

Implementation of DNN side channel attack using AES-128

J.SWATHI, AP18110020069

P.POOJA, AP18110020133

S.MURALI MOHAN, AP18110020099

V.PAVAN SAI, AP18110020123

V.ROHI KARTHIK, AP18110020120

ECE-B

Introduction:

In this present world scenario, cryptography is most and must in many cases for the protection of the data i.e. is being transmitted, cryptographic algorithms are integrated with the devices for the security and integrity of the data. Here in these algorithms which can't be forced to get into but the flaw remains in this physical platform as it can leak the critical information in various ways of electromagnetic radiation, timing, power consumption, and so on. While the power can be assumed in the Side channel Attacks(SCA), we here focus on improvising the platform that prevents the SCA by demonstrating the Attacks and their prevention by the deep learning-based.

Method of Approach:

Advanced Encryption Standard (AES128) which is a DNN model of the DLSCA(Deep Learning Side-Channel Attacks) is an algorithm that is trained by giving various key bytes for the tracing as it's training.

The test chip in AES128 DNN traces out the power traces of the device, these powers are transmitted as the 128byte or byte in fixed plain text (PT) by varying the 1st key byte and labeling each trace with a key byte value.

In this, the unsigned cases are fed into the training of DNN for the correct key byte.

With a few thousand traces, the trained DNN model of DLSCA knows which trace can be easily feasible in a single trace, increasing the threat surface significantly.

With the signal-to-noise ratio of the Minimum trace, Disclosure is inversely proportional to the inverse of the SNR square. as it is motivated by the correlated current signature significantly.

ADVANCED ENCRYPTION STANDARD(AES)

- AES is an encryption standard chosen by the NIST, USA to protect classified information.
- It is a block cipher that operates on a block size of 128 bits for both encrypting and decrypting the data.
- To encrypt the data the AES needs to perform the rounds.
- Each round performs the same operations.
- The AES key size varies between 128,192 and 256 bits.

KEY SIZE(in bits)	ROUNDS
128	10
192	12
256	14

- AES repeats 4 major functions to encrypt data. It takes a 128-bit block of data and a key and gives a ciphertext as output. The functions are

1. Byte substitution.
2. Shift rows.
3. Mix columns.

4. Key addition

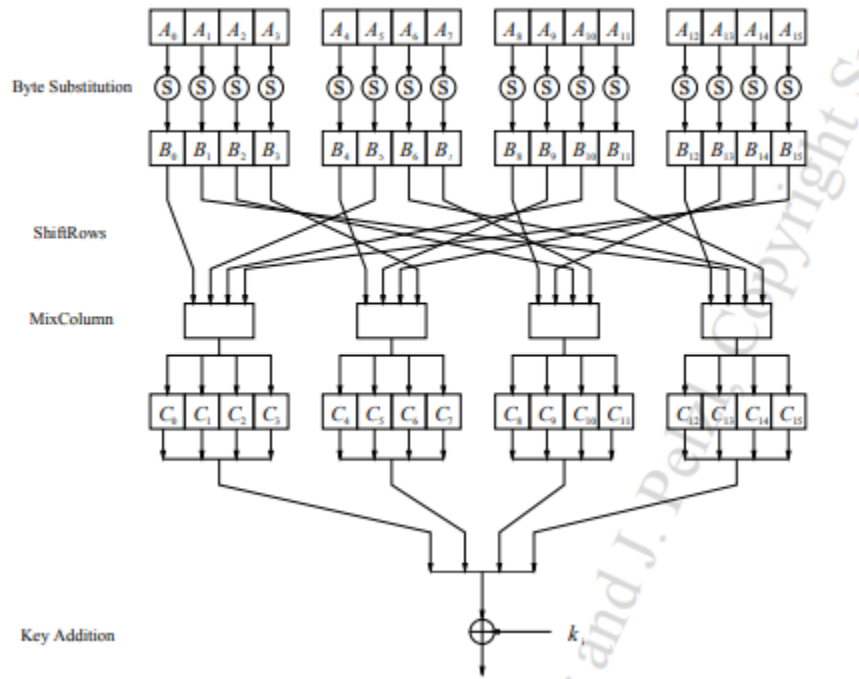


Fig: Structure of AES Encryption

To understand how the data moves through AES, we first imagine that the state A (i.e., the 128-bit data path) consisting of 16 bytes A_0, A_1, \dots, A_{15} is arranged in a four-by-four byte matrix.

A_0	A_4	A_8	A_{12}
A_1	A_5	A_9	A_{13}
A_2	A_6	A_{10}	A_{14}
A_3	A_7	A_{11}	A_{15}

Byte substitution layer:

In the sub-bytes step, we replace each byte of the state with another byte depending on the key. These substitutions are presented as look-up tables in the s-box. This s-box consists of 256-byte substitution arranged in a 6x16 grid. In this layer, each state byte A_i is replaced, i.e., substituted, by another byte B_i :

$$S(A_i) = B_i .$$

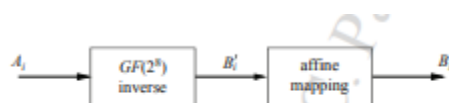
In software implementations, the S-Box is usually realized as a 256-by-8 bit lookup table with fixed entries, as given in the able

AES S-Box: Substitution values in hexadecimal notation for input byte (xy)

	y															
	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Mathematical description of s-box:

An AES S-Box can be viewed as a two-step mathematical transformation



The two operations within the AES S-Box which computes the function $B_i = S(A_i)$

A finite field, sometimes also called Galois field, is a set with a finite number of elements. Roughly speaking, a Galois field is a finite set of elements in which we can add, subtract, multiply and invert.

