Side-channel attacks on Ascon's S-box

Alexane Boldo

OCIF, IRISA

May-July 2025

Supervisor: Hélène Le Bouder

Introduction

Side-Channel Attacks (SCA): observation of computation time, power consumption, electromagnetic radiation, ... to discover a secret

Goal: Study the leaks from the winner for lightweight cryptography Ascon to theorize a SCA attack

Table of Contents

What is Ascon-AEAD?

Authenticated Encryption with Associated Data (AEAD): encrypt, check authentication of content and associated data

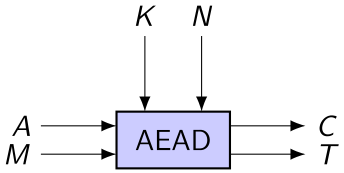
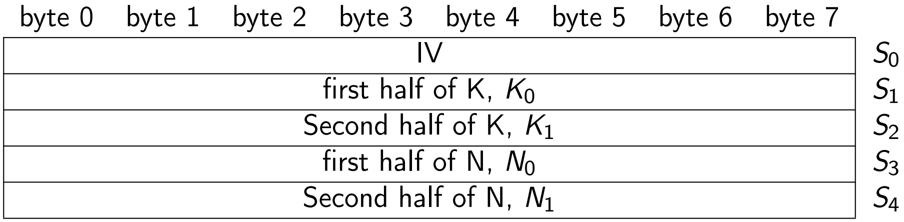


Figure: AEAD algorithm from [1]

Ascon's State



Encryption and decryption phases

4 phases: Initialization, Associated data process, plaintext/ciphertext process, Finalization

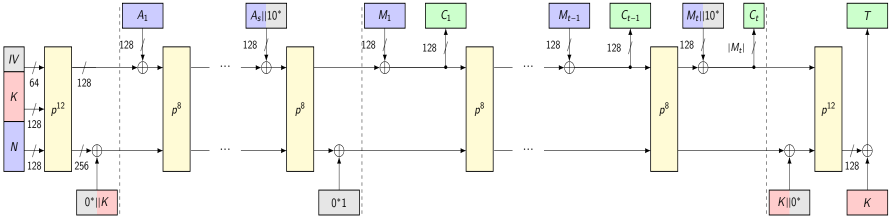


Figure: Ascon-AEAD mode, from [1]

Ascon's permutation

$$p = p_L \circ \underbrace{p_S} \circ p_C$$

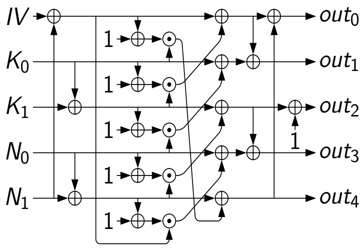


Figure: Circuit to compute the S-box, from [1], bijection $[\![1;31]\!] \to [\![1;31]\!]$

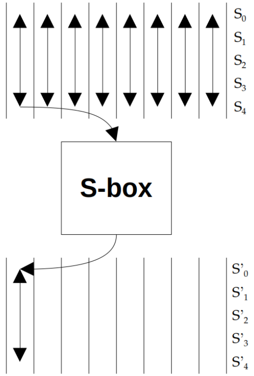


Figure: S-box computation for the first byte of each word

[1] https://extgit.isec.tugraz.at/meichlseder/tikz

Table linking the output of the S-box and the key

(n_0^j, n_1^j, IV^j)	S_4^j
$(0,0,0)$	k_0^j
$(0,0,1)$	0
$(0,1,0)$	1
$(0,1,1)$	$1 \oplus k_0^j$
$(1,0,0)$	$1 \oplus k_0^j$
$(1,0,1)$	1
$(1,1,0)$	0
$(1,1,1)$	k_0^j

Figure: Link between k_0^j and S_4^j depending on IV and N, from [3]

