DES key schedule, DES permutations, Comparisons, Multipliers, Exponentiation. The key schedule, permutations and comparisons will cause large SPA due to the conditional branching. As for the multipliers and exponentiation, the first one leaks information about the data they process, and the second one may cause the exponent to be compromised if squaring and multiplication operations have different power consumption characteristics,
take different amounts of time, or are separated by different code.

Preventing SPA

- Avoid procedures that use secret intermediates or keys for conditional branching operations
- Hard-wired implementations of symmetric cryptography algorithms

Differential Power Analysis

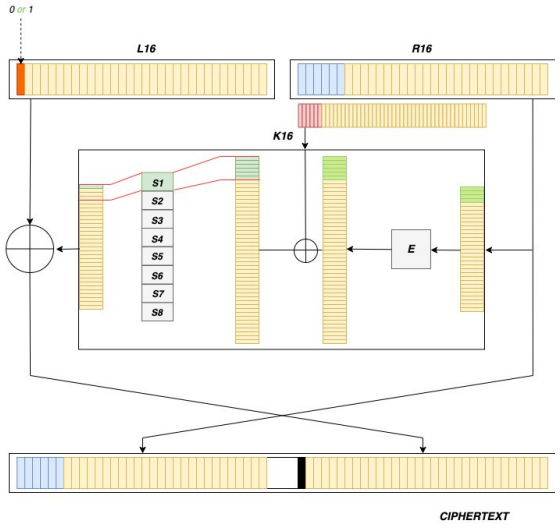
DPA is more powerful than SPA and is much more difficult to prevent. It is based on the effects correlated to data values being manipulated. In the case of DPA we make use of statistical analysis and error correction techniques to extract information correlated to secret keys.

DPA on DES

We know that in each 16 rounds, DES performs 8 S-box lookup operations. An S-box takes 6-bits and produces an output of 4-bits. The whole S-boxes input is the XOR-ed with a 48-bits subkey and the R register which is expanded from 32-bits to 48-bits, just before the xor. (see picture below)

For this attack, we’ll focus on the L register before the last round.
We will create a "selection function" that will compute a specific bit from the left half of the internal state right before the 16th round. The value of this bit will help us to separate the recorded traces into two sets, one which will contain all traces having this bit to "1" and another one, having this bit to "0". This separation is needed because the xor between (L16, f(K16, R16)) will produce a higher power consumption when the target bit from the L16 has a value of "1" then when it has a value of "0".

The selection function, computing the first bit of L16 is represented in the following schema:



To compute the first bit of L16, we need the first bit of the ciphertext, which we already have, and the first bit of the S-box output, which we don't know. To find the output of the S-box, we will make an assumption on the value of the first 6 bits of the subkey, which is xor-ed with the value of R16 register (after expansion). The subkey is the only unknown in our equation because we already know the value of the R register before the expansion function. The R16 register is the left half of our ciphertext.
So, to get the correct bit, we need the ciphertext and the right assumption on the subkey. If we made a wrong assumption on the subkey, the computed bit will be correct half the time.

Before applying the selection function, we already recorded a bunch of traces along with their ciphertext. We will now apply the selection function and separate these traces on two categories , the one with the computed bit supposed to be "1", and another one were the bit is "0". For both subsets, we calculate the average of all traces, then we calculate the difference of these two average traces, which will be the differential trace. If we've made a wrong assumption on the subkey the two subsets will contain similar traces, so they'll be wrongly sorted. The average traces of the two wrong subsets will also be similar, and will cancel each other when we compute the differential trace. So, we will get a flatten trace, which means that the assumption on the subkey is wrong.

On the other hand, if our assumption on the subkey is right, the average traces will be different, because of the xor operation, which will consume less power for the "0" bit subset, and more power for the "1" subset. So the differential trace will cancel all, except this xor operation which will get us a spike in the trace. This means that we've got a correlation, so the assumption on the subkey was right.

We will then repeat this operation for all the bits of the subkeys and get the last round subkey entirely. Having the entire last round subkey will help us get the master key, as the key schedule in DES is a linear operation, so having a subkey is sufficient to get the master key.

Differential Power Analysis of Other Algorithms

DPA can be used to break any symmetric or asymmetric algorithm. We can succeed in reversing different algorithms and protocols using DPA.
Generally, asymmetric operations tend to produce stronger signals leaking than symmetric ones, because of the relatively high computational complexity of multiplication operations.

Preventing DPA

- Increase the number of samples required for an attack by introducing noise into power consumption measurements and reducing signal sizes,
- Design cryptosystems with realistics assumptions, which take into account the hardware component.

Conclusion

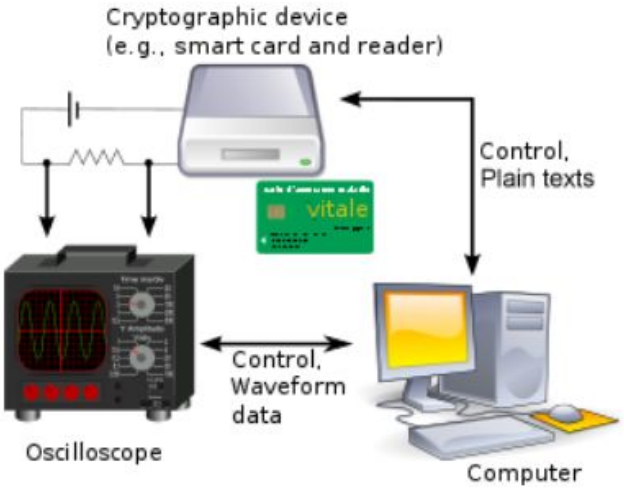
As we have seen, power analysis techniques are of great concern: multiple vulnerable devices, easy to implement, low cost, and difficult to detect. To avoid data leakage by using the power analysis, our systems must be designed with realistic assumptions taking into account all the components (algorithms, protocols, hardware, and software) and their interactions.

Table of contents

Introduction

Power analysis is a branch of side channel attacks where the side channel used is the power consumption. In electronic devices, the instantaneous power consumption is dependent on the data that is being processed in the device as well as the operation performed by that device.

Therefore by analysing the power consumed by a device when it is doing encryption or decryption the key can be deduced.



Consider Figure above. The computer inputs a set of known plain texts to the cryptographic device which does the encryption. While the device performs the encryption the oscilloscope measures the power consumption.

For several hundreds of plaintext samples power traces are obtained and then they are analysed on a computer using an algorithm such as Simple Power Analysis (SPA), Differential Power Analysis (DPA) or Correlation Power Analysis (CPA) to derive the secret key of the system. As the plaintext in this case is known to the attacker this is a known plaintext attack.

This report will focus on using differential power analysis (DPA) to recover the key used in the AES encryption.

When implementing a DPA attack, we need to choose two models: a power model and a statistical analysis model.

In the next sections, we draw theoretical and practical models, namely the Hamming weight model with the correlation coefficient (CPA) and we put them in practice combined with the DPA attack in order to recover an AES key from a non secure target.

Preliminaries

We first need some background knowledge on the Advanced Encryption Standard and differential power analysis attacks.

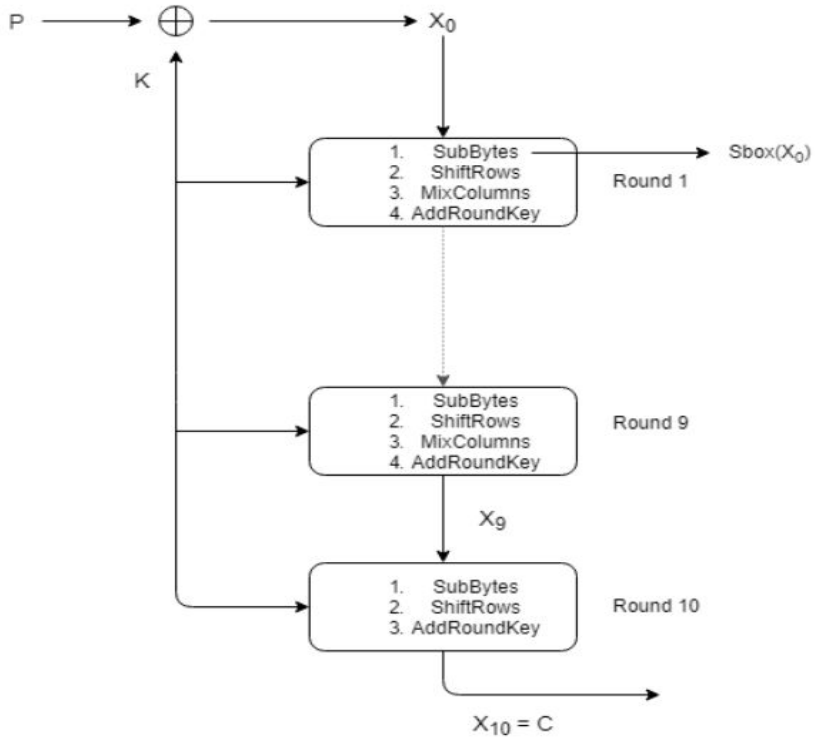
The Advanced Encryption Standard (AES)

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 case we will be attacking a target that uses 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, AddRoundKey. In the 10th round the MixColumns operation is not performed as can be seen in figure below.

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.



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⊕key). The output of the tenth round is the ciphertext C.

We will be using the same LAB architecture and requirements from the last report (Chipswhisperer board, Virtual Machine, Simple Serial Communication Protocol and XMEGA target)

Firmware build

First we need to specify the setup parameters :

- SCOPETYPE = OPENADC # ChipWhisperer Lite's capture hardware
- PLATFORM = CWLITEXMEGA # ChipWhisperer's builtin target
- CRYPTO_TARGET = TINYAES128C # Cryptographic library used by our target

Firmware code

```
int main(void)
{
    uint8_t tmp[KEY_LENGTH] = {DEFAULT_KEY};

    platform_init();
    init_uart();
    trigger_setup();

    aes_indep_init();
    aes_indep_key(tmp);

    simpleserial_init();
    simpleserial_addcmd('k', 16, get_key);
    simpleserial_addcmd('p', 16,  get_pt);
    simpleserial_addcmd('x',  0,   reset);
    simpleserial_addcmd('m', 18, get_mask);
    while(1)
        simpleserial_get();
}
```

This is the main firmware function. In The First Step ,the platform (CWLITEXMEGA) is initialized. Afterwards, the Universal Asynchronous Receiver Transmitter (UART) is initialized. The XMGA target can thus send ascii characters the ChipWhisperer (see the schema above).

Then, the trigger pin is initialized. The trigger is used to signal the Chipwhisperer that a capture must start. Subsequently, the message "hello" is sent to the ChipWhisperer via the UART.

The functions simpleserial_XXXX are for setting up the Simple Serial module :

- simpleserial_init() : add a v command for the version.
- simpleserial_addcmd(char,len,fp) : adds a 'char' command length 'len' and refers its functions 'fp'
- reset() : resets the reception buffer.
- simpleserial_get() : reads input, finds which command is it, reads the parameter for the command, calls the command.
- get_key(): receives a key of 16 bytes.
- get_pt(): receives a plaintext of 16 bytes.
- get_mask(): sends an input of 18 bytes to aes_indep_mask.

The functions aes_indep_XXXX are for setting up the AES library :

- aes_indep_mask(m) : sets the AES mask to m.

```
void aes_indep_mask(uint8_t * m)
{
  int i;
  for (i = 0; i < AESMaskSize; i++)
    mask[i] = m[i];
}
```

- aes_indep_init(): initialize the AES context

```
typedef struct
{
    int nr;                     /*!<  number of rounds  */
    uint32_t *rk;               /*!<  AES round keys    */
    uint32_t buf[68];           /*!<  unaligned data    */
}
mbedtls_aes_context;

void mbedtls_aes_init( mbedtls_aes_context *ctx )
{
    memset( ctx, 0, sizeof( mbedtls_aes_context ) );
}

void aes_indep_init(void)
{
    mbedtls_aes_init(&ctx);
}
```

- aes_indep_key(key) : sets the AES "key" schedule for encryption

```
void aes_indep_key(uint8_t * key)
{
    mbedtls_aes_setkey_enc(&ctx, key, 128);
}

int mbedtls_aes_setkey_enc( mbedtls_aes_context *ctx, const unsigned char *key,
                    unsigned int keybits );
```

Firmware flash

After having the code ready, we compile it as shown in the previous project :

```
SCOPETYPE = 'OPENADC'
PLATFORM = 'CWLITEXMEGA'
CRYPTO_TARGET = 'TINYAES128C'
```

```
%run "Helper_Scripts/Setup_Generic.ipynb"
```

```
fw_path = "../hardware/victims/firmware/simpleserial-aes/simpleserial-aes-{}.hex".format(PLATFORM)
```

```
%%bash -s "$PLATFORM" "$CRYPTO_TARGET"
cd ../hardware/victims/firmware/simpleserial-aes
make PLATFORM=$1 CRYPTO_TARGET=$2
```

We will get a HEX file as a result which represents the firmware to the flash into our target.

Using the Chipwhisperer Python API, we can flash our target after defining the parameters :

```
cw.program_target(scope, prog, fw_path)
```

```
XMEGA Programming flash...
XMEGA Reading flash...
Verified flash OK, 3985 bytes
```

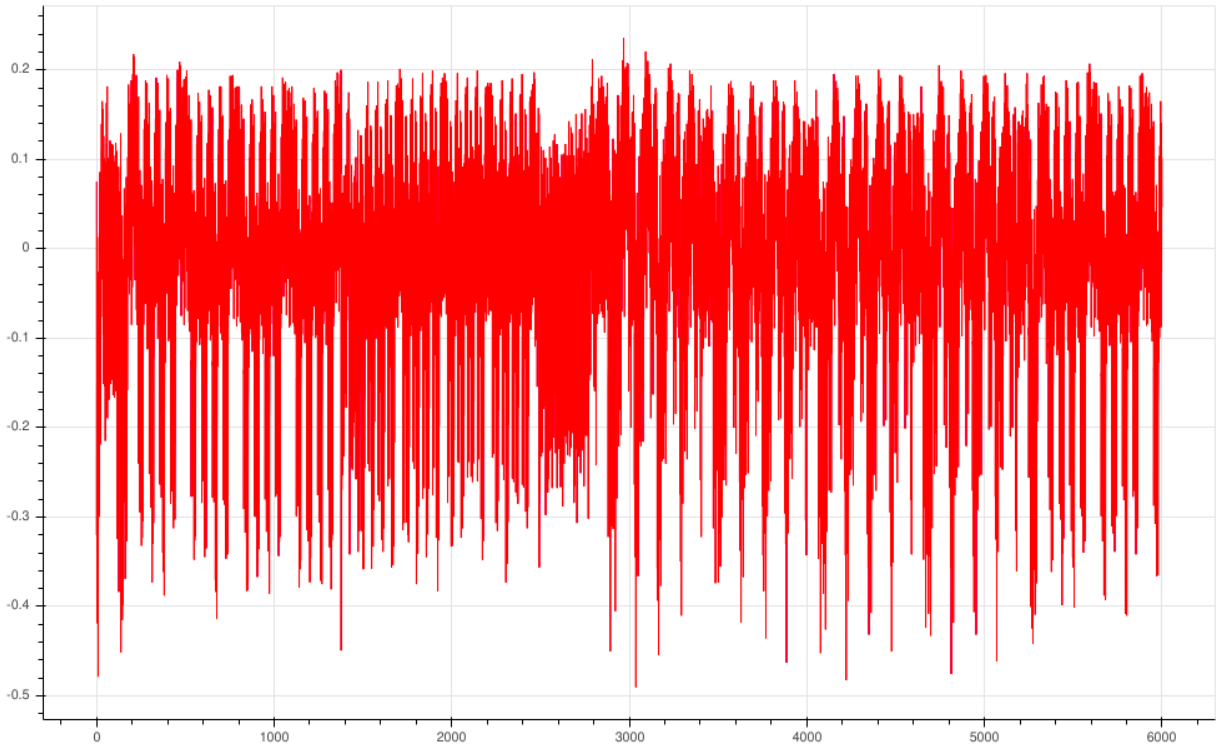
Locating AES Phases and SubBytes operations

Now that we have a general understanding of the AES algorithm, we need to focus on the part we are interested in (SubBytes operations). We will need to approximately locate all the AES Phases.