Inversion in $GF(2^8)$ is the core operation of the Byte Substitution transformation, which contains the AES S-Boxes. For a given finite field $GF(2^m)$ and the corresponding irreducible reduction polynomial $P(x)$, the inverse A^{-1} of a nonzero element $A \in GF(2^m)$ is defined as:

$$A^{-1}(x) \cdot A(x) = 1 \mod P(x).$$

In the affine mapping, each byte B'_i is multiplied by a constant bit matrix followed by the addition of a constant 8-bit vector as shown below,

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} \equiv \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} + \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} \mod 2.$$

Shift Rows layer:

In this layer the rows of the state matrix shift to the left. The first row is not shifted. The second row is shifted by 1 byte to the left and the third row is shifted by two bytes and the final row is shifted by 3 bytes. The purpose of the Shift Rows transformation is to increase the diffusion properties of AES. If the input of the Shift Rows sub-layer is given as a state matrix $B = (B_0, B_1, \dots, B_{15})$.

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

the output is the new state:

B_0	B_4	B_8	B_{12}	no shift
B_5	B_9	B_{13}	B_1	← one position left shift
B_{10}	B_{14}	B_2	B_6	← two positions left shift
B_{15}	B_3	B_7	B_{11}	← three positions left shift

Mix Column Sublayer:

The Mix Column step is a linear transformation that mixes each column of the state matrix. Every input byte influences four output bytes, the Mix Column operation is considered as the major diffusion element in AES. We denote the 16-byte input state by B and the 16-byte output state by C:

$$\text{Mix Column (B)} = \text{C}.$$

Now, each 4-byte column is considered as a vector and multiplied by fixed 4×4 matrices. The matrix contains constant entries. Multiplication and addition of the coefficients are done in GF (2^8). For example

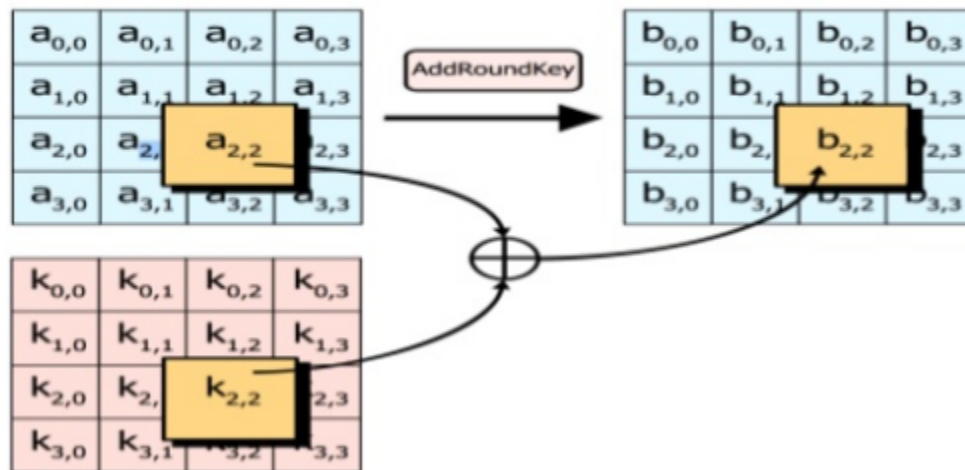
$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{pmatrix}.$$

The second column of output bytes ($C_4, 5, C_6, C_7$) is computed by multiplying the four input bytes (B_4, B_9, B_{14}, B_3) by the same constant matrix, and so on and all the outputs are computed.

Key Addition Layer:

The two inputs to the Key Addition layer are the current 16-byte state matrix and a subkey matrix. The two inputs are combined through a bitwise XOR operation. The AES with a 192-bit

key requires 13 subkeys of length 128 bits, and AES with a 256-bit key has 15 subkeys, subkey



AES DECRYPTION:

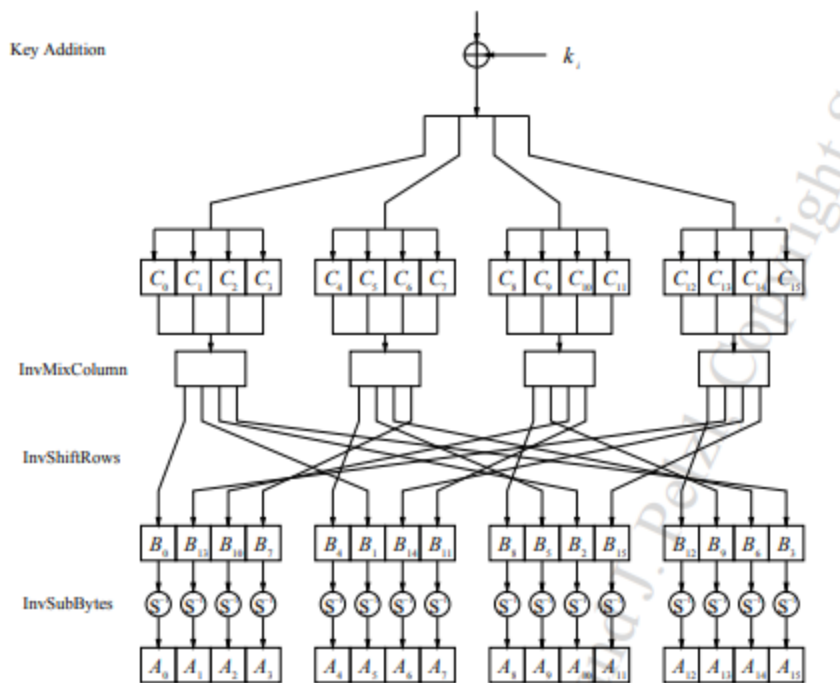


Fig: AES DECRYPTION

In the AES decryption all layers must be inverted, i.e., the Byte Substitution layer becomes the Inverse Byte Substitution layer, the Shift Rows layer becomes the Inverse Shift Rows layer, and the Mix Column layer becomes Inverse Mix Column layer. The last encryption round does not perform the Mix Column operation, so the first decryption round also does not contain the Mix Column inverse layer. All other decryption rounds contain all AES layers. The XOR operation is its inverse, the key addition layer in the decryption mode is the same as in the encryption mode that consists of a row of plain XOR gates.