ChipWhisperer-Lite

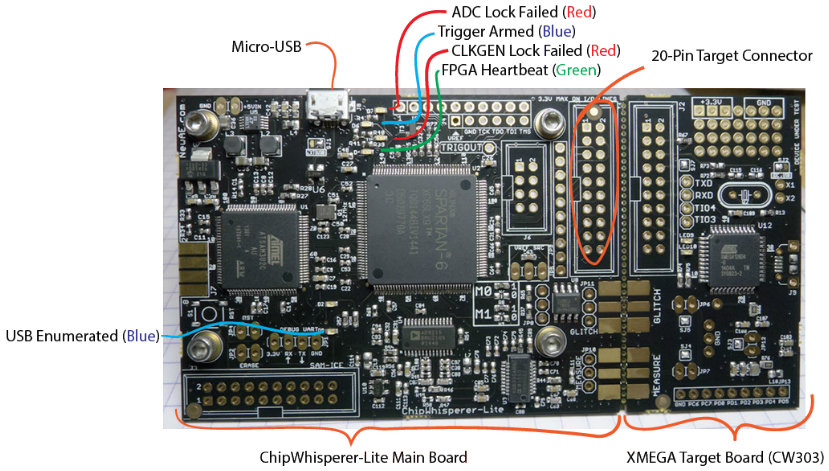


Figure: ChipWhisperer Lite board, from [4]

Steps for a CPA attack

- Compute the algorithm multiple times to gain traces
- Find model for the consumption
- Deduce the hypothesis that correlates best

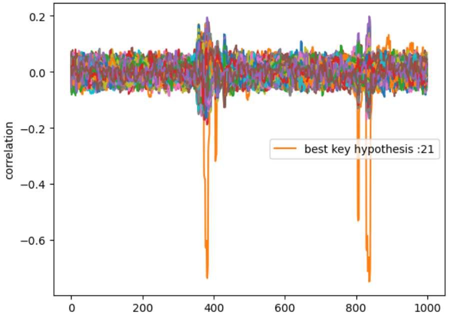


Figure: CPA on another encryption standard

Analyses done

- Finding the best model
  - Vertical vs horizontal
  - HW vs value
- Attack: finding the vertical output and deduct the key

Results vertical vs horizontal and HW vs value

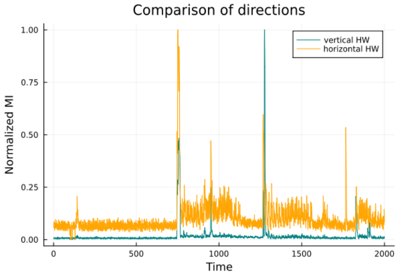


Figure: Mutual information for the horizontal and the vertical value

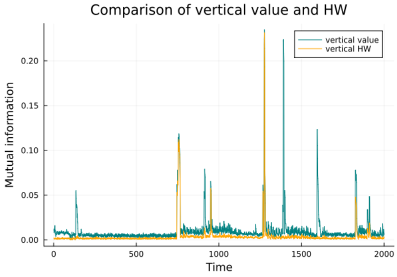


Figure: Mutual information between power consumption and HW or value

Results attack

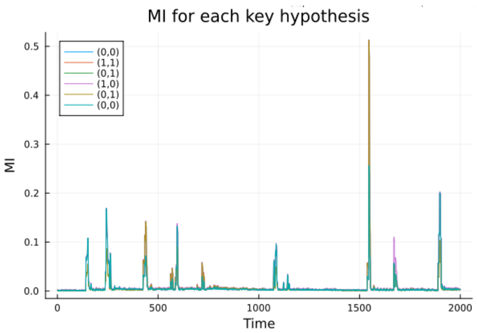


Figure: Mutual information between the Hamming weight of the outputs and the power consumption, for each of the possible outputs for the first nonce

Conclusion

- Good leaks compared to random values
- Though apparent weaknesses, unsuccessful attempts
- Not enough randomness with false key hypotheses
- Leads to follow: belief propagation