Locating AES Phases

NOTE: In the next section, we will be saying 6 NOPs theoretically but practically it corresponds to a loop of 6 iterations.

Let's take a general look at the AES trace :

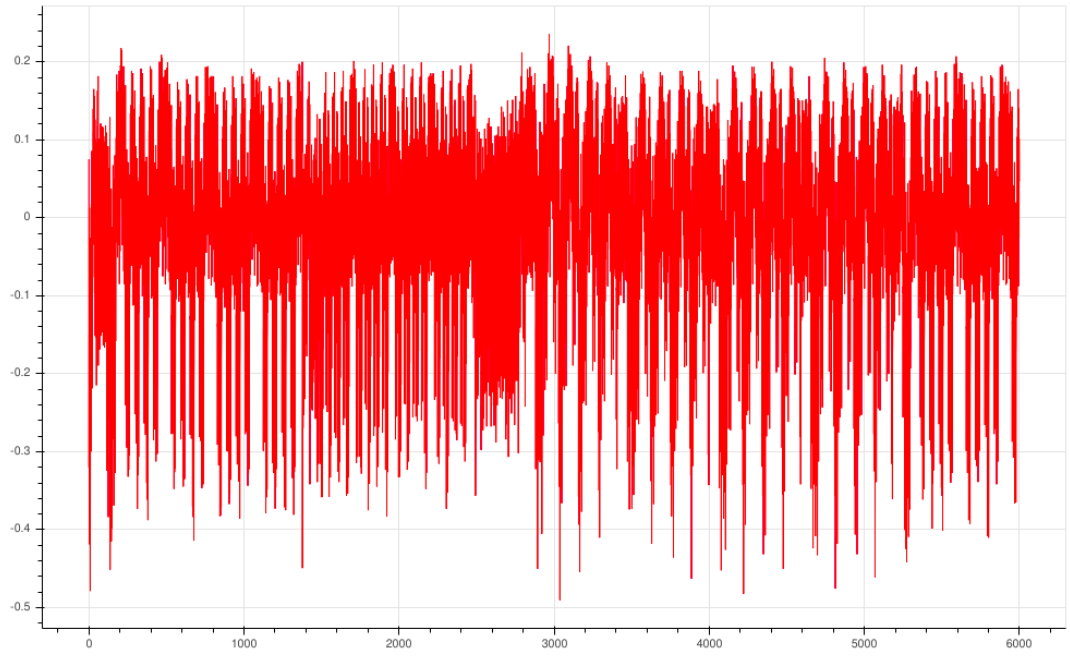


The trace above is so compact and it's a bit hard to differentiate an AES operation from another.

Since we have access to the code, we can add some NOPs between each operation of the Round, by doing that we will be able to see a delay and in result, we will differentiate each operation from an other :

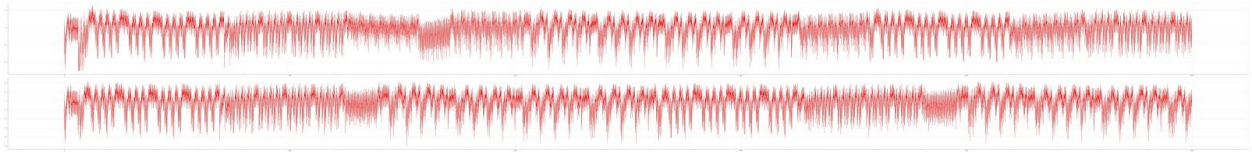
```
static void Cipher(void)
{
  uint8_t round = 0;

  // Add the First round key to the state before starting the rounds.
  AddRoundKey(0);

  // There will be Nr rounds.
  // The first Nr-1 rounds are identical.
  // These Nr-1 rounds are executed in the loop below.
  for(round = 1; round < Nr; ++round)
  {
    SubBytes();
    for(volatile int i=0; i<6; i++);

    ShiftRows();
    for(volatile int i=0; i<6; i++);

    MixColumns();
    for(volatile int i=0; i<6; i++);

    AddRoundKey(round);
    for(volatile int i=0; i<6; i++);
  }

  // The last round is given below.
  // The MixColumns function is not here in the last round.
  SubBytes();
  for(volatile int i=0; i<6; i++);

  ShiftRows();
  for(volatile int i=0; i<6; i++);

  AddRoundKey(Nr);
  for(volatile int i=0; i<6; i++);

}
```

And the trace looks like this :



It's hard to tell where our NOPs are. In consequence it's also hard to distinguish the round operations.

Then, we tried increasing the number of samples and put the traces one above another so that we can see the changes from a normal trace to a trace with NOPs between rounds :



Now that we have a way to compare between operations and spot them, let's do it in 1 shot !

The plan :

- Generate a ***fixed plaintext and key*** so that we make sure the operations consumes the same amount of time and power
- Store both traces (one with NOPs and other without) : add a boolean flag
- Add ticks on the x axis (every 50 samples) so that we don't do much calculations.

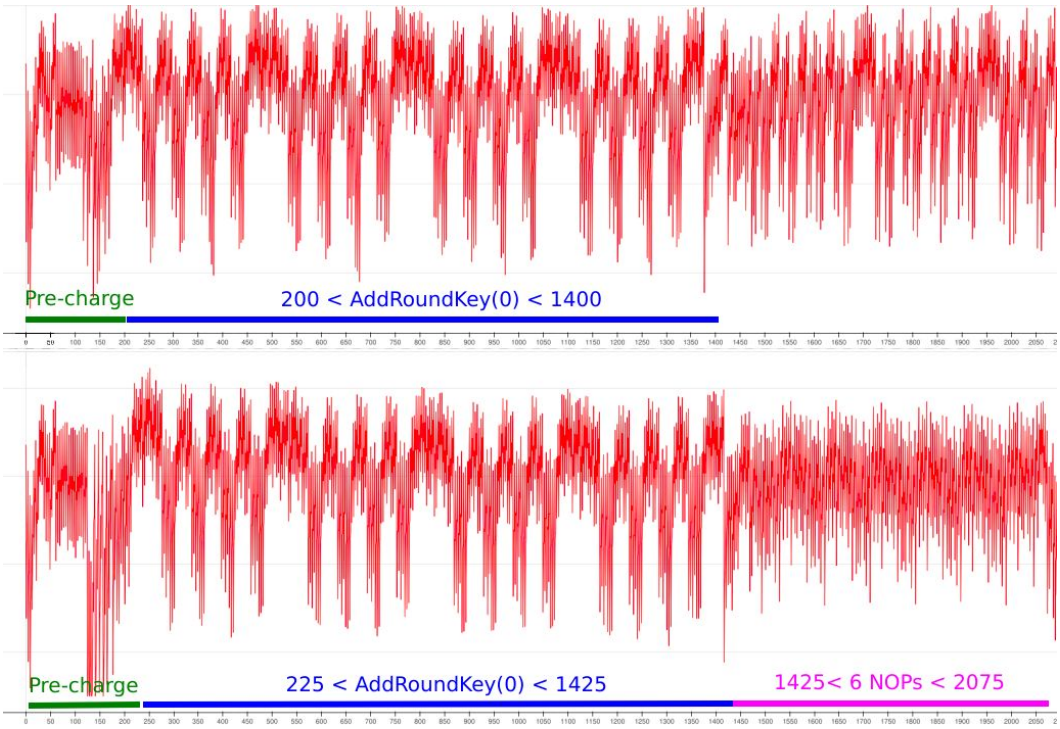
```
key  = bytearray.fromhex('2b7e151628aed2a6abf7158809cf4f3c')
text = bytearray.fromhex('6f27aeb1e96b7f2793639336a546d40a')

NOPS = True
if NOPS :
    trace_nops  = cw.capture_trace(scope, target, text, key)
else :
    trace       = cw.capture_trace(scope, target, text, key)

select_axis_key = list(range(0,21000,50))
p.xaxis.ticker  = FixedTicker(ticks=select_axis_key)

if NOPS :
    p.line(xrange, trace_nops.wave, line_color = "red")
else :
    p.line(xrange, trace.wave,      line_color = "red")

show(p)
```

Also having added 6 NOPs after each operation :

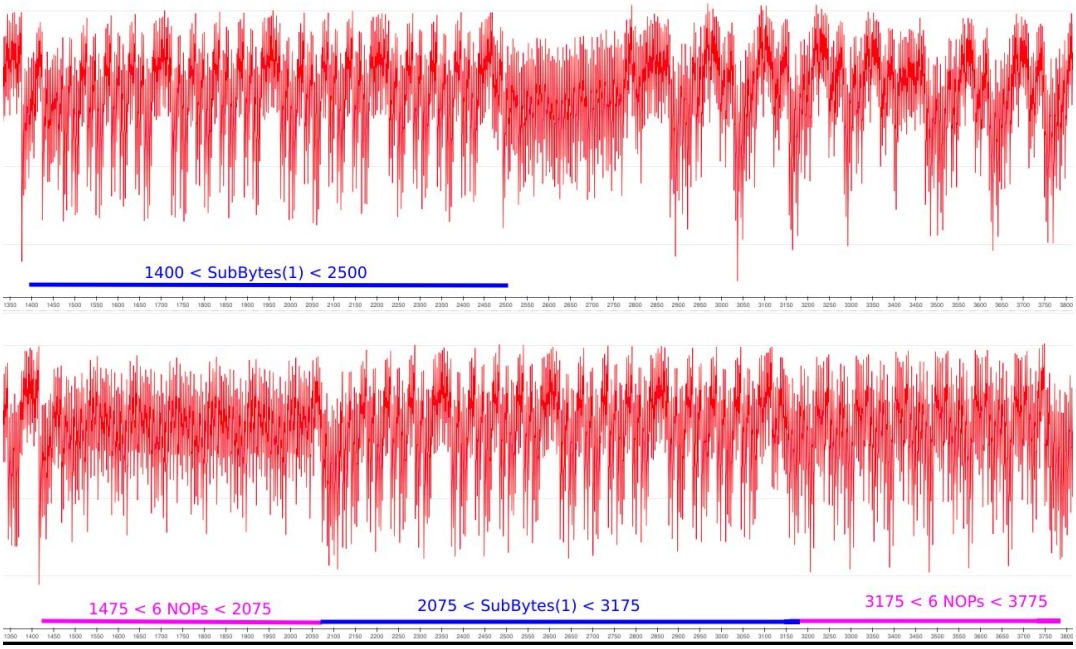
```
static void Cipher(void)
{
  uint8_t round = 0;

  // Add the First round key to the state before starting the rounds.
    AddRoundKey(0);     for(volatile int i=0; i<6; i++);

  // There will be Nr rounds.
  // The first Nr-1 rounds are identical.
  // These Nr-1 rounds are executed in the loop below.
  for(round = 1; round < Nr; ++round)
  {
    SubBytes();         for(volatile int i=0; i<6; i++);
    ShiftRows();        for(volatile int i=0; i<6; i++);
    MixColumns();       for(volatile int i=0; i<6; i++);
    AddRoundKey(round); for(volatile int i=0; i<6; i++);
  }

  // The last round is given below.
  // The MixColumns function is not here in the last round.
    SubBytes();         for(volatile int i=0; i<6; i++);
    ShiftRows();        for(volatile int i=0; i<6; i++);
    AddRoundKey(Nr);    for(volatile int i=0; i<6; i++);
}
```

We can now spot the range of each operation :

- Line 3: ***AddRoundKey(0)*** starts at around 200 samples and takes around 1200 samples to finish ***(200 -> 1400)***

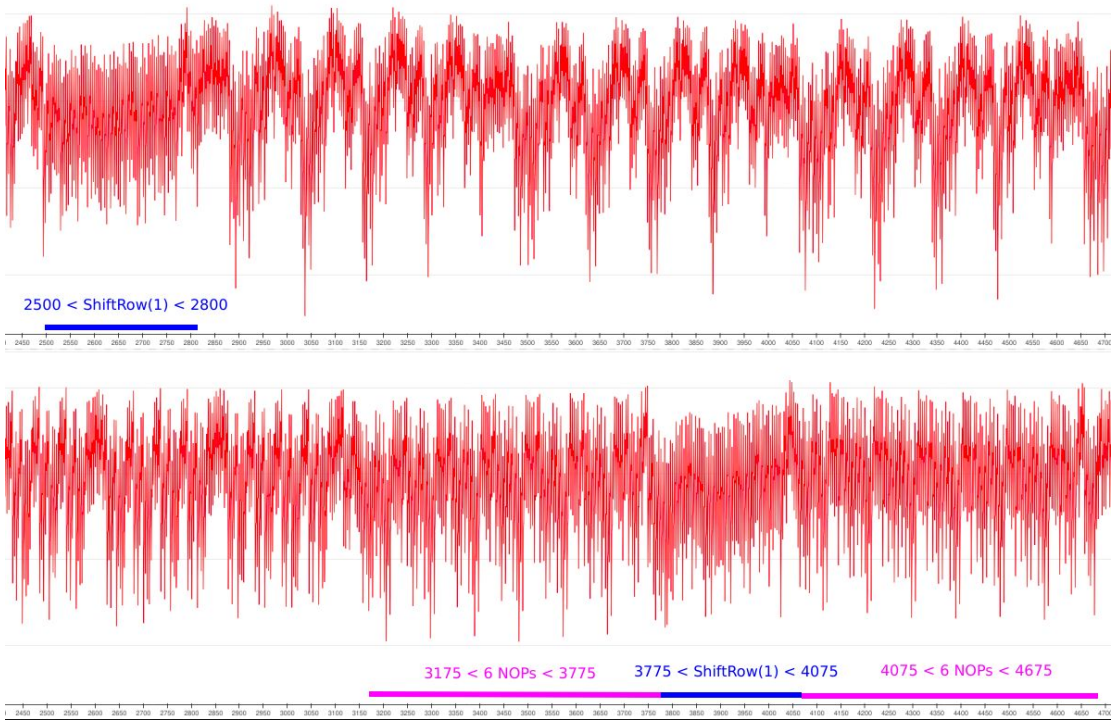


- Line 13: ***SubBytes(1)*** starts at around 1400 samples and takes 1100 samples to finish ***(1400 -> 2500)***