Inverse Mix Column Sub-layer:

After the addition of the subkey, the inverse Mix-Column step is applied to the state. To reverse the Mix-Column operation, the inverse of its matrix must be used. The input is a 4-byte column of the State C which is multiplied by the inverse 4×4 matrix. The matrix contains constant entries.

The second column of output bytes (B_4, B_5, B_6, B_7) is computed by multiplying the four input bytes (C_4, C_5, C_6, C_7) by the same constant matrix, and so on and the output is computed.

$$\begin{pmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{pmatrix} = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix}$$

Inverse Shift Rows Sub layer:

To reverse the Shift Rows operation of the encryption algorithm, we must shift the rows of the state matrix in the opposite direction. The first row is not changed by the inverse Shift Rows transformation and the second row is shifted from one position to right and in the third row, I shifted two positions to the right and the fourth row is shifted to the three positions right. If the input of the Shift Rows sub-layer is given as a state matrix $B = (B_0, B_1, \dots, B_{15})$:

B_0	B_4	B_8	B_{12}
B_1	B_5	B_9	B_{13}
B_2	B_6	B_{10}	B_{14}
B_3	B_7	B_{11}	B_{15}

the inverse ShiftRows sublayer yields the output:

B_0	B_4	B_8	B_{12}	no shift
B_{13}	B_1	B_5	B_9	→ one position right shift
B_{10}	B_{14}	B_2	B_6	→ two positions right shift
B_7	B_{11}	B_{15}	B_3	→ three positions right shift

Inverse Byte Substitution Layer:

The inverse S-Box is used when decrypting a ciphertext. As the AES S-Box is a bijective, i.e., a one-to-one mapping, it is possible to construct an inverse S-Box such that:

$$A_i = S^{-1}(B_i) = S^{-1}(S(A_i))$$

where A_i and B_i are elements of the state matrix.

In the second step of the inverse S-Box operation, the Galois field inverse has to be reversed. For this, note that $A_i = (A^{-1} i)^{-1}$. This means that the inverse operation is reversed by computing the inverse again. In our notation, we thus have to compute $A_i = (B' i)^{-1} \in GF(2^8)$.

$$A_i = s^{-1}(B_i).$$

AES CODE :

```
#include <iostream>
```

```
#include <cstring>
```

```
using namespace std;
```

```
unsigned char s_box[256] =
```

```
{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};
```

```
unsigned char mul2[] = {
```

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

```
unsigned char mul3[] = {
```

```
    0x00,0x03,0x06,0x05,0x0c,0x0f,0x0a,0x09,0x18,0x1b,0x1e,0x1d,0x14,0x17,0x12,0x11,  
    0x30,0x33,0x36,0x35,0x3c,0x3f,0x3a,0x39,0x28,0x2b,0x2e,0x2d,0x24,0x27,0x22,0x21,  
    0x60,0x63,0x66,0x65,0x6c,0x6f,0x6a,0x69,0x78,0x7b,0x7e,0x7d,0x74,0x77,0x72,0x71,  
    0x50,0x53,0x56,0x55,0x5c,0x5f,0x5a,0x59,0x48,0x4b,0x4e,0x4d,0x44,0x47,0x42,0x41,
```

```
0xc0,0xc3,0xc6,0xc5,0xcc,0xcf,0xca,0xc9,0xd8,0xdb,0xde,0xdd,0xd4,0xd7,0xd2,0xd1,  
0xf0,0xf3,0xf6,0xf5,0xfc,0xff,0xfa,0xf9,0xe8,0xeb,0xee,0xed,0xe4,0xe7,0xe2,0xe1,  
0xa0,0xa3,0xa6,0xa5,0xac,0xaf,0xaa,0xa9,0xb8,0xbb,0xbe,0xbd,0xb4,0xb7,0xb2,0xb1,  
0x90,0x93,0x96,0x95,0x9c,0x9f,0x9a,0x99,0x88,0x8b,0x8e,0x8d,0x84,0x87,0x82,0x81,  
0x9b,0x98,0x9d,0x9e,0x97,0x94,0x91,0x92,0x83,0x80,0x85,0x86,0x8f,0x8c,0x89,0x8a,  
0xab,0xa8,0xad,0xae,0xa7,0xa4,0xa1,0xa2,0xb3,0xb0,0xb5,0xb6,0xbf,0xbc,0xb9,0xba,  
0xfb,0xf8,0xfd,0xfe,0xf7,0xf4,0xf1,0xf2,0xe3,0xe0,0xe5,0xe6,0xef,0xec,0xe9,0xea,  
0xcb,0xc8,0xcd,0xce,0xc7,0xc4,0xc1,0xc2,0xd3,0xd0,0xd5,0xd6,0xdf,0xdc,0xd9,0xda,  
0x5b,0x58,0x5d,0x5e,0x57,0x54,0x51,0x52,0x43,0x40,0x45,0x46,0x4f,0x4c,0x49,0x4a,  
0x6b,0x68,0x6d,0x6e,0x67,0x64,0x61,0x62,0x73,0x70,0x75,0x76,0x7f,0x7c,0x79,0x7a,  
0x3b,0x38,0x3d,0x3e,0x37,0x34,0x31,0x32,0x23,0x20,0x25,0x26,0x2f,0x2c,0x29,0x2a,  
0x0b,0x08,0x0d,0x0e,0x07,0x04,0x01,0x02,0x13,0x10,0x15,0x16,0x1f,0x1c,0x19,0x1a  
};
```

```
unsigned char rcon[256] = {
```

```
0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d,  
0x9a,  
0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39,  
0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d,  
0x3a,  
0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c,  
0xd8,  
0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef,  
0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66,  
0xcc,
```

```

0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80,
0x1b,

0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97, 0x35, 0x6a, 0xd4,
0xb3,

0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94,

0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02, 0x04, 0x08, 0x10,
0x20,

0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc, 0x63, 0xc6, 0x97,
0x35,

0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3, 0xbd, 0x61, 0xc2, 0x9f,

0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb, 0x8d, 0x01, 0x02,
0x04,

0x08, 0x10, 0x20, 0x40, 0x80, 0x1b, 0x36, 0x6c, 0xd8, 0xab, 0x4d, 0x9a, 0x2f, 0x5e, 0xbc,
0x63,

0xc6, 0x97, 0x35, 0x6a, 0xd4, 0xb3, 0x7d, 0xfa, 0xef, 0xc5, 0x91, 0x39, 0x72, 0xe4, 0xd3,
0xbd,

0x61, 0xc2, 0x9f, 0x25, 0x4a, 0x94, 0x33, 0x66, 0xcc, 0x83, 0x1d, 0x3a, 0x74, 0xe8, 0xcb,
0x8d});