- L. B. H., T. G., and A. D., "Théorie de la cryptographie."
- S. T. M., M. K., C. D., K. J., and K. J., "Ascon-based lightweight cryptography standards for constrained devices: Authenticated encryption, hash, and extendable output functions," *NIST Special Publication 800, NIST SP 800-232 ipd*, 2024. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-232.ipd.pdf
- S. M., "Side channel analysis against aead." [Online]. Available: https://theses.hal.science/tel-04816066v1
- Chipwhisperer documentation. [Online]. Available: https://chipwhisperer.readthedocs.io/en/latest/getting-started.html

Permutation (1), p_C

Constant for the round i: $const_{16-nb_{rounds}+i}$

i	const$_i$	i	const$_i$
0	0x000000000000003c	8	0x00000000000000b4
1	0x000000000000002d	9	0x00000000000000a5
2	0x000000000000001e	10	0x0000000000000096
3	0x000000000000000f	11	0x0000000000000087
4	0x00000000000000f0	12	0x0000000000000078
5	0x00000000000000e1	13	0x0000000000000069
6	0x00000000000000d2	14	0x000000000000005a
7	0x00000000000000c1	15	0x000000000000004b

Figure: Constant-addition layer, constants

Permutation (2), p_C

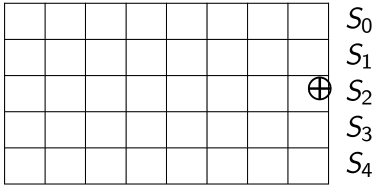


Figure: Constant-addition layer, each box representing a byte of one of the 64-bit words

Permutation (3), p_S

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
$S-box(x)$	4	b	1f	14	1a	15	9	2	1b	5	8	12	1d	3	6	1c	1e	13	7	e	0	d	11	18	10	c	1	19	16	a	f	17

Figure: Lookup table for the 5-bit S-box

Permutation (4), p_S

```
        state[0] ^= state[4];
        state[4] ^= state[3];
        state[2] ^= state[1];
        uint64_t t0 = ~state[0];
        uint64_t t1 = ~state[1];
        uint64_t t2 = ~state[2];
        uint64_t t3 = ~state[3];
        uint64_t t4 = ~state[4];
        t0 &= state[1];
        t1 &= state[2];
        t2 &= state[3];
        t3 &= state[4];
        t4 &= state[0];
        state[0] ^= t1
        ; state[1] ^= t2;
        state[2] ^= t3;
        state[3] ^= t4;
        state[4] ^= t0;
        state[1] ^= state[0];
        state[0] ^= state[4];
        state[3] ^= state[2];
        state[2] =~ state[2];

```

Figure: Equations to compute the S-box

Permutation (5), p_L

Diffusion: $S_i \leftarrow \Sigma_i(S_i)$:

$$
\begin{aligned}
\Sigma_0(S_0) &= S_0 \oplus (S_0 >>> 19) \oplus (S_0 >>> 28)\\
\Sigma_1(S_1) &= S_1 \oplus (S_1 >>> 61) \oplus (S_1 >>> 39)\\
\Sigma_2(S_2) &= S_2 \oplus (S_2 >>> 1) \oplus (S_2 >>> 6)\\
\Sigma_3(S_3) &= S_3 \oplus (S_3 >>> 10) \oplus (S_3 >>> 17)\\
\Sigma_4(S_4) &= S_4 \oplus (S_4 >>> 7) \oplus (S_4 >>> 41)
\end{aligned}
$$

Finding this table (1)

$$S_4^j = n_o^j \oplus n_1^j \oplus k_0^j \times (1 \oplus IV^j \oplus n_1^j)$$

$$S_4^j = \begin{cases} k_0^j \times (1 \oplus IV^j) & if \ \ (n_0^j, n_1^j) = (0,0) \\ k_0^j \times IV^j & if \ \ (n_0^j, n_1^j) = (1,1) \\ 1 \oplus k_0^j \times IV^j & if \ \ (n_0^j, n_1^j) = (0,1) \\ 1 \oplus k_0^j \times (1 \oplus IV^j) & if \ \ (n_0^j, n_1^j) = (1,0) \end{cases}$$