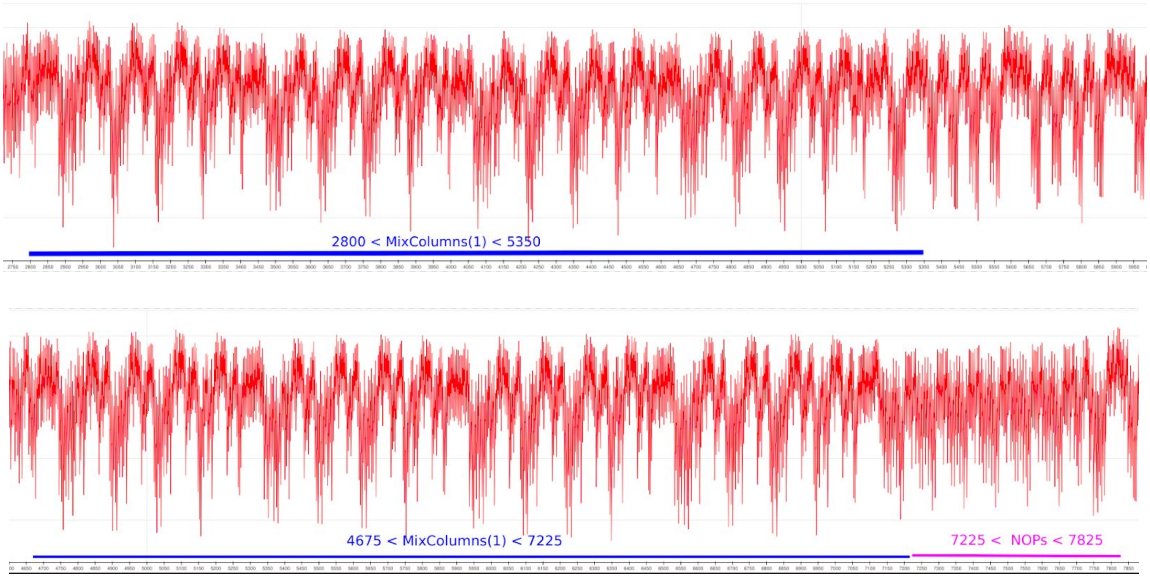
- Line 14: ***ShiftRows(1)*** starts around 2500 samples and takes around 300 samples to finish

(2500 -> 2800)

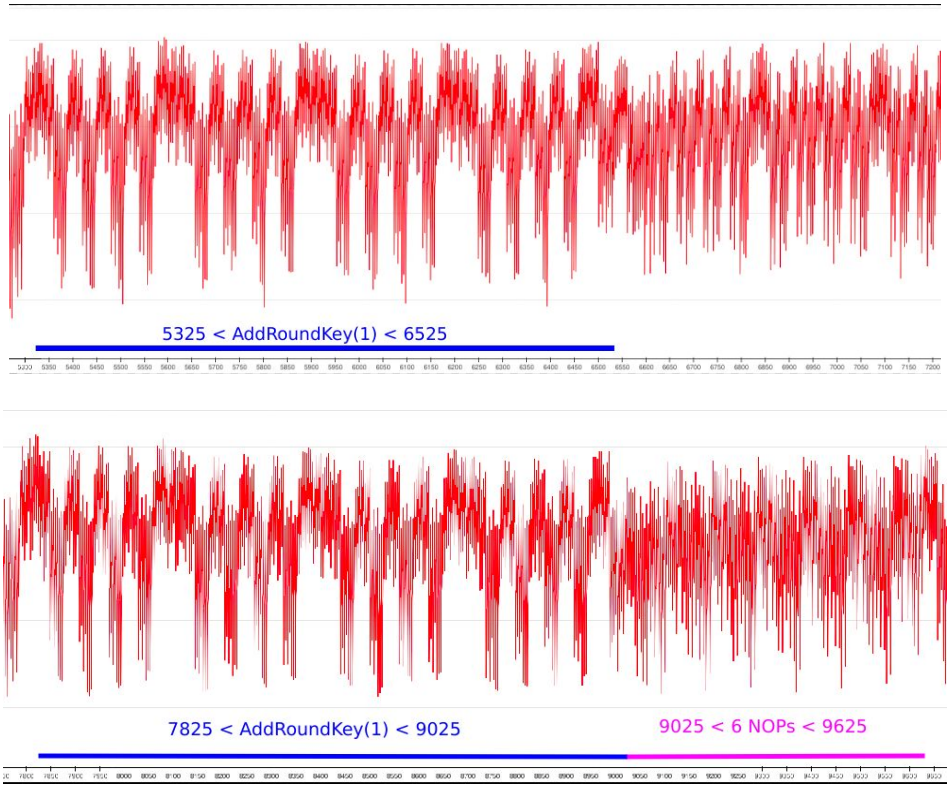


- Line 15: ***MixColumns(1)*** start around 2800 samples and takes around 2550 samples to finish

(2800 -> 5350)



- Line 16: ***AddRoundKey(1)*** start around 5325 samples and takes around 1200 samples to finish ***(5325 - > 6525)***



This confirms our calculations for the 1st AddRounKey(0) where we found that this operation takes around 1200 samples.

AddRoundKey(1) goes from 5325 to 6525 and that's around 1200 samples.

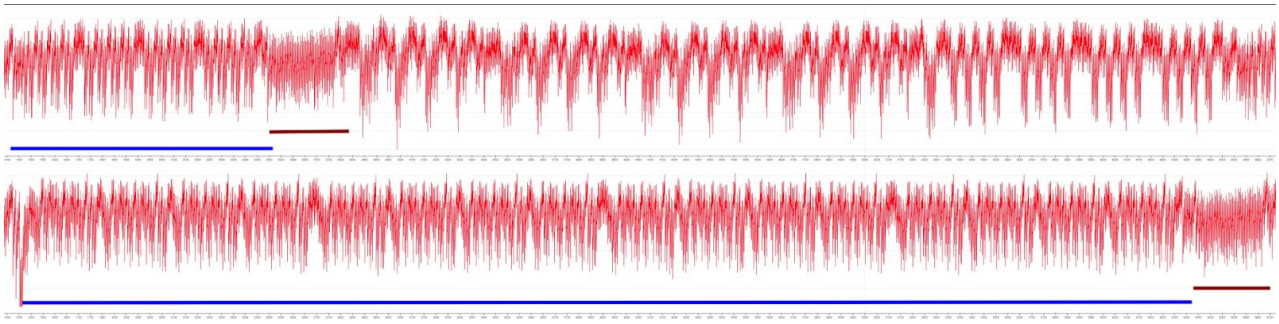
SubBytes operations

Meanwhile our calculation for the AES phases were approximative, we don't wanna do that for the SBox computation since it's our attack point !

We will be adding some NOPs before each SBox operation :

```
// The SubBytes Function Substitutes the values in the
// state matrix with values in an S-box.
static void SubBytes(void)
{
  uint8_t i, j;
  for(i = 0; i < 4; ++i)
  {
    for(j = 0; j < 4; ++j)
    {
      for(volatile int i=0; i<2; i++);
      (*state)[j][i] = getSBoxValue((*state)[j][i]);
    }
  }
}
```

The picture below shows a trace of SubBytes operation with and without NOPs before each SBox operation :



The blue color represents the SubBytes operation and the brown corresponds to the ShiftRows operation.

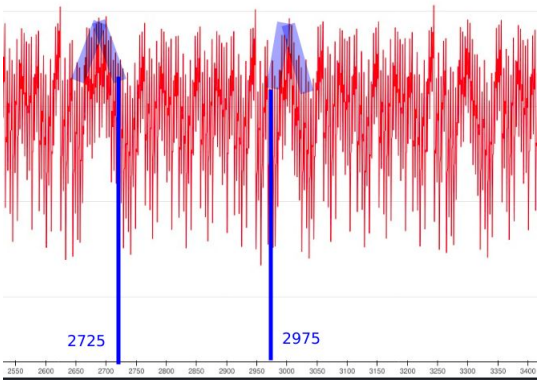
We notice that :

- Without NOPs (1st trace) :

The SubBytes operation takes around 980 samples, going from 1470 to 2450. We know that the total number of SBox operations is 16. We conclude that each operation takes around 60 samples.

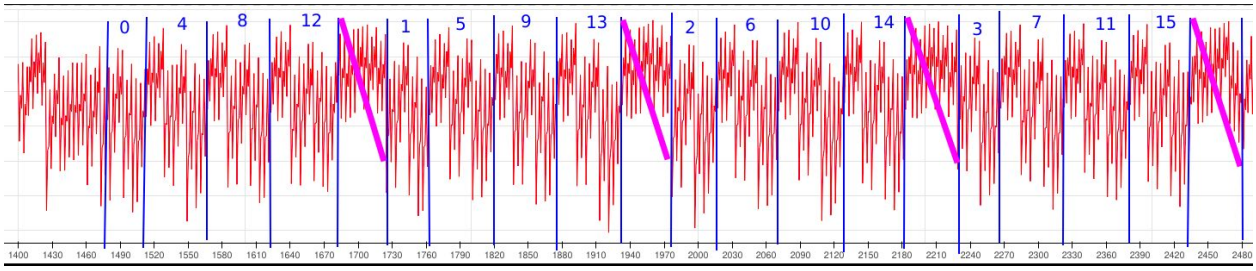
- With 2 NOPs (2nd trace):

The SubBytes operation takes around 4800 samples, going from 1710 to 6510. We try to calculate the number of samples for 2 loop iterations :



Knowing that 2 NOPs takes around 240 samples :
(240 + 60) * 16 = 4800 samples for the SubBytes with 2 NOPs before each SBox operation. This proves the exactitude of our calculation if all the SBox operations take the same amount of samples.

To view the SBox operations in detail, we will move our adc offset to around 1400 samples and capture to the 2600th sample, we will also make more accurate precision in the x axis tick :



We detected the location where the first byte in the S-box is computed at around 1472-th sample, and the end of the last byte at 2452-th sample. The total number of samples required for capturing each of the 16 operations is around 980, with around 56 samples per operation.

All four sets of four bytes manipulated in the Sbox are stored in columns. In consequence, the second set of samples for example would correspond to the first byte of the second row in the output of the S-box and so forth.

We end up with the following ranges:

```
ranges = [
          """0"""                """1"""              """2"""              """3"""
      range(1472 , 1528), range(1724 , 1780), range(1976 , 2032), range(2228 , 2284),
          """4"""                """5"""              """6"""              """7"""
      range(1528 , 1584), range(1780 , 1836), range(2032 , 2088), range(2284 , 2340),
          """ 8"""               """9"""               """10"""             """11"""
      range(1584 , 1640), range(1836 , 1892), range(2088 , 2144), range(2340 , 2396),
          """12"""               """13"""              """14"""             """15"""
      range(1640 , 1696), range(1892 , 1948), range(2144 , 2200), range(2396 , 2452)
]
```

Introduction to DPA & HW Assumption

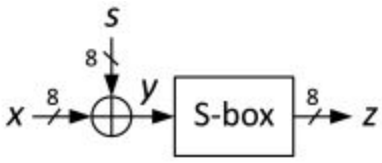
Now that we have identified the SubBytes phase and the SBox operations, we wish to attack them using Differential Power Analysis based on the Hamming Weight Assumption.

DPA Attack theory

The AES (and all block ciphers in general) is designed such that small size blocks (typically 8 bits) can be processed independently, providing efficient implementations on all kinds of platforms. The figure below illustrates a typical 8-bit key addition (i.e., AddRoundKey) and S-box layer (i.e., SubBytes) in a round of the AES.

For instance, in the case of the first round, the input “x” corresponds to a plaintext byte, and the value “s” corresponds to a byte of the first round key (hence a byte of the master key). The output of the AddRoundKey and the SubBytes are respectively denoted as “y” and “z”. Given that the input “x”

is assumed to be known (as well as the AES operations), by capturing power leakages while "y" and "z" are produced, the adversary indirectly recovers information about the key byte "s".



A DPA attack follows a *divide-and-conquer* strategy. In other words, key bytes are separately attacked, then reassembled to recover the 128-bit round key. The attack on an individual key byte "s" is performed with the following steps :

1. The adversary captures a series of power leakage traces (corresponding to the target round) and their respective input "x" . As a reminder, the input "x" is known but not necessarily chosen. The key byte "s" keeps the same value for all of the traces.

2. For now, "s" is unknown, but since it is a byte, it can be easily brute forced (s is a value between 0 and 255). Therefore, for each possible value of "s", the adversary predicts the intermediate values "y" and "z".

3. For each prediction of "z" (or "y"), the adversary makes use of a ***leakage model***. The role of a leakage model is to emulate the way intermediate values are reflected in leakage traces.

4. The modeled predictions (corresponding to a key candidate) are then compared to the observed leakages using a *distinguisher*. The correct key candidate is the one returning the highest similarity.

The efficiency of a DPA attack resides in appropriately choosing the leakage model and the distinguisher.

The Hamming weight is used as a leakage model. It requires no training, and it is ***based on the assumption that power leakages vary linearly with the number of 1's in the binary representation of the target value.*** For instance, for an 8-bit AES S-box output "z", the Hamming weight is expressed as follows:

$$HW(z) = \sum_{i=0}^{7} b_i \quad \text{with } z = (b_0 b_1 \cdots b_7)_2.$$

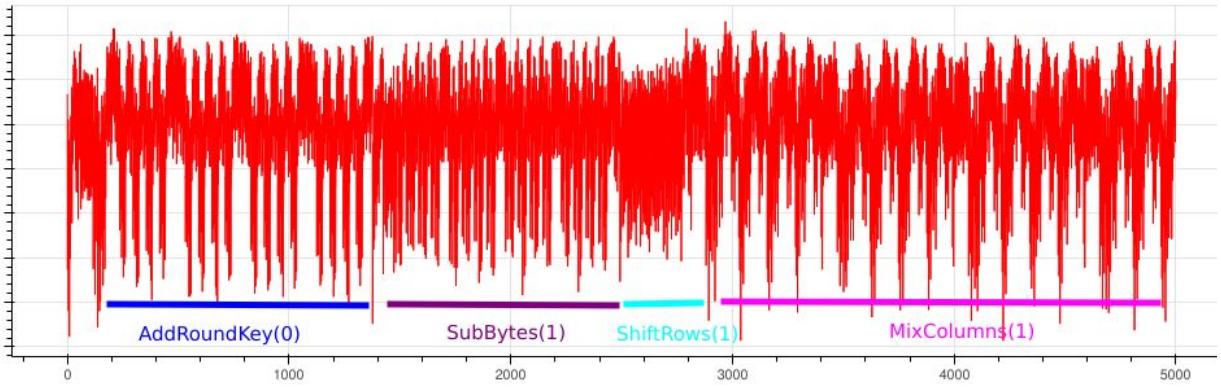
How do we prove that there is a linear relation between the power consumption and the hamming weight ? Let's plot the Hamming weight (HW) of the data to figure this out along with the power traces!