```

```

void keyExpansionCore(unsigned char* in, unsigned char i)

```

```

{
    //Rotate left

    unsigned int* q = (unsigned int*)in;

    *q = (*q >> 8) | ((*q & 0xff) << 24);

    //s-Box four bytes:

    in[0] = s_box[in[0]]; in[1] = s_box[in[1]];

    in[2] = s_box[in[2]]; in[3] = s_box[in[3]];

```

```

        // Rcon

        in[0] ^= rcon[i];
    }

```

```

void KeyExpansion(unsigned char* inputKey,unsigned char* expandedKeys)
{
    //The first 16 bytes are the original key

    for(int i = 0; i < 16; i++)
        expandedKeys[i] = inputKey[i];

    int bytesGenerated = 16;

    int rconIteration = 1;

    unsigned char temp[4];

    while(bytesGenerated < 176)
    {
        //Read 4 bytes of the core

        for(int i = 0; i < 4; i++)

            temp[i] = expandedKeys[i + bytesGenerated - 4];

        if(bytesGenerated % 16 == 0)

```

```

    {
        keyExpansionCore(temp,rconIteration++);

        rconIteration++;
    }

    for(unsigned char a = 0; a < 4; a++)
    {
        expandedKeys[bytesGenerated] = expandedKeys[bytesGenerated - 16] ^ temp[a];
        bytesGenerated++;
    }

}

```

```

void SubBytes(unsigned char* state)
{
    for(int i =0; i < 16; i++)
        state[i] = s_box[state[i]];
}

```

```
void ShiftRows(unsigned char* state)
```

```
{
```

```
    unsigned char tmp[16];
```

```
    tmp[0] = state[0];
```

```
    tmp[1] = state[5];
```

```
    tmp[2] = state[10];
```

```
    tmp[3] = state[15];
```

```
    tmp[4] = state[4];
```

```
    tmp[5] = state[9];
```

```
    tmp[6] = state[14];
```

```
    tmp[7] = state[3];
```

```
    tmp[8] = state[8];
```

```
    tmp[9] = state[13];
```

```
    tmp[10] = state[2];
```

```
    tmp[11] = state[7];
```

```
    tmp[12] = state[12];
```

```
    tmp[13] = state[1];
```

```
    tmp[14] = state[6];
```

```
    tmp[15] = state[11];
```



```

        for(int i = 0; i < 16; i++)

            state[i] = tmp[i];

    }

void MixColumns(unsigned char* state)
{
    unsigned char tmp[16];

    tmp[0] = (unsigned char)(mul2[state[0]] ^ mul3[state[1]] ^ state[2] ^ state[3]);
    tmp[1] = (unsigned char)(state[0] ^ mul2[state[1]] ^ mul3[state[2]] ^ state[3]);
    tmp[2] = (unsigned char)(state[0] ^ state[1] ^ mul2[state[2]] ^ mul3[state[3]]);
    tmp[3] = (unsigned char)(mul3[state[0]] ^ state[1] ^ state[2] ^ mul2[state[3]]);

    tmp[4] = (unsigned char)(mul2[state[4]] ^ mul3[state[5]] ^ state[6] ^ state[7]);
    tmp[5] = (unsigned char)(state[4] ^ mul2[state[5]] ^ mul3[state[6]] ^ state[7]);
    tmp[6] = (unsigned char)(state[4] ^ state[5] ^ mul2[state[6]] ^ mul3[state[7]]);
    tmp[7] = (unsigned char)(mul3[state[4]] ^ state[5] ^ state[6] ^ mul2[state[7]]);

    tmp[8] = (unsigned char)(mul2[state[8]] ^ mul3[state[9]] ^ state[10] ^ state[11]);
    tmp[9] = (unsigned char)(state[8] ^ mul2[state[9]] ^ mul3[state[10]] ^ state[11]);
    tmp[10] = (unsigned char)(state[8] ^ state[9] ^ mul2[state[10]] ^ mul3[state[11]]);
    tmp[11] = (unsigned char)(mul3[state[8]] ^ state[9] ^ state[10] ^ mul2[state[11]]);

    tmp[12] = (unsigned char)(mul2[state[12]] ^ mul3[state[13]] ^ state[14] ^ state[15]);
    tmp[13] = (unsigned char)(state[12] ^ mul2[state[13]] ^ mul3[state[14]] ^ state[15]);