Finding this table (2)

Then if $IV^j = 0$:

$$S_4^j = \begin{cases} k_0^j & \textit{if } (n_0^j, n_1^j) = (0,0) \\ 0 & \textit{if } (n_0^j, n_1^j) = (1,1) \\ 1 & \textit{if } (n_0^j, n_1^j) = (0,1) \\ 1 \oplus k_0^j & \textit{if } (n_0^j, n_1^j) = (1,0) \end{cases}$$

Finding this table (3)

Otherwise, if $IV^j = 1$:

$$S_4^j = \begin{cases} 0 & if \ (n_0^j, n_1^j) = (0,0) \\ k_0^j & if \ (n_0^j, n_1^j) = (1,1) \\ 1 \oplus k_0^j & if \ (n_0^j, n_1^j) = (0,1) \\ 1 & if \ (n_0^j, n_1^j) = (1,0) \end{cases}$$

Complementary graph (1)

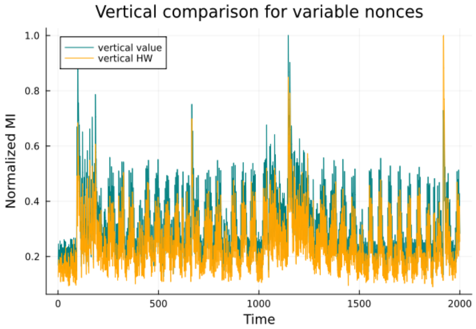


Figure: Mutual information between power consumption and Hamming weight of the concatenation of the first bit of each of the word of S and its value like 9 but for random nonces

Complementary graph (2)

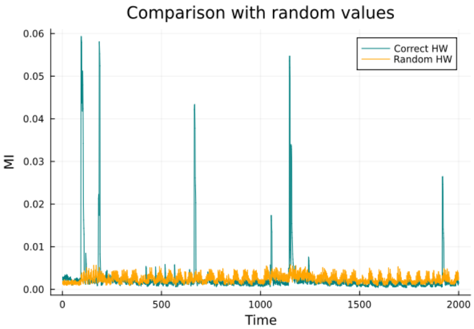


Figure: Mutual information between power consumption and vertical HW or random possible HW

```
#include "ascon.h"
#include "permutation.h"
#include <stdint.h>
#include <stdlib.h>
#include <stdbool.h>

//--------------------------------------------------------
// ASCON implementation following the NIST Special Publication 800, NIST SP
    800-232 ipd
//--------------------------------------------------------

//unsure of the generalization of this IV, as t=128 is not defined,
//only given for this specific algorithm
uint64_t IV = ((((((((uint64_t) RATIO<<16) + 128)<<4) + NB_RNDS_B)<<4) +
                                                        NB_RNDS_A)<<16) + VERSION;

//main functions
void ASCONEncrypt(uint8_t* C, uint8_t* T, uint8_t* P, int len_p,
                  uint8_t* A, int len_a, uint8_t* n, uint8_t* key){
    State_t* state = initialization(P, len_p, A, len_a, n, key, true);

    associated_data_process(state);

    plaintext_process(state, len_p%RATIO);

    finalization(state, key, T);

    slice(state->blocktext_out, C, len_p);

    free_state(state);
    return;
}


void ASCONDecrypt(uint8_t* C, uint8_t* P, int len_p, uint8_t* A, int len_a,
```