Capturing traces

In order to plot the HW, we first need to have some data. For that, we will be capturing around 1000 traces and store them in an array.

Each trace contains 3 fields : a wave (the data), textin (input plaintext), key (input key).

After capturing the traces, we can plot one for example :



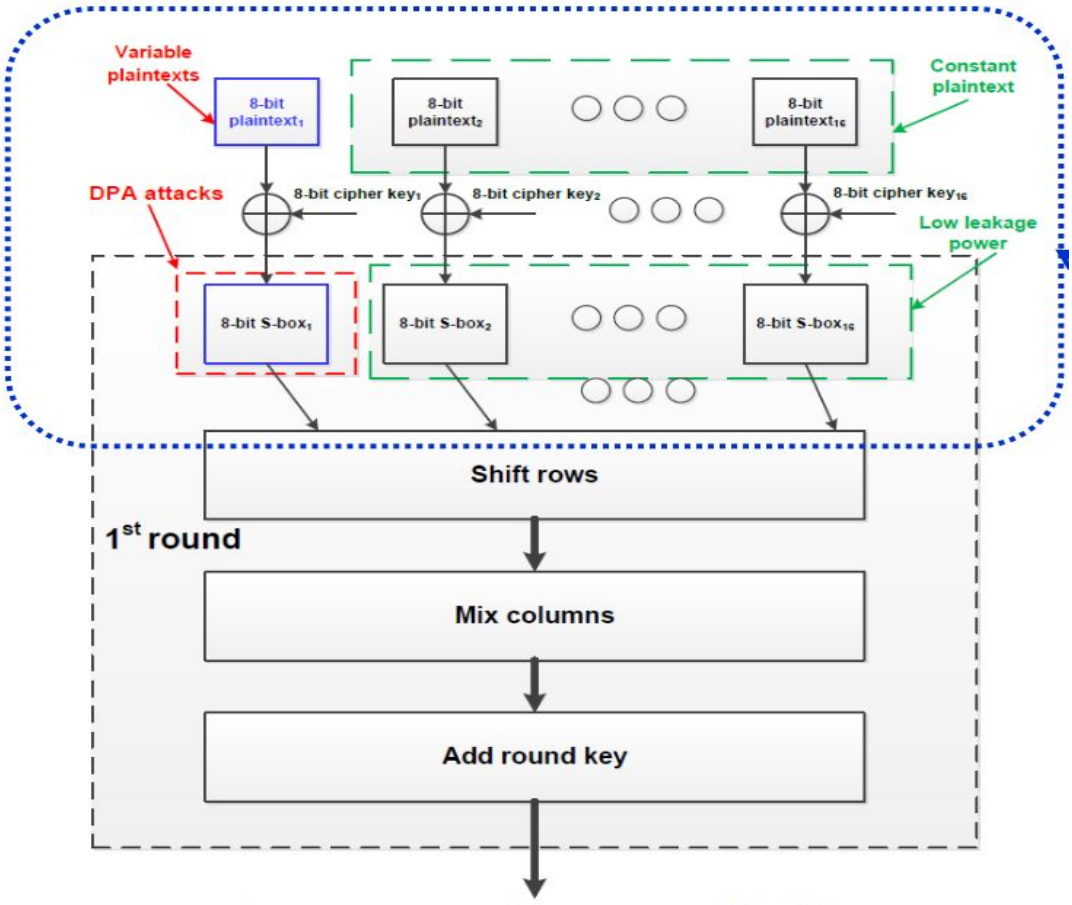
It looks like now we are identifying the AES phases easily!

Using the trace data

Now that we have some data, we can loop through all the bytes of the key for all the captured traces to make a key guess :

```
trace_array = np.asarray([trace.wave for trace in traces]) #storing traces

numtraces = np.shape(trace_array)[0] #total number of traces
numpoints = np.shape(trace_array)[1] #samples per trace

for bnum in range(0, 16):
    for tnum in range(0, numtraces):
        guess(tracearray[tnum], bnum) #make a guess for the byte "bnum" of the key for the trace "tnum"
```

The function guess isn't yet implemented and for that we need to go back to how AES works and understand how we gonna perform the attack in order to have the best guess :



The exact point of attack is the output of the 1st SBox round, why ?
Because the first SBox round contains the master key as it is due to the AddRoundKey(0) which adds a 0 to the master key which means no changes to the target key.
Then it's passed to the SubBytes(1) operation which represents mathematically :

SBox (XOR(Pi , Ki))

This is easily reversible or brute forceable, why ?

The SBox is a well known matrix and available for everyone, XOR is a simple operation.

We can perform a byte guess of the key and then compare the trace at the leakage point and then compare it to the original trace at that same point. The good guess will have a bigger difference than other wrong guesses.

At this point, we have all the necessary information to brute force the key byte by byte. The total tries is ***256 (possible key) * 16 rounds = 4096 tries*** which is easily computable and a lot better than brute forcing the key bit by bit ***(2^128)*** which is almost impossible to compute nowadays.

Calculating the Hamming Weight

We previously said that the output of SBox is obviously visible on trace but this assumption is due to the property of the power consumption is lineare to the Hamming Weight !

Now, we need to calculate the HW of the SBox result, let's define the function that does that :

```
sbox = (
    0x63, 0x7c, 0x77, 0x7b, 0xf2, 0x6b, 0x6f, 0xc5, 0x30, 0x01, 0x67, 0x2b, 0xfe, 0xd7, 0xab, 0x76,
    0xca, 0x82, 0xc9, 0x7d, 0xfa, 0x59, 0x47, 0xf0, 0xad, 0xd4, 0xa2, 0xaf, 0x9c, 0xa4, 0x72, 0xc0,
    0xb7, 0xfd, 0x93, 0x26, 0x36, 0x3f, 0xf7, 0xcc, 0x34, 0xa5, 0xe5, 0xf1, 0x71, 0xd8, 0x31, 0x15,
    0x04, 0xc7, 0x23, 0xc3, 0x18, 0x96, 0x05, 0x9a, 0x07, 0x12, 0x80, 0xe2, 0xeb, 0x27, 0xb2, 0x75,
    0x09, 0x83, 0x2c, 0x1a, 0x1b, 0x6e, 0x5a, 0xa0, 0x52, 0x3b, 0xd6, 0xb3, 0x29, 0xe3, 0x2f, 0x84,
    0x53, 0xd1, 0x00, 0xed, 0x20, 0xfc, 0xb1, 0x5b, 0x6a, 0xcb, 0xbe, 0x39, 0x4a, 0x4c, 0x58, 0xcf,
    0xd0, 0xef, 0xaa, 0xfb, 0x43, 0x4d, 0x33, 0x85, 0x45, 0xf9, 0x02, 0x7f, 0x50, 0x3c, 0x9f, 0xa8,
    0x51, 0xa3, 0x40, 0x8f, 0x92, 0x9d, 0x38, 0xf5, 0xbc, 0xb6, 0xda, 0x21, 0x10, 0xff, 0xf3, 0xd2,
    0xcd, 0x0c, 0x13, 0xec, 0x5f, 0x97, 0x44, 0x17, 0xc4, 0xa7, 0x7e, 0x3d, 0x64, 0x5d, 0x19, 0x73,
    0x60, 0x81, 0x4f, 0xdc, 0x22, 0x2a, 0x90, 0x88, 0x46, 0xee, 0xb8, 0x14, 0xde, 0x5e, 0x0b, 0xdb,
    0xe0, 0x32, 0x3a, 0x0a, 0x49, 0x06, 0x24, 0x5c, 0xc2, 0xd3, 0xac, 0x62, 0x91, 0x95, 0xe4, 0x79,
    0xe7, 0xc8, 0x37, 0x6d, 0x8d, 0xd5, 0x4e, 0xa9, 0x6c, 0x56, 0xf4, 0xea, 0x65, 0x7a, 0xae, 0x08,
    0xba, 0x78, 0x25, 0x2e, 0x1c, 0xa6, 0xb4, 0xc6, 0xe8, 0xdd, 0x74, 0x1f, 0x4b, 0xbd, 0x8b, 0x8a,
    0x70, 0x3e, 0xb5, 0x66, 0x48, 0x03, 0xf6, 0x0e, 0x61, 0x35, 0x57, 0xb9, 0x86, 0xc1, 0x1d, 0x9e,
    0xe1, 0xf8, 0x98, 0x11, 0x69, 0xd9, 0x8e, 0x94, 0x9b, 0x1e, 0x87, 0xe9, 0xce, 0x55, 0x28, 0xdf,
    0x8c, 0xa1, 0x89, 0x0d, 0xbf, 0xe6, 0x42, 0x68, 0x41, 0x99, 0x2d, 0x0f, 0xb0, 0x54, 0xbb, 0x16)

def intermediate(pt, keyguess):
    return sbox[pt ^ keyguess]
```

After that we have the output of the SBox, we will need to calculate its HW, which is basically the numbers of 1s in its binary representation.
We will calculate the HW for all the possible outputs :

```
HW = [bin(n).count("1") for n in range(0, 256)]

#Example - PlainText is 0x12, key is 0xAB
HW[intermediate(0x12, 0xAB)]
#HW(0x12, 0xAB) =  4
```

Plotting the HW

In order to spot the leakage points, we will be tracing the 16 SubBytes operations using the ranges extracted above. Then, since we have the same Key for 1000 different plaintexts, it means that the HW will vary from 0 to 8 :

SBox(XOR(K, Pi)) ; 0 <= Pi < 256; 0 < i < 1000

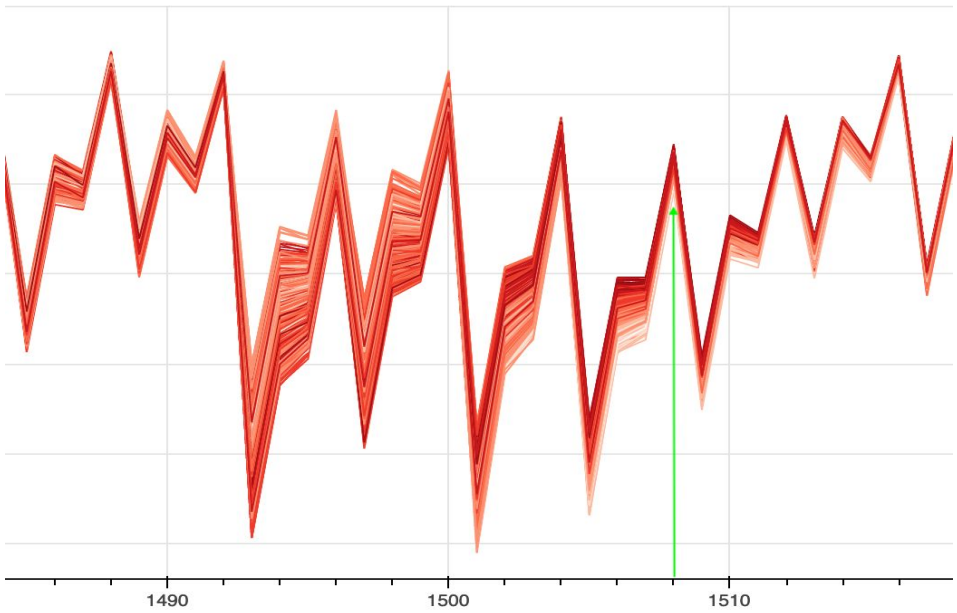
With a high probability, we will have different outputs for the SBox operation and this implies all possible values for the HW.

A "red" palette (brewer) will be used to show the HW value :

- HW = 0 -> trace color is dark red
- ……..
- HW = 8 -> trace color is light red

From the section where we located the AES operations and exactly from identifying the offset of 16 SBox operations, we can put the plot_start at 1472 and the plot_end at 1528 for the 1st SBox in order to leak the first byte of the key :

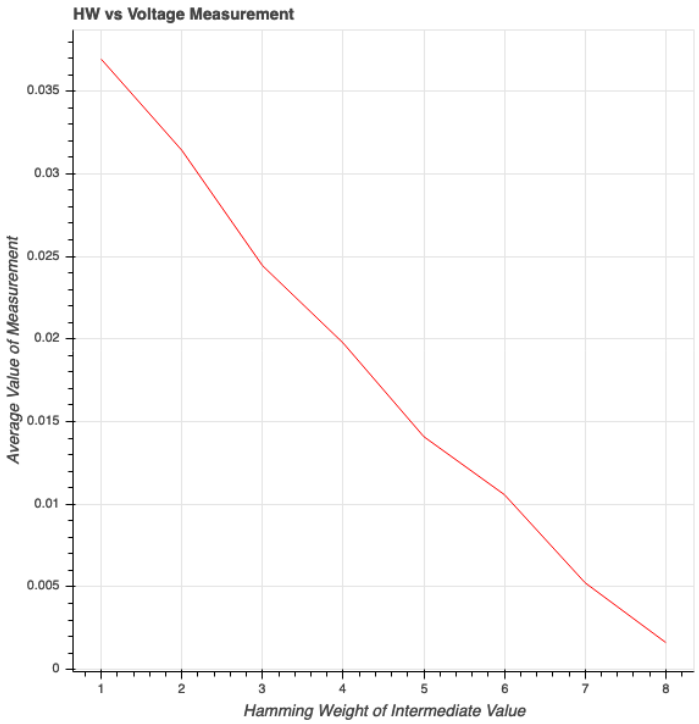
```
from bokeh.plotting import figure, show
from bokeh.io import output_notebook
from bokeh.palettes import brewer

output_notebook()
p = figure(plot_width = 1000, plot_height = 400)

bnum = 0   # change bnum to play with different key byte (from 0 to 15)

plot_start = 1472 # set here the sample where to start for the selected key byte
plot_end = 1528  # set here the sample where to end for the selected key byte

xrange = range(len(traces[0].wave))[plot_start:plot_end]

color_mapper = (brewer['Reds'][9])

for trace in traces:
    hw_of_byte = HW[intermediate(trace.textin[bnum], trace.key[bnum])]
    p.line(xrange, trace.wave[plot_start:plot_end], line_color=color_mapper[hw_of_byte])

show(p)
```



There is more than one point with a smooth degradation of red. After zooming a little bit, we decided to choose the one before the 1510 sample, which is 1508.

But how do we know if the chosen point is correct ?