```

```
tmp[14] = (unsigned char)(state[12] ^ state[13] ^ mul2[state[14]] ^ mul3[state[15]]);  
tmp[15] = (unsigned char)(mul3[state[12]] ^ state[13] ^ state[14] ^ mul2[state[15]]);
```

```
for(int i = 0; i < 16; i++)
```

```
state[i] = tmp[i];
```

```
}
```

```
void AddRoundKey(unsigned char* state, unsigned char* roundKey)
```

```
{
```

```
    for(int i = 0; i < 16; i++)
```

```
        state[i] ^= roundKey[i];
```

```
}
```

```
void AES_Encrypt(unsigned char* message, unsigned char* key)
```

```
{
```

```
    unsigned char state[16];
```

```
    for(int i = 0; i < 16; i++)
```

```
        state[i] = message[i];
```

```
    int numberOfRounds = 9;
```

```
    unsigned char expandedKey[176];
```

```
    KeyExpansion(key, expandedKey);
```

```
    AddRoundKey(state, key); //Whitening/ AddRoundKey
```

```

        for(int i = 0; i < numberOfRounds;i++)
        {
            SubBytes(state);

            ShiftRows(state);

            MixColumns(state);

            AddRoundKey(state,expandedKey + (16 * (i + 1)));

        }

        //Final Round

        SubBytes(state);

        ShiftRows(state);

        AddRoundKey(state,expandedKey + 160);

        //copy over the message with the encrypted message!

        for(int i = 0; i < 16; i++)

            message[i] = state[i];

    }

    void PrintHex(unsigned char x)

    {

        if(x / 16 < 10) cout<<(char)((x / 16) + '0');

        if(x / 16 >= 10) cout<<(char)((x / 16 - 10) + 'A');

        if(x % 16 < 10) cout<<(char)((x % 16) + '0');

        if(x % 16 >= 10) cout<<(char)((x % 16 - 10) + 'A');

```

```
}
```

```
int main()
```

```
{
```

```
    unsigned char message[] = "Hello world";
```

```
    unsigned char key[16] =
```

```
    {
```

```
    1,2,3,4,
```

```
    5,6,7,8,
```

```
    9,10,11,12,
```

```
    13,14,15,16,
```

```
    };
```

```
    int originalLen = strlen((const char*)message);
```

```
    int lenOfPaddedMessage = originalLen;
```

```
    if(lenOfPaddedMessage % 16 != 0)
```

```
        lenOfPaddedMessage = (lenOfPaddedMessage / 16 + 1) * 16;
```

```
    unsigned char* paddedMessage = new unsigned char[lenOfPaddedMessage];
```

```
    for(int i = 0; i < lenOfPaddedMessage; i++)
```

```
    {
```

```

        if(i >= originalLen) paddedMessage[i] = 0;

        else paddedMessage[i] = message[i];

    }

    //Encrypt padded message

    for(int i = 0; i < lenOfPaddedMessage; i+= 16)

    AES_Encrypt(paddedMessage+i,key);

    cout<<"\nEncrypted message:"<<endl;

    for(int i = 0; i < lenOfPaddedMessage; i++)

    {

    PrintHex(paddedMessage[i]);

    cout<<" ";

    }

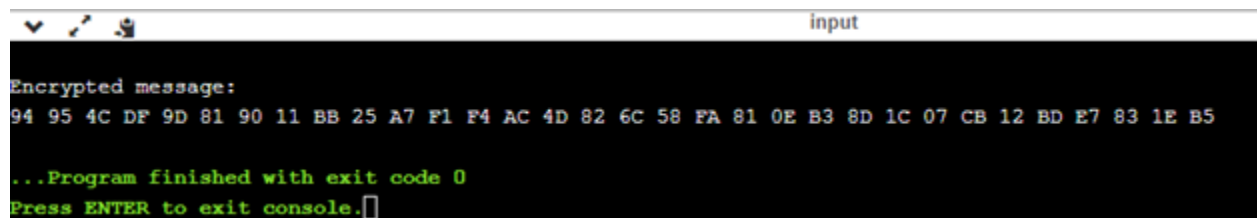
    delete[] paddedMessage;

    return 0;

}

```

OUTPUT-

A screenshot of a terminal window with a black background and white text. The window has a title bar with the word "input" on the right and three icons (a checkmark, a magnifying glass, and a trash can) on the left. The text in the terminal reads: "Encrypted message:" followed by a line of hexadecimal values: "94 95 4C DF 9D 81 90 11 BB 25 A7 F1 F4 AC 4D 82 6C 58 FA 81 0E B3 8D 1C 07 CB 12 BD E7 83 1E B5". Below this, it says "...Program finished with exit code 0" and "Press ENTER to exit console." with a cursor at the end.

```
input
Encrypted message:
94 95 4C DF 9D 81 90 11 BB 25 A7 F1 F4 AC 4D 82 6C 58 FA 81 0E B3 8D 1C 07 CB 12 BD E7 83 1E B5
...Program finished with exit code 0
Press ENTER to exit console.
```

Side-channel-attack-Power-Analysis

Aim

This document aims to explain a side channel attack using differential power analysis and then presents an implementation.