Main code

```c
                        uint8_t* n, uint8_t* key, uint8_t* T, bool* fail){
    State_t* state = initialization(C, len_p, A, len_a, n, key,false);

    associated_data_process(state);

    ciphertext_process(state, len_p%RATIO);

    uint8_t* Tprim = malloc(KEY_LENGTH/BYTE_LENGTH*sizeof(uint8_t));

    finalization(state, key, Tprim);

    if (same_tag(T,Tprim)){
        *fail = false;
        slice(state->blocktext_out, P, len_p);
    }
    else{
        *fail = true;
    }

    free(Tprim);
    free_state(state);
    return;
}

//initialzation functions
uint64_t get_IV(void){
    return IV;
}

uint8_t** combine(uint8_t* text, int len, bool padding){
    //new_len : has to be incremented either for the added padding
    //or because the value has been floored
    //if padding=false and len%RATIO=0, adding an empty block
    int new_len  = len/RATIO + 1;
    uint8_t** new_text = malloc(new_len*sizeof(uint8_t*));
    for(int i=0; i<new_len;i++){
        new_text[i] = calloc(RATIO,sizeof(uint8_t));
    }

    for(int i=0; i<len; i++){//little endian for each 64bit block
        new_text[i/RATIO][8*((i%RATIO)/8) + 7 +(-i)%8] = text[i];
    }
    //adding a 1 at the end of the block if a padding is needed
    if(padding)
        new_text[new_len-1][8*((len%RATIO)/8) + 7 +(-len)%8] += 0x01;
    return new_text;
}

State_t* state_initialization(uint8_t* textin, int len_p, uint8_t* A,
                              int len_a, uint8_t* n, uint8_t* key, bool plain){

    State_t* state = (State_t*)malloc(sizeof(State_t));

    state->words[0] = get_IV();

    //|K| + |n| + 64 = 320, so K U n has 32 bytes,
    //adds K then n to the four last words of state
    state->words[1] = 0;
    state->words[2] = 0;
    state->words[3] = 0;
    state->words[4] = 0;
    for(int i = 0; i<32; i++){
        uint64_t curr_byte = (i<KEY_LENGTH/BYTE_LENGTH)?
                              key[i] : n[i-KEY_LENGTH/BYTE_LENGTH];
        state->words[1 + i/8] += curr_byte<<((i*BYTE_LENGTH)%64);
    }

    state->Ai = combine(A, len_a, true);
    state->s = len_a/RATIO + 1;

    state->blocktext_in = combine(textin, len_p, plain);
    state->t = len_p/RATIO + 1;

    state->blocktext_out = malloc(state->t*sizeof(uint8_t*));
    for(int i=0; i<state->t; i++){
        state->blocktext_out[i] = calloc(RATIO,sizeof(uint8_t));
    }

    return state;
}

State_t* initialization(uint8_t* textin, int len_p, uint8_t* A, int len_a,
                        uint8_t* n, uint8_t* key, bool plain){
    State_t* state = state_initialization(textin,len_p,A,len_a,n,key,plain);

    permutation(state,NB_RNDS_A);

    for(int i=0; i<KEY_LENGTH/BYTE_LENGTH;i++){
        //finds the first modified word to add the key at the end of the state
        int ind = 5 - KEY_LENGTH/64 - (KEY_LENGTH%64 != 0) + i/8;
        state->words[ind] ^= (uint64_t) key[i]<<((i*BYTE_LENGTH)%64);
    }
    return state;
}

//processing associated data
void associated_data_process(State_t* state){
    for(int i=0; i<state->s; i++){
        for(int j=0; j<RATIO; j++){
            int shift = (RATE-((j+1)*BYTE_LENGTH))%64;
            state->words[j/8] ^= (uint64_t) state->Ai[i][j]<<shift;
        }
        permutation(state, NB_RNDS_B);
    }
    //update for the least significant byte in little endian
    state->words[4] ^= (uint64_t) 1<<63;
    return;
}

//processing plaintext/ciphertext
void plaintext_process(State_t* state, int l){
    for(int i=0; i<state->t-1; i++){
        for(int j=0; j<RATIO; j++){
            int shift = (RATE-((j+1)*BYTE_LENGTH))%64;
            state->words[j/8] ^= (uint64_t) state->blocktext_in[i][j]<<shift;
            //too much bits for a uint8_t, only takes the smallest byte
            state->blocktext_out[i][j] = state->words[j/8]>>shift;
        }
        permutation(state,NB_RNDS_B);
    }
    //C_l is going to have its RATIO-l least significant bytes uninitialized,
    //but it is unimportant because C will be computed only with its beginning
    for(int j=0; j<l+1; j++){
        int i = state->t - 1;
        //not every byte will be explored so the order matters
        int ind = 8*((j%RATIO)/8) + 7 +(-j)%8;
        int shift = (RATE-((ind+1)*BYTE_LENGTH))%64;
        state->words[ind/8] ^= (uint64_t) state->blocktext_in[i][ind]<<shift;
        state->blocktext_out[i][ind] = state->words[(ind-1)/8]>>shift;
    }
    return;
}

void ciphertext_process(State_t* state, int l){
    for(int i=0; i<state->t-1; i++){
        for(int j=0; j<RATIO; j++){
            uint8_t wbyte = state->words[j/8]>>((RATE-((j+1)*BYTE_LENGTH))%64);
            state->blocktext_out[i][j] = wbyte^state->blocktext_in[i][j];
            char* word = (char*) &(state->words[j/8]);
            word[(RATIO-1-j)%8] = state->blocktext_in[i][j];
        }
        permutation(state, NB_RNDS_B);
    }

    for(int j=0; j<l; j++){
        int i = state->t - 1;
        //not every byte will be explored so the order matters
        int ind = 8*((j%RATIO)/8) + 7 +(-j)%8;
        uint8_t wbyte = state->words[ind/8]>>((RATE-((ind+1)*BYTE_LENGTH))%64);
        state->blocktext_out[i][ind] = wbyte^state->blocktext_in[i][ind];
        char* word = (char*) &(state->words[j/8]);
        word[(RATIO-1-ind)%8] = state->blocktext_in[i][ind];
    }
    state->words[l/8] ^= (uint64_t) 1<<((BYTE_LENGTH*l)%64);
    return;
}

//finalization functions
void slice(uint8_t** blocktextout, uint8_t* textout, int len){
    for(int i=0; i<len; i++){
        textout[i] = (blocktextout[i/RATIO][8*((i%RATIO)/8) + 7 +(-i)%8]);
    }
    return;
}

void finalization(State_t* state, uint8_t* key, uint8_t* T){
    for(int i=0; i<KEY_LENGTH/BYTE_LENGTH; i++){
        //adding the key after the rate
        int ind = (RATIO-1)/8 + 1 + i/8;
        state->words[ind] ^= (uint64_t) key[i]<<((i*BYTE_LENGTH)%64);
    }

    permutation(state, NB_RNDS_A);

    for(int i=0; i<TAG_LENGTH/BYTE_LENGTH; i++){
```

```c
        //xors the last TAG_LENGTH bits of the key and the state
        int ind = 5 - TAG_LENGTH/64 - (TAG_LENGTH%64 != 0) + i/8;
        uint8_t key_byte = key[i + (KEY_LENGTH-TAG_LENGTH)/BYTE_LENGTH];
        T[i] = state->words[ind]>>((i*BYTE_LENGTH)%64) ^ key_byte;
    }
    return;
}

bool same_tag(uint8_t* T,uint8_t* Tprim){
    bool res = true;
    for(int i=0; i<TAG_LENGTH/BYTE_LENGTH; i++){
        //to have the same computation time no matter the value of T and T'
        res = res && (T[i] == Tprim[i]);
    }
    return res;
}

void free_state(State_t* state){
    for(int i=0; i<state->s; i++)
        free(state->Ai[i]);
    free(state->Ai);
    for(int i=0; i<state->t; i++){
        free(state->blocktext_in[i]);
        free(state->blocktext_out[i]);
    }
    free(state->blocktext_in);
    free(state->blocktext_out);
    free(state);
    return;
}
```