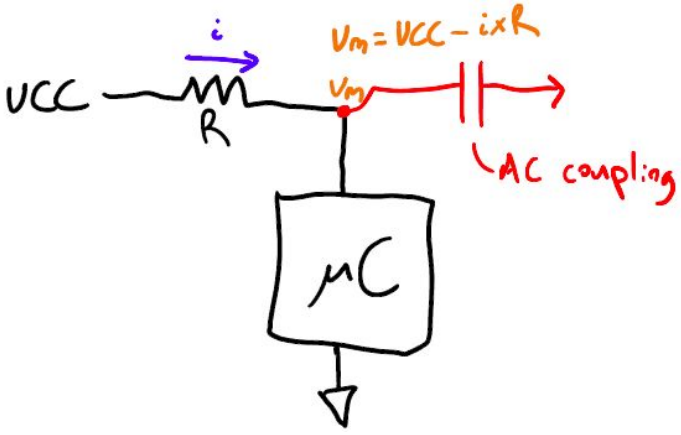
Let's plot the averages of HW points at the chosen point, which is 1508.



We got a linear plot, which means that the chosen point is the right one.

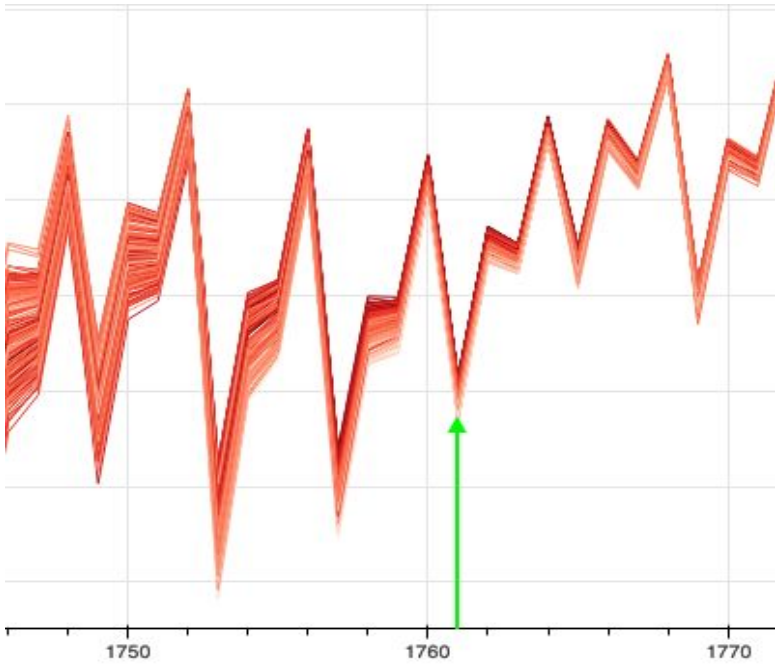
You might have noticed that the slope is opposite what we expected ! why is that ?

The figure below shows how the measurement process is done on the Chipwhisperer :



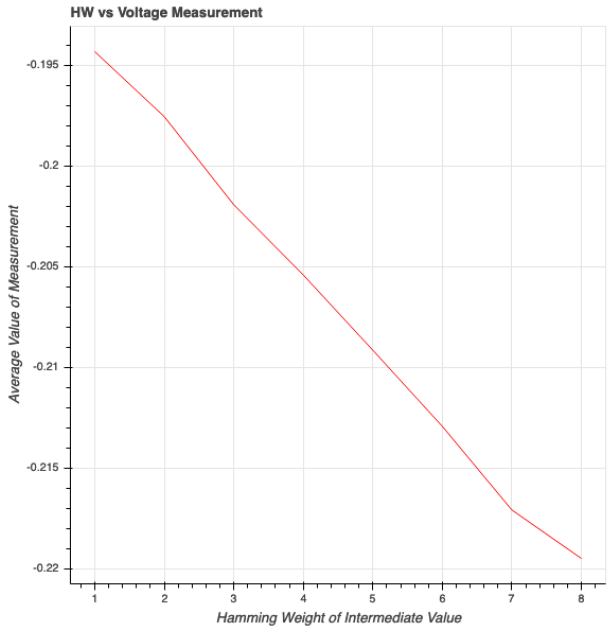
We are measuring the drop across the shunt resistor. An increase in the current causes a higher voltage across the resistor. When no current flows there is no drop across the resistor. But since we only measure a single end of the resistor, we see a higher voltage when no current flows.

Next, we want to make sure that our methodology is correct and for that we will re do the same process for the next range (1724 , 1780) and try to spot the leakage point.



We observe a smooth, beautiful gradation of red somewhere between 1760 and 1768. We'll pick the 1761-th point.

We then plot the averages, as we did before.



And voilà, the long-awaited linear plot.

We have to follow the same process and find all leakage points for all rages that we defined. However, this is a long process to follow and for that we have implemented a function that finds for us all the points that are lineare on a specific range :

```
import numpy as np

leakage_ranges = [ \
    range(1472 , 1528), range(1724 , 1780), range(1976 , 2032), range(2228 , 2284),
    range(1528 , 1584), range(1780 , 1836), range(2032 , 2088), range(2284 , 2340),
    range(1584 , 1640), range(1836 , 1892), range(2088 , 2144), range(2340 , 2396),
    range(1640 , 1696), range(1892 , 1948), range(2144 , 2200), range(2396 , 2452)
]

def is_linear(l):
    return all(l[i] >= l[i+1] for i in range(len(l)-1)) or all(l[i] <= l[i+1] for i in range(len(l)-1))


def find_leakage_points() :
    bnum=0
    for r in leakage_ranges :
        leakage_points = []
        for avg_point in r :
            hw_list = [ [], [], [], [], [], [], [], [], [] ]
            for i in range(numtraces):
                hw_of_byte = HW[intermediate(pt[i][bnum], key[bnum])]
                hw_list[hw_of_byte].append(traces[i][avg_point])
            hw_mean_list = [np.mean(hw_list[i]) for i in range(0, 9)]
            if is_linear(hw_mean_list) :
                leakage_points.append(avg_point)

        print("Leakage point for SBox{}: {}".format(bnum,leakage_points))
        bnum += 1

find_leakage_points()
```

```
Leakage point for SBox byte 0:  [1505, 1506, 1507, 1508, 1510, 1511, 1512, 1513, 1514, 1515]
Leakage point for SBox byte 1:  [1761, 1762, 1763, 1764, 1765, 1766, 1767, 1768]
Leakage point for SBox byte 2:  [2010, 2011, 2012, 2020, 2021, 2025, 2026]
Leakage point for SBox byte 3:  [2263, 2264, 2270, 2271, 2272, 2273, 2277, 2278]
Leakage point for SBox byte 4:  [1561, 1562, 1563, 1564]
Leakage point for SBox byte 5:  [1815, 1825]
Leakage point for SBox byte 6:  [2066, 2067, 2068]
Leakage point for SBox byte 7:  [2320, 2321, 2322, 2323, 2324, 2326, 2327]
Leakage point for SBox byte 8:  [1618, 1619, 1620, 1623, 1624, 1626, 1627, 1628]
Leakage point for SBox byte 9:  [1873, 1874, 1875, 1876]
Leakage point for SBox byte 10: [2121, 2122, 2123, 2124, 2126, 2127, 2128, 2129, 2130, 2131, 2132, 2137]
Leakage point for SBox byte 11: [2381, 2382]
Leakage point for SBox byte 12: [1678, 1679, 1680, 1682, 1683]
Leakage point for SBox byte 13: [1930, 1931, 1932, 1933]
Leakage point for SBox byte 14: [2184, 2193]
Leakage point for SBox byte 15: [2432, 2433, 2434, 2435, 2436, 2437, 2438]
```

As we've seen before, there can be multiple leakage points, more precisely an interval of points. We choose one of them, which represents the output of the S-box and use it to find the key before it fades out in the following AES rounds.

Performing the DPA Attack

Attack code

At this point, we have enough information to recover the AES key. We need to find a way to derive the value of this bytes using the Hamming Weight of the result of the SBox operation which, as we've seen, has a measurable effect on the power consumed by the microcontroller.

As we said on the DPA Attack theory section, we will need to capture enough traces with same key and different plaintext :

```
from tqdm import tnrange
import numpy as np
import time

def capture_traces() :
    scope.adc.samples = 3000
    scope.adc.offset = 0

    ktp = cw.ktp.Basic() # object dedicated to the generation of key (fixed by default) and plain text

    traces = [] # list of traces
    N = 8000  # Number of traces


    for i in tnrange(N, desc = 'Capturing traces'):
        key, text = ktp.next()  # creation of a pair comprising (fixed) key and text
        trace = cw.capture_trace(scope, target, text, key) # a trace is composed of the following fields :
                                                           #    a wave (samples)
                                                           #    textin (input text), textout (output text)
                                                           #    key (input key)
        if trace is None:
            continue
        traces.append(trace)
    return traces
```

This capture process takes time to finish. Because of that we want to save the traces into files and load them everytime we want to perform an attack :

```
def store_traces():
    traces = capture_traces()

    trace_array = np.asarray([trace.wave for trace in traces])
    textin_array = np.asarray([trace.textin for trace in traces])
    known_keys = np.asarray([trace.key for trace in traces])

    np.save("DPA_traces.npy", trace_array)
    np.save("DPA_textin.npy", textin_array)
    np.save("DPA_keys.npy", known_keys[0])

def load_traces() :
    traces = np.load(r'./DPA_traces.npy')
    pt     = np.load(r'./DPA_textin.npy')
    key    = np.load(r'./DPA_keys.npy')
    return traces, pt, key
```

Here is our final Attack code :

```python
import numpy as np

leakage_points = [1508, 1761, 2024, 2260,                                   # set the leakage points recovered from the HW experiece
                  1557, 1813, 2110, 2321,
                  1615, 1870, 2127, 2376,
                  1673, 1925, 2182, 2429 ]

def DPA_Attack(traces, pt) :

    threshold = 4                                                           # set the threshold to differentiate 2 groups
    recovered_key = []                                                      # list for all the recovered SubKeys

    for bnum in range(16):                                                  # loop through all SubKeys
        mean_diffs = np.zeros(256)                                          # store mean differences for each guess from 0 to 255
        maximum = 0                                                         # used to find the largest mean difference

        avg_point = leakage_points[bnum]                                    # set the leakage point for the actual SubKey

        for guess in range(0, 256):                                         # loop through all the possible key  guesses
            group1 = []                                                     # intiliaze 2 groups for each guess
            group2 = []

            for trace_index in range(len(traces)):                          # loop through all the traces
                hw_of_byte = HW[intermediate(pt[trace_index][bnum], guess)] # calculate the HW of SBox output for the actual guess
                if hw_of_byte < threshold:                                  # if the HW is under the threshold
                    group1.append(traces[trace_index][avg_point])           # add it to group1 ( low consumption group )
                else:                                                       # else
                    group2.append(traces[trace_index][avg_point])           # add it to group2 ( high consumption group )

            group1_avg = np.asarray(group1).mean(axis=0)                    # calculate the mean for group1
            group2_avg = np.asarray(group2).mean(axis=0)                    # calculate the mean for group2
            mean_diffs[guess] = np.max(abs(group1_avg - group2_avg))        # calculate the mean difference for the 2 groups
                                                                            # and store it at index of "guess"
            if mean_diffs[guess] > maximum :                                # if the mean diff is higher than the maximum mean diff
                maximum = mean_diffs[guess]                                 # update the maximum mean diff
                best_guess = guess                                          # update the best guess to the actual guess

        recovered_key.append(best_guess)                                    # add the best guess to the list of recovered key

    return recovered_key

if __name__ == '__main__' :
    traces, pt, key = load_traces()                                         # load the necessary parameters : traces, textin, key
    recovered_key = DPA_Attack(traces, pt)                                  # perform DPA Attack and recover the key

    print("Recovered key: {}".format(recovered_key))                        # compare the recovered key
    print("Correct   key: {}".format(list(key)))                            # with    the correct   key
```

Once executed :

```
Recovered key: [43, 126, 21, 22, 40, 174, 210, 166, 171, 247, 21, 136, 9, 207, 79, 60]
Correct   key: [43, 126, 21, 22, 40, 174, 210, 166, 171, 247, 21, 136, 9, 207, 79, 60]
```

It seems like we have successfully recovered the correct key !

Technical explanation

By now, we have a lot of plaintexts and their associated traces. Also we know the approximative points corresponding to the S-box outputs on the traces.

For each SubKey from the master key and for all the traces at the associated leakage point, we will make a sub key guess and calculate the Hamming Weight of the guess we made and the associated sub plaintext of that part (from line 13 to line 24).

Since we proved that the power consumption is linear to the HW, we gonna use this fact as a distinguisher of 2 groups :

- group1 : low power consumption group (HW is under a certain threshold)
- group2 : high power consumption group (HW is above a certain threshold)

The best guess (correct subkey) will have the highest mean difference between the values of these 2 groups. Why is that ?

Let's print these values (group1 mean, group2 mean, mean difference) for both a correct subkey and a wrong one, then we can compare them manually :

```
for trace_index in range(len(traces)):
    hw_of_byte = HW[intermediate( pt[trace_index][bnum], guess)]
    if hw_of_byte < threshold:
        group1.append(traces[trace_index][avg_point])
    else:
        group2.append(traces[trace_index][avg_point])

group1_avg = np.asarray(group1).mean(axis=0)
group2_avg = np.asarray(group2).mean(axis=0)
mean_diffs[guess] = np.max(abs(group1_avg - group2_avg))

if bnum == 0 and guess == 43 :
    print("correct sub key -> group1 mean: {}".format(group1_avg))
    print("correct sub key -> group2 mean: {}".format(group2_avg))
    print("correct sub key -> mean difference: {}".format(mean_diffs[guess]))

if bnum == 0 and guess == 72 :
    print("wrong sub key -> group1 mean: {}".format(group1_avg))
    print("wrong sub key -> group2 mean: {}".format(group2_avg))
    print("wrong sub key -> mean difference: {}".format(mean_diffs[guess]))

if mean_diffs[guess] > maximum :
    maximum = mean_diffs[guess]
    best_guess = guess
```

Once executed :

```
correct sub key -> group1 mean: -0.3112663223992706
correct sub key -> group2 mean: -0.3384356387912517
correct sub key -> mean difference: 0.027169316391981113

wrong sub key   -> group1 mean: -0.33032545371563576
wrong sub key   -> group2 mean: -0.32770482870825146
wrong sub key   -> mean difference: 0.002620625007384303
```

The mean difference of the correct sub key is 10 times bigger than the wrong one. Why is that too ?