Side channel attack

How it works

In modern cryptography, the deciphering process is binary. You either have the correct decryption key, or you don't. There's no friendly message to tell you, 'Hey you are on the right track! Keep going!'.

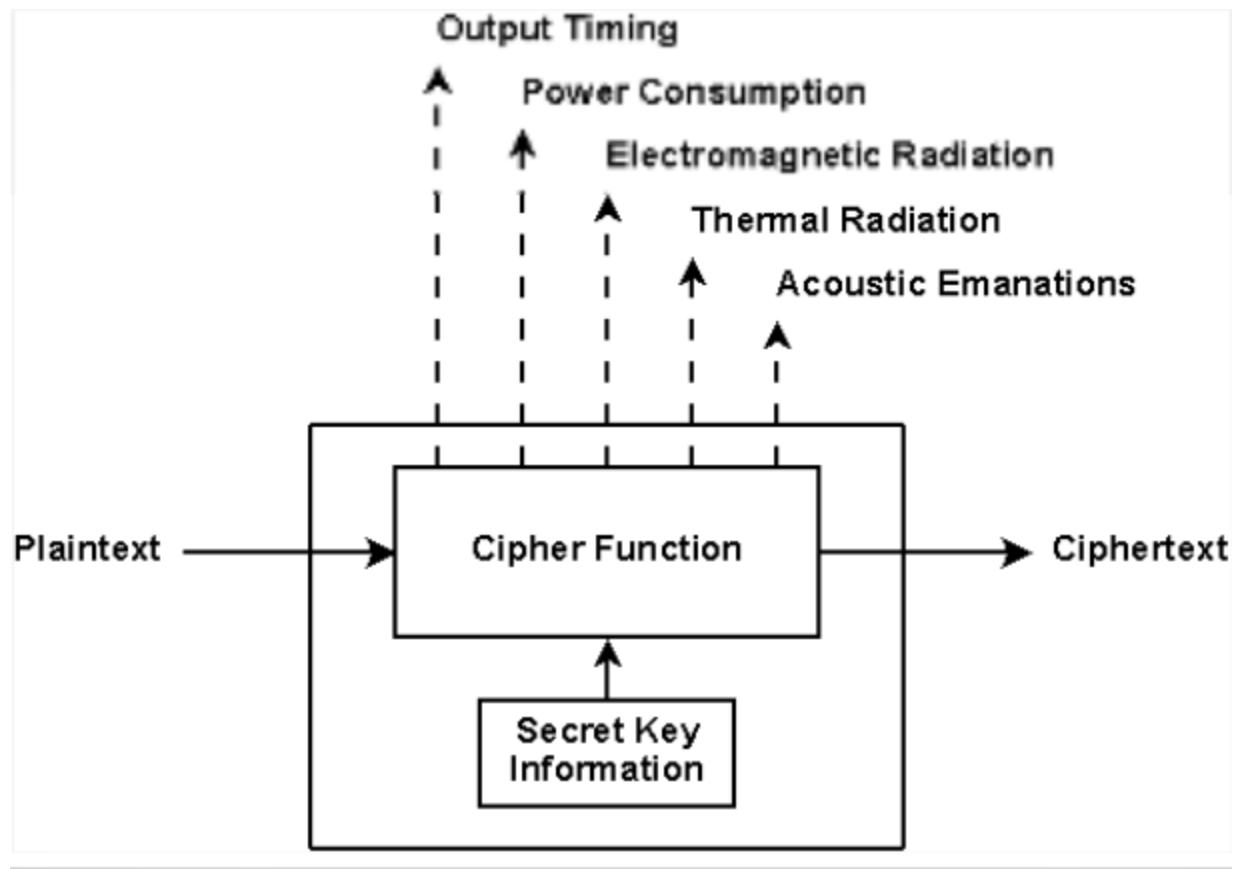
Side channel attacks look for **physical behaviors** of the cryptographic process. There are a lot of them (and more yet to be discovered):

- Magnetic field analysis
- Sound leaks
- Timing attack
- Row hammer
- Differential power analysis

The last one being explained in this document.

Why do those attacks exist ?

Computer security is a deep layered domain. Predicting and modeling those attacks is very difficult. Each layer of security impacts the assumptions made by the others. The software developer assumes that the hardware designer did his job well. As a result, security faults often involve unanticipated interactions between components. Components who are made by different people.

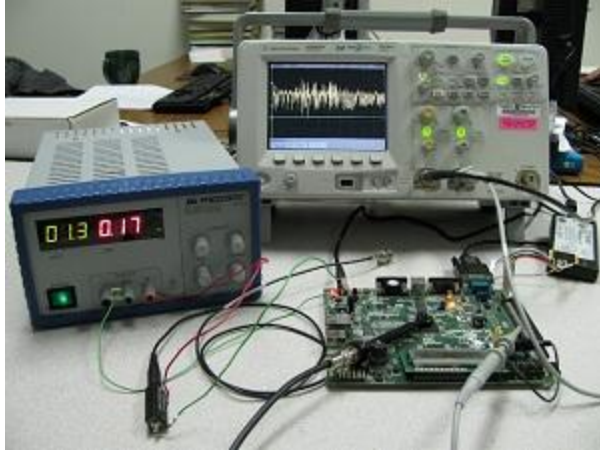


Within the existing model, side channel attacks can't be avoided. The model is vulnerable by design.

Differential Power Analysis

On the hardware level, cryptographic algorithms are implemented using semiconductor and logic gates (made of transistors). Those components have a power consumption, which can be measured. First of all, this attack can't be easily implemented at home.