Listing: Implementation for ascon.c

```c

#ifndef ASCON_H
#define ASCON_H

#define KEY_LENGTH 128
#define TAG_LENGTH 128
#define NONCE_LENGTH (256 - KEY_LENGTH)
#define RATE 128
#define BYTE_LENGTH 8
#define NB_RNDS_A 12
#define NB_RNDS_B 8
#define VERSION 1
#define RATIO (RATE/BYTE_LENGTH)

#include <stdint.h>
#include <stdbool.h>

//defining a type State_t in order to implement the state with the data
typedef struct State_s {
    uint64_t words[5];
    uint8_t** Ai;//asociated data in blocks of size RATE
    int s; //number of blocks in Ai
    uint8_t** blocktext_in; //blocks of the plaintext when encryption,
                            // when decryption blocks of ciphertext
    int t; //number of blocks in the given text, whether plain or ciphered
    uint8_t** blocktext_out;// output blocks (resp. ciphered or plain)
} State_t;


//main functions
void ASCONEncrypt(uint8_t* C, uint8_t* T, uint8_t* P, int len_p,
                  uint8_t* A, int len_a, uint8_t* n, uint8_t* key);
//takes a pointer C pointing towards allocated memory of size len_p,
//and a pointer T for the tag for a space of TAG_LENGTH