Let's suppose we have used the correct subkey :

Correct SubKey ->
Correct HW of the SBox output ->
Correct distinguishing of 2 groups depending on the HW ->
points that will be added to group2 have higher HW ->
that means they have higher power consumption at that point -> [0.1, 0.2, ….]
points that will be added to group1 have lower HW ->
that means they have lower power consumption at that point -> [0.01, 0.02, ….]
means difference these 2 groups will be high : 0.2

Meanwhile with a wrong guess :

Wrong SubKey ->
Wrong HW of the SBox output ->
Wrong distinguishing of 2 groups depending on the HW ->
points that will be added to group2 are random -> [0.1, , 0.02, …]
points that will be added to group1 are random -> [0.01, 0.2, …]
means difference these 2 groups will be low : 0.02

That's how the mean difference is used to get the best guess for all

Sub Keys (from line 25 to line 38)

Attack correctness

In order to confirm that the attack works for any chosen key, we will choose our key :

```
fixed_key = bytearray.fromhex("000102030405060708090a0b0c0d0e0f")

for i in tnrange(N, desc = 'Capturing traces'):
    _ , text = ktp.next()  # creation of a pair comprising (fixed) key and text
    trace = cw.capture_trace(scope, target, text, fixed_key) # a trace is composed of the following fields :
```

Then we will run our attack :

```
if __name__ == '__main__' :
    store_traces()
    traces, pt, key = load_traces()
    start = timer()
    recovered_key = DPA_Attack(traces, pt)
    end = timer()

    print("Recovered key: {}".format(recovered_key))
    print("Correct   key: {}".format(list(key)))
    print("The attack took: {}".format(timedelta(seconds=end-start)))
```

```
Recovered key: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
Correct   key: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
The attack took: 0:01:51.283470
```

The attack succeeded and it took around 1mn51sec to finish.

Playing with the HW threshold and number of samples

In order to analyze our method and its performance, we are gonna try to play with HW threshold and the samples to be used on the key recovery process.

We did some adjustments to our code :

- Pass the correct key as a parameter
- Pass the the HW threshold as a parameter

```
def DPA_Attack(traces, pt, key, threshold=4) :
```

- Return the number of wrong keys extracted

```
        if best_guess != key[bnum] :
            wrong_keys += 1

    return wrong_keys
```

- Add a function that do some statistics on the correctness of the key recovery regarding the number of samples and the value of the threshold :

```
def statistics() :

    for threshold in range(3,6):
        for num_traces in range(40,140,20) :
            total = 16 * 100
            for i in range(100) :
                traces, pt, key = load_traces(i)
                wrong_keys = DPA_Attack(traces[:num_traces], pt, key, threshold)
                total  -= wrong_keys

            percentage = ( total / 1600 ) * 100

            print("Threshold : {}".format(threshold))
            print("Number of traces used : {}".format(num_traces))
            print("Percentage of correct key recovery out : {}%".format(percentage))
            print("_____________________________________________________")
```

We will need to capture 100 traces and then apply the statistics on them :

```
if __name__ == '__main__' :

    num_traces = 100
    for i in range(num_traces) :
        store_traces(i)
```

The result is shown here :

```
Threshold : 3
Number of traces used : 40
Percentage of correct key recovery out : 98.4375%
--------------------------------------------------
Threshold : 3
Number of traces used : 60
Percentage of correct key recovery out : 99.1875%
--------------------------------------------------
Threshold : 3
Number of traces used : 80
Percentage of correct key recovery out : 99.625%
--------------------------------------------------
Threshold : 3
Number of traces used : 100
Percentage of correct key recovery out : 99.875%
--------------------------------------------------
Threshold : 3
Number of traces used : 120
Percentage of correct key recovery out : 99.875%
--------------------------------------------------
Threshold : 3
Number of traces used : 140
Percentage of correct key recovery out : 99.9375%
--------------------------------------------------
Threshold : 3
Number of traces used : 160
Percentage of correct key recovery out : 100.0%
--------------------------------------------------
Threshold : 3
Number of traces used : 180
Percentage of correct key recovery out : 100.0%
--------------------------------------------------
```

```
Threshold : 4
Number of traces used : 40
Percentage of correct key recovery out : 99.625%
--------------------------------------------------
Threshold : 4
Number of traces used : 60
Percentage of correct key recovery out : 99.9375%
--------------------------------------------------
Threshold : 4
Number of traces used : 80
Percentage of correct key recovery out : 100.0%
--------------------------------------------------
Threshold : 4
Number of traces used : 100
Percentage of correct key recovery out : 100.0%
--------------------------------------------------
Threshold : 4
Number of traces used : 120
Percentage of correct key recovery out : 100.0%
--------------------------------------------------
Threshold : 4
Number of traces used : 140
Percentage of correct key recovery out : 100.0%
--------------------------------------------------
Threshold : 4
Number of traces used : 160
Percentage of correct key recovery out : 100.0%
--------------------------------------------------
Threshold : 4
Number of traces used : 180
Percentage of correct key recovery out : 100.0%
--------------------------------------------------
```

```
Threshold : 5
Number of traces used : 40
Percentage of correct key recovery out : 99.6875%
--------------------------------------------------
Threshold : 5
Number of traces used : 60
Percentage of correct key recovery out : 99.9375%
--------------------------------------------------
Threshold : 5
Number of traces used : 80
Percentage of correct key recovery out : 99.9375%
--------------------------------------------------
Threshold : 5
Number of traces used : 100
Percentage of correct key recovery out : 100.0%
--------------------------------------------------
Threshold : 5
Number of traces used : 120
Percentage of correct key recovery out : 100.0%
--------------------------------------------------
Threshold : 5
Number of traces used : 140
Percentage of correct key recovery out : 100.0%
--------------------------------------------------
Threshold : 5
Number of traces used : 160
Percentage of correct key recovery out : 100.0%
--------------------------------------------------
Threshold : 5
Number of traces used : 180
Percentage of correct key recovery out : 100.0%
--------------------------------------------------
```

In order to reach 100% (full correct key recovery) :

- With threshold 3, you will need at least 160 traces
- With threshold 4, you will need at least 80 traces
- With threshold 5, you will need at least 100 traces

We conclude that the best parameters are : **4 HW THRESHOLD & 80 SAMPLES**
And with these parameters, we will have a quicker key recovery (we went from 1mn 54 to 1 sec 54 !)

```
Recovered key: [127, 75, 9, 225, 178, 212, 84, 157, 108, 85, 160, 8, 255, 183, 0, 255]
Correct   key: [127, 75, 9, 225, 178, 212, 84, 157, 108, 85, 160, 8, 255, 183, 0, 255]
The attack took:  0:00:01.548099
```

Same attack different leakage model & distinguisher

In the previous attack, we used the Hamming Weight as the leakage model and the HW threshold as a distinguisher. There are other leakage models and distinguishers. LSB is one of the most known distinguisher

Everything is the same as the previous attack, except for the way of splitting the leakage points on 2 groups. This process will be based on the least significant bit of the SBox output with our guest and the plaintext as an input. if the LSB is 1 then we add it to group1 else to group2 :

```
destinguisher = intermediate( pt[trace_index][bnum], guess)
if destinguisher & 1:
    group1.append(traces[trace_index][avg_point])
else:
    group2.append(traces[trace_index][avg_point])
```

This distinguisher is based only on 1 bit of the SBox output which isn't good, this attack will rely more on probabilities than exploiting a side channel on the SBox output.

It supposes that with a good key guess, the mean difference of a group that has LSB at 1 and an other group that has LSB 0 will be bigger than the other wrong guesses :

```
if __name__ == '__main__' :
    store_traces()
    traces, pt, key = load_traces()
    start = timer()
    recovered_key = DPA_Attack(traces, pt)
    end = timer()

    print("Recovered key: {}".format(recovered_key))
    print("Correct   key: {}".format(list(key)))
    print("The attack took: {}".format(timedelta(seconds=end-start)))
```

```
Recovered key: [0, 1, 191, 3, 133, 164, 52, 7, 137, 9, 10, 11, 50, 13, 156, 253]
Correct   key: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
The attack took: 0:01:51.755840
```

This attack didn't totally succeed. It recovered half of the key (8 out of 16) and it took the same execution time as the previous attack.

To improve this attack :

- Replace the leakage points by leakage ranges

```
"""
leakage_points = [1505, 1760, 2010, 2262,\
                  1563, 1814, 2067, 2321,\
                  1618, 1871, 2124, 2376,\
                  1677, 1927, 2180, 2429]
"""

leakage_ranges = [ \
    range(1500 , 1515), range(1755 , 1765), range(2005 , 2015), range(2254 , 2278),
    range(1558 , 1569), range(1805 , 1820), range(2060 , 2075), range(2314 , 2326),
    range(1615 , 1625), range(1865 , 1878), range(2120 , 2130), range(2370 , 2384),
    range(1670 , 1680), range(1920 , 1934), range(2175 , 2185), range(2425 , 2435)
]
```

- Increase the number of traces from 1000 to 2000

```
if __name__ == '__main__' :
    traces, pt, key = load_traces()
    recovered_key = DPA_Attack(traces[:2000], pt)

    print("Recovered key: {}".format(recovered_key))
    print("Correct   key: {}".format(list(key)))
```

```
Recovered key: [98, 26, 193, 17, 198, 84, 39, 74, 54, 119, 124, 242, 117, 28, 136, 203]
Correct   key: [98, 26, 193, 17, 84, 39, 95, 74, 54, 119, 124, 242, 244, 28, 151, 203]
```

We notice that we recovered more keys (10 out of 16)

- We change our distinguished from LSB to MSB (Most Significant Bit)

```
msb = intermediate( pt[trace_index][bnum], guess)
if msb & 0x80:
    #group1.append(traces[trace_index][avg_point])
    group1.append(traces[trace_index][avg_range])
else:
    #group2.append(traces[trace_index][avg_point])
    group2.append(traces[trace_index][avg_range])
```

- Increase the number of samples to 3000 :

```
    traces, pt, key = load_traces()
    start = timer()
    recovered_key = DPA_Attack(traces[:num_traces], pt, destinguisher)
    end = timer()
    print("Recovered key: {}".format(recovered_key))
    print("Correct   key: {}".format(list(key)))
    print("The attack took:  {}".format(timedelta(seconds=end-start)))
```

```
Recovered key: [127, 75, 9, 225, 178, 212, 84, 157, 108, 85, 160, 8, 255, 183, 0, 255]
Correct   key: [127, 75, 9, 225, 178, 212, 84, 157, 108, 85, 160, 8, 255, 183, 0, 255]
The attack took:  0:02:36.620095
```

It seems like now, we are recovering the whole correct key.

The LSB/MSB distinguishers do work but they do rely a lot on parameters and probabilities (the more samples you use, the higher probably you will recover the correct key) and also they depend only on 1 bit, either the most significant one or the least significant one. We can adjust them to use more than 1 bit. MSB is better than LSB because the group that has the MSB at 1 will consume more power on the output of the SBox operation.

After all they will take so long to recover the key if you compare it to the HW Threshold distinguisher.

Conclusion

In the previous section, we presented results of practical tests sof the differential power analysis on the AES algorithm, implemented on the particular microcontroller in the C language and in the assembler.

The results were positive, we were able to reveal all 16 bytes of the secret key. The time needed for the attack was reduced to 1 sec using the HW threshold as a distinguisher and 2mn 36 seconds using LSB_MSB as a distinguisher. These timings are a negligible time in comparison with a naive brute force attack on this algorithm (2^16)

DPA attacks rely on the leakage models, distinguishers, number of traces and offsets !

A better choice of these parameters means a correct and a faster key recovery.

The problem with the DPA attack is that:

- It's quite susceptible to noise
- If you don't select the right offset and samples the attack can easily pick up other parts of the AES operation (you can try with different values to see this point)
- The attack typically requires a lot of traces. These software unprotected AES implementations are pretty weak against power analysis, but they still required thousands of traces to break

Can’t we improve these inconveniences on another variant of side channel attack ?

Introduction to CPA

Before we get into details of the Correlation Power Analysis (CPA) attack, we will need to change the way we capture, store and load traces.

This time we would have to store traces in their raw format (tess space and less time) :

```python
def store_traces():
    traces = capture_traces()

    traces_dir = 'traces'
    if not os.path.exists(traces_dir) :
        os.mkdir(traces_dir)

    trace_array = np.asarray([trace.wave for trace in traces])
    for i in range(10):
        print('%f' % trace_array[0][i])

    trace_array.astype('float').tofile(os.path.join(traces_dir, 'traces.raw'))
```

For example, we are capturing 100 traces of 3000 samples and storing them as raw array of floats inside a file ‘traces.raw’

Parsing traces

After we have stored the raw traces, we would have to use them in a C program, parse them and check if we are doing right by comparing to the Pythonish way :

```c
#define NB_SAMPLES 3000 // should be in a header file !

const char *path = "traces.raw";

double traces[NB_SAMPLES]; // only one trace here, to be adapted if needed

int main(int argc, char const *argv[]) {
  int fd;
  if ((fd = open(path, O_RDONLY)) == -1) {
    perror("open");
    return -1;
  }

  if (read(fd, traces, sizeof(double) * NB_SAMPLES) == -1) {
    perror("read");
    return -1;
  }

  for (int i = 0; i < 10; i += 1) {
    printf("%f\n", traces[i]);
  }
  return 0;
}
```

This code does open the raw file then read NB_SAMPLES of size double, then it prints the first 10 values.

We compile and execute the program :

```
(3.6.7/envs/cw) vagrant@stretch:~/work/projects/chipwhisperer/TME_SCA/traces$ ./parser
0.074219
-0.317383
-0.143555
-0.142578
0.012695
-0.414062
-0.215820
-0.200195
-0.026367
-0.472656
```

We compare it with the python way and it's the same values :

```
for i in range(10):
    print('%f' %trace_array[0][i])
```

```
0.074219
-0.317383
-0.143555
-0.142578
0.012695
-0.414062
-0.215820
-0.200195
-0.026367
-0.472656
```

At this point, we are asked to :

- Change the python code above in order to convert the paintext and the key in raw type, using astype('uint8'), and write them in separate files.
- Modify your script in order to generate raw files for a set of several traces with different random plaintexts
- The file containing the traces and the file containing the plaintext must have a length of nb_samples * nb_traces * sizeof(double) and 16 * nb_traces respectively

To solve all these problems in one shot, we opted for a specific type of file and we will be using just one, which is a database.

First, we define the structure of our database (Model), we will keep the same the structure of python trace :

```python
from peewee import Model, CharField, BlobField, SqliteDatabase, DatabaseProxy
import struct

KEY_LEN = 16

database = DatabaseProxy()

class Trace(Model):
    key = CharField(max_length=KEY_LEN)
    textin = CharField(max_length=KEY_LEN)
    wave = BlobField()

    class Meta:
        database = database

class Database:
    def __init__(self, file='traces.db'):
        self.db = SqliteDatabase(file)
        database.initialize(self.db)

    def fill_db(self, traces, nb_samples):
        self.db.create_tables([Trace])
        for wave, textin, _, key in traces:
            trace = Trace.create(
                    key    = key.hex(),
                    textin = textin.hex(),
                    wave   = struct.pack('d' * nb_samples, *wave))
            trace.save()
```

The code above does store 1 trace per row in a table of type Trace (key : char type, textin: char type, wave : Blob type)

The key and the plaintext are stored in their hexadecimal format and the wave is stored in Blob (Binary Large OBject) format which corresponds to the wave values in their float format.