You need an oscilloscope and a physical access to the processor/chip tested.



In addition of the SPA (simple power analysis) we add statistical functions to deduce sensitive information from power consumption.

AES Encryption

As Wikipedia says:

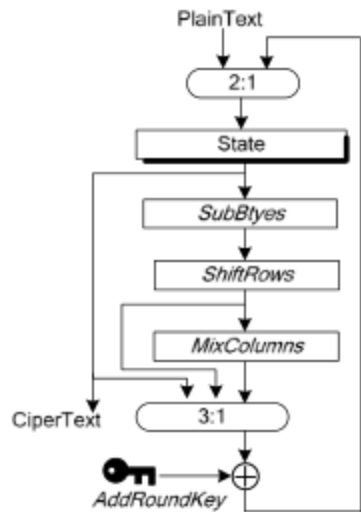
'AES (acronym of Advanced Encryption Standard) is a symmetric encryption algorithm. The algorithm was developed by two Belgian cryptographer Joan Daemen and Vincent Rijmen. AES was designed to be efficient in both hardware and software, and supports a block length of 128 bits and key lengths of 128, 192, and 256 bits.'

AES is a standard and widely used. Characteristics that make it very interesting for hackers.

AES Weakness

In AES, the plain text block is first XORed with the primary key and then goes through 10 rounds of processing. Each round consists of :

- SubByte
- ShiftRow
- MixColumn
- AddRoundKey



Implementation

To be successful a DPA attack needs to follow a few steps:

- Generate a huge amount of encryption/decryption operations using the target cryptosystem and key.
- Trace, for each operation, both power consumption and I/O (clear text and/or cipher text)
- Take a byte from the SBOX output and use it as a separator.
- Make the average out of each sub group.
- Repeat operation for each possible value of a byte (255).
- Take the highest average and enjoy finding a byte of the key.
- Repeat for each byte of the key.

Conclusion

Through this exercise, we were able to implement a successful DPA attack on AES-128. Even if the attack has a lot of requirements and needs a lot of

work to be effective, we clearly see the possibilities and threats of side channel attacks.

In the actual state of art, nothing cheap and effective can be found to avoid those attacks. The consumption randomization and reduction is clearly too expensive for entities others than governments or big companies.

The progressive centralisation of computing process in the 'cloud' may help governments and agencies perform side channels attack. Indeed, a datacenter is a warehouse with physical access to a lot of computing hardware. And the standardisation of side channel attacks on datacenter may already be a reality.

Code output:

Fundamentally, DPA can be thought of as a test that gauges the correlation between the bits that are being processed and the power the device consumes or the EM signals it emits. These power or EM measurements are known as traces, with trace measurements typically made across the entire operation. In the images attached for mat lab output , we can see raw traces, with the differential traces. The secret key is revealed in the standout spikes of the differential traces.

Code:

%%%%%%%%
 %%%%

```
% Matlab key recovery exercise template %
```

%

% 2014, Florent Bruquier, PCM - CNFM %

%%%%%%%%
 %%

% declaration of the SBOX (might be useful to calculate the power hypothesis)

SBOX=[099 124 119 123 242 107 111 197 048 001 103 043 254 215 171
118 ...

202 130 201 125 250 089 071 240 173 212 162 175 156 164 114 192

...

183 253 147 038 054 063 247 204 052 165 229 241 113 216 049 021

...

004 199 035 195 024 150 005 154 007 018 128 226 235 039 178 117

...

009 131 044 026 027 110 090 160 082 059 214 179 041 227 047 132

...

083 209 000 237 032 252 177 091 106 203 190 057 074 076 088 207

...

208 239 170 251 067 077 051 133 069 249 002 127 080 060 159 168

...

081 163 064 143 146 157 056 245 188 182 218 033 016 255 243 210

...

205 012 019 236 095 151 068 023 196 167 126 061 100 093 025 115

...

096 129 079 220 034 042 144 136 070 238 184 020 222 094 011 219

...

224 050 058 010 073 006 036 092 194 211 172 098 145 149 228 121

...

```

231 200 055 109 141 213 078 169 108 086 244 234 101 122 174 008
...
186 120 037 046 028 166 180 198 232 221 116 031 075 189 139 138
...
112 062 181 102 072 003 246 014 097 053 087 185 134 193 029 158
...
225 248 152 017 105 217 142 148 155 030 135 233 206 085 040 223
...
140 161 137 013 191 230 066 104 065 153 045 015 176 084 187 022];

```

```

%%%%%%%%%%

```

```

% LOADING the DATA %

```

```

%%%%%%%%%%

```

```

% modify following variables so they correspond

```

```

% your measurement setup

```

```

numberOfTraces = 200;

```

```

traceSize = 370000;

```

```

% modify the following variables to speed-up the measurement

```

```

% (this can be done later after analysing the power trace)

```

```

offset = 0;

segmentLength = 370000; % for the beginning the segmentLength =
traceSize

% columns and rows variables are used as inputs

% to the function loading the plaintext/ciphertext

columns = 16;

rows = numberOfTraces;

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% Calling the functions %

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% function myload processes the binary file containing the measured
traces and

% stores the data in the output matrix so the traces (or their reduced parts)

% can be used for the key recovery process.

% Inputs:

% 'file' - name of the file containing the measured traces

% traceSize - number of samples in each trace

% offset - used to define different beginning of the power trace

```

% segmentLength - used to define different/reduced length of the power trace

% numberOfTraces - number of traces to be loaded

%

% To reduce the size of the trace (e.g., to speed-up the computation process)

% modify the offset and segmentLength inputs so the loaded parts of the

% traces correspond to the trace segment you are using for the recovery.

```
traces = myload('traces-00112233445566778899aabbccddeeff.bin',  
traceSize, offset, segmentLength, numberOfTraces);
```

% function myin is used to load the plaintext and ciphertext

% to the corresponding matrices.

% Inputs:

% 'file' - name of the file containing the plaintext or ciphertext

% columns - number of columns (e.g., size of the AES data block)

% rows - number of rows (e.g., number of measurements)

```
plaintext = myin('plaintext-00112233445566778899aabbccddeeff.txt',  
columns, rows);
```

```
ciphertext = myin('ciphertext-00112233445566778899aabbccddeeff.txt',  
columns, rows);
```

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

```
% EXERCISE 1 -- Plotting the power trace(s): %
```

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

```
% Plot one trace (or plot the mean value of traces) and check that it is  
complete
```

```
% and then select the appropriate part of the traces (e.g., containing the  
first round).
```

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

```
figure(1);
```

```
plot(traces(1,:));
```

```
title('Consommation total d'un chiffrement');
```

```
% --> With zoom on first round <--
```

```
offset = 50000;
```

```
segmentLength = 40000;
```

```
traces2    =    myload('traces-00112233445566778899aabbccddeeff.bin',  
traceSize, offset, segmentLength, numberOfTraces);
```

```
figure(2);
```

```
plot(traces2(2,:));
```

```
title('Consommation d'un round AES');
```


%%

% EXERCISE 2 -- Key recovery: %

%%

% Create the power hypothesis for each byte of the key and then correlate
% the hypothesis with the power traces to extract the key.

% Task consists of the following parts:

% - calculate the intermediate values

% - create the power hypothesis

% - extract the key

% variables declaration

byteStart = 1;

byteEnd = 16;

keyCandidateStart = 0;

keyCandidateStop = 255;

solvedKey = zeros(1,byteEnd);

% for every byte in the key do:

for BYTE=byteStart:byteEnd

```

% Create the hypothesis matrix (dimensions:
% rows = numberOfTraces, columns = 256).
% The number 256 represents all possible bytes (e.g., 0x00..0xFF)
groupFin(1,:) = zeros(1,segmentLength);
Hypothesis = zeros(numberOfTraces,256);
cmptPlainText=0;

for K = keyCandidateStart:keyCandidateStop
    % --> calculate hypothesis here <--
    % Two AES first steps
    Hypothesis(:,K+1)=bitxor(plaintext(:,BYTE),K);
    Hypothesis(:,K+1)=SBOX(Hypothesis(:,K+1)+1);

    group1 = zeros(1,segmentLength);
    group2 = zeros(1,segmentLength);

    % --> do the differential analysis here <--
    % Séparation des traces en fonction d'une sous-clée
    nbTracesG1 = 0;

```

```
nbTracesG2 = 0;
```

```
% On parcourt l'ensemble des 200 traces
```

```
for L = 1:numberOfTraces
```

```
    %Récupération du premier bit du résultat
```

```
    %des deux premières étapes de l'AES
```

```
    firstByte = bitget(Hypothesis(L,K+1),1);
```

```
    if firstByte == 1
```

```
        %Incrémement des traces dans les groupes
```

```
        group1(1,:) = group1(1,:) + traces2(L,:);
```

```
        nbTracesG1 = nbTracesG1 + 1;
```

```
    else
```

```
        group2(1,:) = group2(1,:) + traces2(L,:);
```

```
        nbTracesG2 = nbTracesG2 + 1;
```

```
    end
```

```
end
```

```
%Calcul de la moyenne des groupes
```

```
group1(1,:) = group1(1,:) / nbTracesG1;
```

```
group2(1,:) = group2(1,:) / nbTracesG2;
```

```
%Création du groupe final composé de la soustraction
```

```

        %en valeur absolue des deux groupes précédents
        groupFin(K+1,:) = abs(group1(1,:)-group2(1,:));
    end

    %Récupération de la ligne qui comporte le point le plus haut
    %de la courbe de puissance
    %1°) -- On recherche tous les max de toutes les lignes
    %2°) -- On récupère l'index de ligne et de colonne des MAX
                                [ligne,colonne]=ind2sub(size(groupFin),
find(groupFin==max(groupFin(:))));

    solvedKey(1,BYTE) = ligne - 1;

    %Affichage temporaire de la courbe pour vérification
    %figure(3);
    %plot(groupFin(1,:));
    %title('Courbe DPA !');

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

end

```

```
function [traces] = myload(fname,trlen,start,len,n)
```

```
myfile=fopen(fname,'r');
```

```
traces=zeros(n,len);
```

```
for i=1:n
```

```
    fseek(myfile, start, 'cof');
```

```
    if (len+start > trlen)
```

```
        t=fread(myfile, len-start, 'uint8');
```

```
    else
```

```
        t=fread(myfile, len, 'uint8');
```

```
    end
```

```
    fseek(myfile, (trlen-len-start), 'cof');
```

```
    traces(i,:)=t;
```

```
end
```

```
fclose(myfile);
```

```
end
```

```
function [inputs]=myin(fname,ilen,n)

myfile=fopen(fname,'r');

inputs=zeros(n,ilen);

for i=1:n

    s = fgets(myfile, 1024);

    [ii, ~]=sscanf(s, '%02x ', ilen);

    inputs(i,:)=ii;

end

fclose(myfile);

End
```

Outputs:

```
a6d0dbcb2146f291ff8376eb4355e3a882cfccfc
```