Our goal now is to create a C library containing a function which will take as arguments an array of traces, the number of traces and the number of samples and which will perform the CPA attack.

Before digging into details we have to see the project structure :

```
.
|-- cpa                 # folder containg the header file and src file which performes CPA Attack
|   |-- cpa_attack.c    # cpa attack src file
|   `-- cpa_attack.h    # cpa attack header file
|-- database_utils.py   # contains the database model and the function which fills the DB rows
|-- launch_attack.py    # main file which lanches the cpa attack
|-- Makefile            # compile and create .so
`-- traces.db           # db file
```

The launch_attack is the front door to our cpa_attack. The number of traces, the offset and number of samples were asked to be modular, so we added some options to the set by the user.

```python
# user controlled parameters
db_file = "traces.db"
new_capture = False
nb_samples = 3000
nb_traces = 50
traces_start_offset = 0
use_all_wave = False

leakage_ranges = [ \
    (1505 , 1510), (1755 , 1765), (2005 , 2015), (2254 , 2278),
    (1561 , 1565), (1805 , 1820), (2060 , 2075), (2314 , 2326),
    (1615 , 1625), (1865 , 1878), (2120 , 2130), (2370 , 2384),
    (1670 , 1680), (1920 , 1934), (2175 , 2185), (2425 , 2435)
]
```

As we said before, we wish to select a given number of traces from the database into an array of traces which will be accepted as a parameter to a C function.

To do so, we define a structure, which we'll call `t_trace` in C:

```c
typedef union
{
  double* all;
  double* intervals[KEY_SIZE];
} u_wave ;

typedef struct s_trace
{
  u_wave     wave;
  int        textin[KEY_SIZE];
  int        key[KEY_SIZE];

}  t_trace;
```

We create the same structure using ctypes in python:

```python
class U_WAVE(Union):
    _fields_ = [
        ("intervals", POINTER(c_double) * KEY_LEN),
        ("all", POINTER(c_double)),
    ]

class TRACE(Structure):
    _fields_ = [
        ("wave", U_WAVE),
        ("textin", c_int * KEY_LEN),
        ("key", c_int * KEY_LEN)
    ]
```

The `textin` and `key` sizes are fixed and have a value of 16. As for `wave`, we have 2 cases :

- It can be either a whole wave from the start to the end of samples (0-3000 by default)
- Or it can be a list of ranges, since later we want to attack only specific parts of the wave.

For test proposes , we'll create a function in C which prints an array of traces:

```c
void print_array(int *array)                                // prints an array of size KET_SIZE
{
    for(int i = 0; i < KEY_SIZE; i++)
        printf("%d ", array[i]);
    printf("\n");
}
void print_samples(double **samples, int intervals[])       // prints a wave from trace
{
     int nr_samples = 0;
     for (int i = 0; i < KEY_SIZE; i++)
     {
           nr_samples = intervals[i];
           for(int j = 0; j < nr_samples; j++)
                printf("%.10f ", samples[i][j]);
           printf("\n");
     }
}

void print_traces(t_trace *tr, int nr_traces, int intervals[]) // prints the whole trace
{
    for(int i = 0; i < nr_traces; i++)
    {
        printf("Trace %d :\n", i);

        print_samples(tr[i].wave.intervals, intervals);
        printf("\n");
        print_array(tr[i].textin);
        print_array(tr[i].key);
    }
}
```

We'll perform the `make` command which creates our library `cpa_attack.so`.

After we created the library, we'll load it in python,

```python
# load C lib
try:
    lib = CDLL(pathlib.Path("cpa/cpa_attack.so").absolute())
except OSError as err:
    print("Make sure that `make` command was performed and the library was created. \nError : {}".format(err))
```

perform a query which selects from the database, the number of traces requested by the user,

```python
# perform a query which selects args.number_of_traces traces starting from args.traces_start_offset offset
traces_from_db = Trace.select().limit(args.number_of_traces).offset(args.traces_start_offset)
```

create a list of TRACES (wave, textin, key) from the output of the select query and use it as a parameter of the C `print_traces` function we saw above.

```
# defining python pointers to C functions
intervals = [ (end - start) for start, end in leakage_ranges]  # calculating the nb_samples for each interval
c_intervals = (c_int * KEY_LEN)(*intervals)                    # convert it to C int array

init_intervals = lib.init_intervals                            # python pointer to the C init_intervals function
init_intervals.restype = U_WAVE

fill_interval = lib.fill_interval                              # python pointer to the C fill_intervals function
fill_interval.restype = None

# creates a C wave from a binary wave stored in the db
def handle_intervals(c_intervals):

    wave = init_intervals(c_intervals)                         # allocate memory for all the intervals

    for i in range(len(leakage_ranges)):
        start, end = leakage_ranges[i]
        wave_bin = struct.unpack_from("d" * intervals[i], trace.wave, start * DOUBLE_SIZE)
        wave_bin = [round(x, 8) for x in wave_bin]             # convert the packed data inside the db to samples
        fill_interval(wave, i, (c_double * intervals[i])(*wave_bin), intervals[i])   # fill the wave structure

    return wave

# Create a list of C traces
traces = []

for trace in traces_from_db:                                   # create Traces in C from the Traces in the db
    wave = handle_intervals(c_intervals)                       # create a wave
    traces.append(TRACE(
        wave,
        (c_int * KEY_LEN)(*list(bytearray.fromhex(trace.textin))),   # convert from Python Array to C Array
        (c_int * KEY_LEN)(*list(bytearray.fromhex(trace.key))),      # convert from Python Array to C Array
    ))

traces = (TRACE * nb_traces)(*traces)
lib.print_traces(traces, nb_traces, c_intervals)
```

We will try to print in python and in C a trace to see if we have the same values.

```
! python3 launcher.py
```

```
Trace 0 :
        intervals[0]: -0.3017578100 -0.1289062500 -0.1357421900 0.0263671900 -0.2177734400
        intervals[1]: -0.1191406200 0.0507812500 -0.3193359400 -0.1416015600 -0.1367187500 0.0253906200 -0.20898438
00 -0.0449218800 -0.0644531200 0.0800781200
        intervals[2]: -0.2783203100 -0.0947265600 -0.0888671900 0.0732421900 -0.2597656200 -0.0937500000 -0.0996093
800 0.0566406200 -0.1875000000 -0.0351562500
        intervals[3]: -0.1015625000 -0.0957031200 0.0625000000 -0.3691406200 -0.1669921900 -0.1396484400 0.02832031
00 -0.2968750000 -0.1220703100 -0.1230468800 0.0390625000 -0.1875000000 -0.0263671900 -0.0566406200 0.0878906200 -
0.0341796900 0.0957031200 0.0439453100 0.1660156200 -0.0810546900 0.0693359400 0.0371093800 0.1708984400 -0.0771484


        intervals[15]: -0.3388671900 -0.1435546900 -0.1220703100 0.0498046900 -0.3056640600 -0.1279296900 -0.123046
8800 0.0410156200 -0.1816406200 -0.0283203100

        textin: 110 253 19 20 234 173 130 192 19 69 119 255 240 49 73 232
        key: 86 100 27 167 242 95 204 107 102 191 148 91 115 1 68 10
Trace 1 :
        intervals[0]: -0.2744140600 -0.1074218800 -0.1123046900 0.0390625000 -0.2021484400
        intervals[1]: -0.1279296900 0.0429687500 -0.3261718800 -0.1484375000 -0.1416015600 0.0234375000 -0.21582031
```

In python:

```
! python3 launcher.py
```

```
From python
Trace0:
        intervals[0]: [-0.30175781, -0.12890625, -0.13574219, 0.02636719, -0.21777344]
        intervals[1]: [-0.11914062, 0.05078125, -0.31933594, -0.14160156, -0.13671875, 0.02539062, -0.20898438, -0.
04492188, -0.06445312, 0.08007812]
        intervals[2]: [-0.27832031, -0.09472656, -0.08886719, 0.07324219, -0.25976562, -0.09375, -0.09960938, 0.056
64062, -0.1875, -0.03515625]
        intervals[3]: [-0.1015625, -0.09570312, 0.0625, -0.36914062, -0.16699219, -0.13964844, 0.02832031, -0.29687
5, -0.12207031, -0.12304688, 0.0390625, -0.1875, -0.02636719, -0.05664062, 0.08789062, -0.03417969, 0.09570312, 0.0
4394531, 0.16601562, -0.08105469, 0.06933594, 0.03710938, 0.17089844, -0.07714844]
```

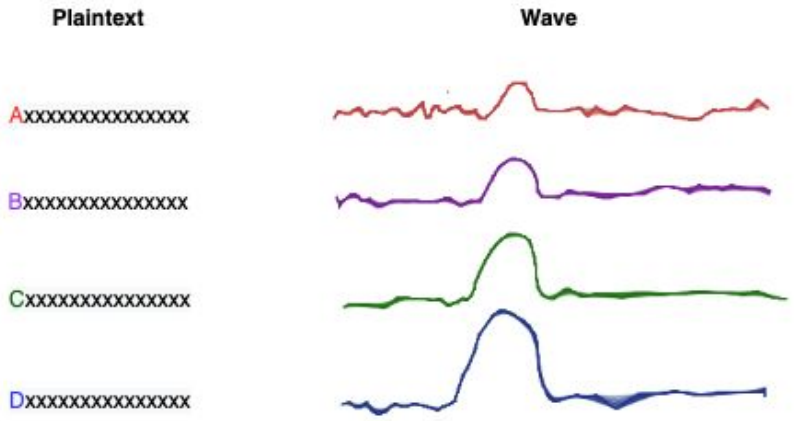
Looks like we have the correct values, so we can start to implement the CPA attack.

CPA attack theory

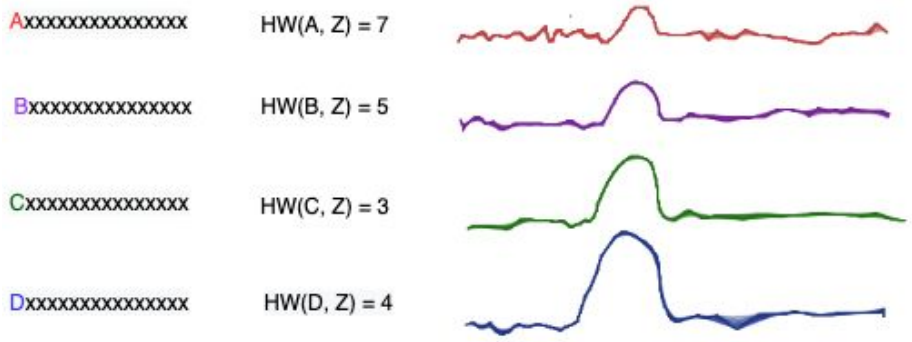
Attack illustration

The CPA attack is based on the correlation of the power consumption according to a chosen model. In our case, the chosen model is the Hamming Weight. Before we explain the mathematics, we'll take the time to visualise our goal. We will take the example of the first key byte and try to visualise the correlation we are searching for.

Supposing we have the following plaintexts and their associated waves of AES operations performed with the correct key byte.

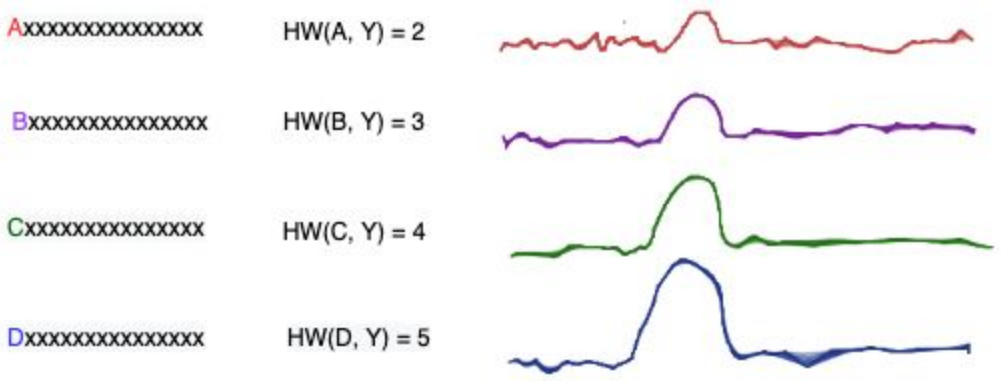


Now, we try to make a guess on the first key byte. We chose the first byte of the key to have a value of **Z** and we will compute the HW of the first byte of the plaintext xor-ed avec our guess **Z**.



We see that the HW outputs are not correlated with the waves. We can convince ourselves by taking the first plaintext's power consumption wave which is the smallest but has the greatest value of HW, which is clearly a sign of decorrelation .

Let's try another guess (the right one), which would be **Y** and compute the HW as we did for Z.



We can see that compared to the other guess, **Y** gets us HW values which are way more correlated with the waves.

Theoretical explanation

As we've seen from the attack illustration presented above, we have **two datasets:**

- **Waves** associated to the power consumption of AES operations of a set of plaintexts and the correct key
- The **Hamming Weights** of the given plaintexts xored with a key guess

We will try to measure the linear correlation between them.

To determine if there is a relationship and the strength of this relationship between these two datasets we'll use the **Pearson's correlation coefficient**. Pearson's correlation coefficient is considered to be the best method one can use to determine if there is an association between two datasets, because it is based on the method of **covariance**.

The **covariance**, in its turn**,** evaluates how much or to what extent two assets change together. This can sound sufficient to see if there is a relationship between our HWs and the waves. However the covariance used alone will only tell us if our assets tend to move in the same direction or to the inverse direction, but will not tell us if there is a dependency between our assets. That's why we have to use Pearson's correlation, because unlike the covariance, the correlation coefficient tells us not only that there is a relationship between two datasets, but also the strength of this relationship, the dependency between the two datasets and that's what we wish to know.

The Pearson's correlation can be expressed with the following formula:

$$\rho(X, Y) = \frac{\mathrm{Cov}(X, Y)}{\sigma_X \sigma_Y}$$

Where :

ρ **(X,Y)** - the correlation between two variables X and Y

Cov(X,Y) - the covariance between the variables X and Y

σ_X – the standard deviation of the X-variable

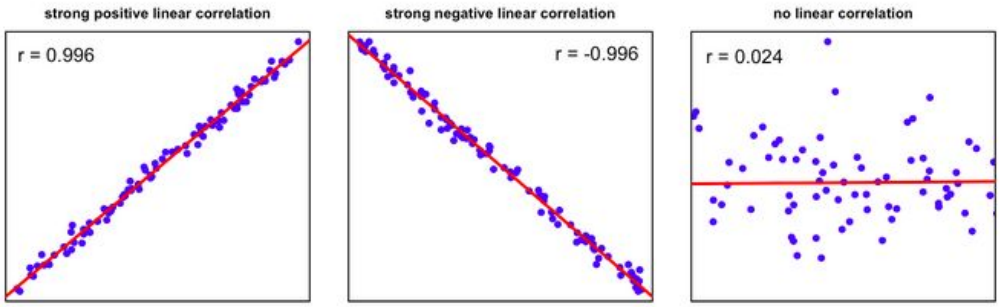
σ_Y – the standard deviation of the Y-variable

This formula can also be written as :

$$\rho(X,Y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}.$$

Where y stands for traces (waves) and x for hypothesis, respectively *x bar* and *y bar* are the values of the hypothesis mean and wave's mean.

The Pearson's coefficient will always return a value between **-1 and 1** and stronger the association of the two variables is, closer the Pearson correlation coefficient will be to either 1 or -1, respectively closer the output of the formula is to 0, stronger will be the variation between the two variables.



Implementing CPA

Now that we have all the theoretical tools to perform our attack, we can start to implement it following the steps below:

- Encrypt several plaintexts and record the waves representing the power consumption associated with these encryptions.
- Store the waves and plaintexts along with the encryption key in files/ database as we'll need them later to perform the attack
- Choose a model of power consumption, in our case Hamming Weight
- For each of the key's bytes, apply the power consumption model. For our example, compute the HW of the plaintext's each byte and the possible key's bytes hypothesis.
- For each key byte, apply the Pearson's correlation formula on the obtained hypothesis and the recorded waves(actual power consumption).
- Decide which subkey guess correlates best to the measured traces according to the Pearson's output interpretation explained above.
- Put together the best subkey guesses to obtain the full secret key

Attack code

First we need to capture some traces and store them in a database file :

```python
def create_db(new_capture, db_file='traces.db'):

    if os.path.exists(db_file) :
        os.remove(db_file)

    db  = Database(file=db_file)
    if new_capture:
        # capture traces for fixed key
        fixed_key = bytearray.fromhex(os.urandom(16).hex())
        print("[*] Capturing new traces with a fixed key: {}".format(list(fixed_key)))
        traces  = capture_traces(nb_samples, nb_traces, fixed_key)
        db.fill_db(traces, nb_samples)

nb_traces = 50
nb_samples  = 3000
new_capture = True
db_file     = "traces.db"
create_db(new_capture,db_file)
```

```
[*] Capturing new traces with a fixed key: [86, 100, 27, 167, 242, 95, 204, 107, 102, 191, 148, 91, 115, 1, 68, 10]
```

Capturing traces 100% 50/50 [00:05<00:00, 9.24it/s]

Once our database is filled with the captured traces, we need to load the traces from the database to the C Structure of the trace :

```python
for trace in traces_from_db:                                    # create Traces in C from the Traces in the db
    wave = handle_intervals(c_intervals)                        # create a wave
    traces.append(TRACE(
        wave,
        (c_int * KEY_LEN)(*list(bytearray.fromhex(trace.textin))),   # convert from Python Array to C Array
        (c_int * KEY_LEN)(*list(bytearray.fromhex(trace.key))),      # convert from Python Array to C Array
    ))

traces = (TRACE * nb_traces)(*traces)
```

After that we have allocated the needed space for C Traces, we can call the C function that performs the attack :

```python
# call the function that performs the attack
lib.cpa_attack(traces, nb_traces, c_intervals)
print("Correct   key: ", end='')
for i in range(KEY_LEN): print(correct_key[i],end=' ')
```

The CPA attack core code :

```c
int attack_byte(t_trace *tr, int nb_traces, int bnum, int nb_samples) {      // recover a subkey using cpa attack

    double corr_array[256];
    int best_guess = -1;

    for (int guess = 0; guess < 256; guess++)
        corr_array[guess] = corr_coef(tr, nb_traces, guess, nb_samples, bnum);    // for each guess calculate the best correlation

    best_guess = argmax_array(corr_array, 256);               // the correct subkey is the guess that has the maximum correlation
    return best_guess;
}

void cpa_attack(t_trace *tr, int nb_traces, int* intervals) {                     // recover all the subkeys using cpa attack
    int recovered_key[KEY_SIZE];

    for (int bnum = 0; bnum < KEY_SIZE; bnum++)                                   // for each subkey
        recovered_key[bnum] = attack_byte(tr, nb_traces, bnum, intervals[bnum]);    // apply cpa

    printf("Recovered key: ");
    print_array(recovered_key);
}
```

For each subkey we will apply correlation between the HW of all the guesses and the traces, the guess that has maximum correlation is our best guess and highly probable to be our subkey.

By applying this to all the subkeys, we will end up recovering the whole key.

Technical explanation

The whole attack relies on the calculation of the pearson's correlation coefficient, which is presented by this formula :

$$\rho(X,Y) = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}.$$

This form can be simplified to :

$$\rho(X,Y) = \frac{n\sum_{i=1}^{n} x_i y_i - \sum_{i=1}^{n} x_i \sum_{i=1}^{n} y_i}{\sqrt{n\sum_{i=1}^{n} x_i^2 - \left(\sum_{i=1}^{n} x_i\right)^2}\sqrt{n\sum_{i=1}^{n} y_i^2 - \left(\sum_{i=1}^{n} y_i\right)^2}}$$

[Incremental Pearson]

We notice that all the sums are done over `n` which represents the number of traces. We can exploit this to calculate them in one loop :

```
for (int tnum=0; tnum<nb_traces; tnum++)
    hypothesis[tnum] = HW[ intermediate(traces[tnum].textin[bnum], guess) ];
    // defining our hypothesis : for all traces, calculate the HW using 'guess' and 'traces[i].plaintext[bnum]'

hypothesis_mean = mean(hypothesis, nb_traces);                                  // calculating the mean of our hypothesis
mean_columns(traces, traces_col_means, nb_traces, nb_samples, bnum);

for (int tnum=0; tnum<nb_traces; tnum++) {          // https://eprint.iacr.org/2015/260.pdf  (4- Incremental Pearson)
    hypothesis_diff = (hypothesis[tnum] - hypothesis_mean);
    sub_arrays( traces[tnum].wave.intervals[bnum],  traces_col_means, traces_col_diff, nb_samples ); // ∑ni=1 yi

    init_array(result, nb_samples, hypothesis_diff);                        // ∑ni=1 xi
    mul_arrays(traces_col_diff, result, result, nb_samples);                // ∑ni=1 xi . ∑ni=1 yi
    add_arrays(COV_X_Y, result, COV_X_Y, nb_samples);                       // n . ∑ni=1 xi . yi - ∑ni=1 xi . ∑ni=1 yi  = COV(X,Y)

    init_array(result, nb_samples, hypothesis_diff * hypothesis_diff );     // ∑ni=1 xi^2
    add_arrays(STD_DEV_X, result, STD_DEV_X, nb_samples);                   // n . ∑ni=1 xi^2 - ( ∑ni=1 xi )^2 = Deviation(X)


    mul_arrays(traces_col_diff, traces_col_diff, result, nb_samples);
    add_arrays(STD_DEV_Y, result, STD_DEV_Y, nb_samples);                   // n . ∑ni=1 yi^2- ( ∑ni=1 yi )^2 = Deviation(Y)

}
```

Once the sums are calculated, we divide the numerator (Covariance(X,Y)) by the square root of the multiplication of the 2 denominators (Standard_Deviation(X), Standard_Deviation(Y)) :

```
mul_arrays(STD_DEV_X, STD_DEV_Y, result, nb_samples);                   // Deviation(Y) . Deviation(Y)
sqrt_array(result, result, nb_samples);                                 // SQRT( Deviation(Y) . Deviation(Y) )
div_arrays(COV_X_Y, result, result, nb_samples ) ;                      // COV(X,Y) /  SQRT( Deviation(Y) . Deviation(Y) )
```

The pearson's correlation coefficient is the maximum of the last result :

```
return max_array(result, nb_samples);
```

Attack correctness

Correlation applied only on leakage intervals of the wave

Now that our code is ready, let's launch an attack using 50 traces and we will target only leakage ranges of the wave :

```
! python3 launcher.py

Recovered key: 86 100 27 167 242 95 204 107 102 191 148 91 115 1 68 10
Correct   key: 86 100 27 167 242 95 204 107 102 191 148 91 115 1 68 10
The attack took:  0:00:00.071484
```

Impressive ! we recovered the whole key in less than a second, it takes around 71 milliseconds.

If we decrease the number of samples to 25 :

```
! python3 launcher.py

Recovered key: 7 240 201 65 126 219 172 175 160 104 176 199 84 96 36 195
Correct   key: 7 240 132 65 126 219 172 175 97 104 176 199 84 96 36 195
The attack took:  0:00:00.038499
```

It takes less time but recovers 14 correct subkeys out of 16.

Using 35 seems to be the threshold of number of traces to use :

```
! python3 launcher.py

Recovered key: 76 113 158 49 243 224 150 51 228 148 199 44 3 201 183 54
Correct   key: 76 113 158 49 243 224 150 51 228 148 199 44 3 201 183 54
The attack took:  0:00:00.051257
```

When attacking only the leakage ranges of the wave, we need to use 35 traces in order to recover the whole key correctly and that takes 51 milliseconds.

Correlation applied on the whole wave

Now, we want to apply the correlation on the whole wave :

```
! python3 launcher.py

Recovered key: 8 133 244 241 0 109 105 151 116 122 48 63 130 45 43 101
Correct   key: 8 133 244 241 0 109 105 151 107 122 48 63 130 45 43 101
The attack took:  0:00:17.663090
```

It recovered 15 subkeys out of 16 in 17 seconds using 50 traces.

We play a little bit with the number of traces and then we concluded that with 50 trace the 9th SubKey isn't correctly recovered but adding 1 trace will fix the problem :

```
! python3 launcher.py

Recovered key: 102 242 221 223 129 58 182 166 234 120 196 162 141 147 164
Correct   key: 102 242 221 223 129 58 182 166 234 120 196 162 141 147 164
The attack took:  0:00:17.838923
```

When attacking with the whole wave instead of leakage ranges, we need to use 51 traces in order to recover the whole key correctly and that takes around 17 seconds.

DPA vs CPA

As a bonus, we have implemented DPA attack in C, we basically translated the Python Code we already had to C version:

```c
int dpa_attack_byte(t_trace *tr, int nb_traces, int bnum)
{

    double mean_diffs[256];  init_array(mean_diffs, 256, 0.0);
    double threshold = 4.0;

    double group1[nb_traces]; int g1_index; double g1_avg = 0;
    double group2[nb_traces]; int g2_index; double g2_avg = 0;

    int avg_point = -1;
    double hw = 0;
    double maximum = 0;
    int best_guess = -1;

    for (int guess = 0; guess < 256; guess++)
    {

        init_array(group1, nb_traces, 0.0);
        init_array(group2, nb_traces, 0.0);
        g1_index = 0;
        g2_index = 0;

        for (int tnum = 0; tnum < nb_traces; tnum++ )
        {

            avg_point = dpa_leakage_points[bnum];
            hw = HW[ intermediate(tr[tnum].textin[bnum], guess) ];

            if ( hw < threshold )
                group1[g1_index++] = tr[tnum].wave.all[avg_point];
            else
                group2[g2_index++] = tr[tnum].wave.all[avg_point];
        }

        g1_avg = mean(group1, g1_index);
        g2_avg = mean(group2, g2_index);

        mean_diffs[guess] = fabs(g1_avg-g2_avg);
        if (mean_diffs[guess] > maximum ) {
            maximum = mean_diffs[guess];
            best_guess = guess;
        }
    }

    return best_guess;
}

void dpa_attack(t_trace *tr, int nb_traces)
{
    int recovered_key[KEY_SIZE];

    for (int bnum = 0; bnum < KEY_SIZE; bnum++)
        recovered_key[bnum] =  dpa_attack_byte(tr, nb_traces, bnum);

    printf("Recovered key: ");
    print_array(recovered_key);
}
```

We execute DPA with HW Threshold as a distinguisher :

```
if use_all_wave :
    start = timer()
    #lib.cpa_attack(traces, nb_traces, c_intervals, 1)
    lib.dpa_attack(traces, nb_traces)
    end   = timer()
else :
    start = timer()
    lib.cpa_attack(traces, nb_traces, c_intervals, 0)
    end   = timer()

print("Correct   key: ", end='')
for i in range(KEY_LEN): print(correct_key[i],end=' ')

print("\nThe attack took:  {}".format(timedelta(seconds=end-start)))
```

```
Overwriting /home/vagrant/work/projects/chipwhisperer/TME_SCA/TME4/launcher.py
```

2.0.7 ATTACK_LAUNCHER_CELL

```
! python3 launcher.py
```

```
Recovered key: 102 242 221 223 129 58 182 147 234 120 196 162 141 116 219 0
Correct   key: 102 242 221 223 129 58 182 166 234 120 196 162 141 147 164 0
The attack took:  0:00:00.003835
```

Using 40 traces, we recovered 14 subkeys correct out of 16 in 3 milliseconds.

Playing with the number of traces , we found that :

```
! python3 launcher.py
```

```
Recovered key: 65 178 246 77 154 88 34 136 78 77 13 19 134 94 109 105
Correct   key: 65 178 246 77 154 88 34 136 78 77 13 19 134 94 109 105
The attack took:  0:00:00.005770
```

When attacking with DPA and Hamming Weight as a distinguisher, we need to use 60 traces in order to recover the whole key and that takes 5 milliseconds.

Overall comparison :

	Destinguisher	Full recovery	Time In Python	Time In C	Number of traces	Advantage	Disadvantage
DPA	HW Threshold	YES	1 sec 54	5 ms	80	- Fast - Few traces	- Leakage points needed
	LSB	NO (14/16)	5 mn 55		8000		- A lot of traces - Blind probability
	MSB	YES	2 mn 36		3000		- Medium traces - Hard probability
CPA	HW Model (Leakage Intervals)	YES		54 ms	35	- Fast - No exact offset - Fewer traces	- Leakage intervals needed
	HW Model (All wave)	YES		17 sec	51	- Quick - No exact offset - Few traces	

From this comparison table, we can say that :

- CPA Attack using the whole wave is the best
- DPA Attack using HW with leakage points is the fastest but it's not convenient to search each time for the offsets and number of samples for each SBox operation
- CPA Attack using leakage ranges is also good if you can spot the ranges easily this will make win some seconds
- DPA Attack using LSB as a distinguisher is the worst one, because it requires a lot of traces, takes a lot of time and doesn't fully recover the key.

Conclusion

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.

For both attacks (DPA & CPA), we succeeded in revealing the secret key of AES. A Noticeable difference between the two attacks is that the CPA attack is more stable and less dependent on finding the correct key .

The LSB model only considers the last bit of a byte and ignores all other bits. The Hamming weight model/destinguisher which is used for the CPA/DPA attack does include all bits, which makes CPA/DPA more efficient and more accurate.

The correlation coefficient considers more aspects of statistics, which leads to a better comparison between the columns of the matrices and that makes CPA way stable than DPA.

Unfortunately in the security world there are no such hundred percent perfect countermeasures. Therefore the objective of a countermeasure against power analysis is to make an attack extremely difficult. When the time and the cost for an attack is large enough such that it would be infeasible for an attacker, such a countermeasure can be considered good enough.