```

```c
void ASCONDecrypt(uint8_t* C, uint8_t* P, int len_p, uint8_t* A, int len_a,
                  uint8_t* n, uint8_t* key, uint8_t* T, bool* fail);
//same thing but for P
//modifies fail thanks to the pointer if authentication failed


//initialization functions
uint64_t get_IV(void);//getter for the IV

State_t* state_initialization(uint8_t* textin, int len_p, uint8_t* A,
                              int len_a, uint8_t* n, uint8_t* key, bool plain);
//creates the state with the five words
//and the blocks for the plaintext, ciphertext and associated data

uint8_t** combine(uint8_t* text, int len, bool padding);
//transforms list of uint8_t in list of lists of RATE uint8_t,
//gives the responsability to free the list, adds a padding if padding=true

State_t* initialization(uint8_t* textin, int len_p, uint8_t* A, int len_a,
                        uint8_t* n, uint8_t* key, bool plain);
//creates the state and does the initialization phase, if plain=true encryption
//therefore there will be a padding, otherwise not
//gives responsability to free the state

//processing associated data
void associated_data_process(State_t* state);


//processing plaintext/ciphertext
void plaintext_process(State_t* state, int l);
//l, the length in bytes of the last block in the original text
void ciphertext_process(State_t* state, int l);


//finalization functions
```

```c
void slice(uint8_t** blocktextout, uint8_t* textout, int len);
//flattens the list of list blocktextout in a list textout
//returns text of length len

void finalization(State_t* state, uint8_t* key, uint8_t* T);

bool same_tag(uint8_t* T,uint8_t* Tprim); //checks T=T'

void free_state(State_t* state);

#endif
```

Listing: Implementation for ascon.h

```c

#include "permutation.h"

//————————————————————————————————————————————————————————————————
// permutation implementation for Ascon, following the NIST Special Publication
   800, NIST SP 800-232 ipd
//————————————————————————————————————————————————————————————————

void permutation(State_t* state, int nb_rounds, bool attack){ //modifies the
   current state by applying nb_rounds times the permutation
    for (int r=0; r<nb_rounds; r++){
        uint8_t c = round_const(nb_rounds,r);
        perm_const(state, c);

        perm_sub(state);

        if(attack && r==0){
            trigger_high();
            perm_lin(state);
            trigger_low();
        }
        else
          perm_lin(state);
    }
    return;
}

uint8_t round_const(int nb_rounds, int round){
    //finds the right index, the table of consts defining for 12 rounds c_0 = 0xf0
    int i = 12 - nb_rounds + round;

    //relationship between index and value defined for i in (-4,11)
    return 0xf0 - i*0x10 + (0x10+i)%0x10;
}

void perm_const(State_t* state, uint8_t c){
    state->words[2] ^= (uint64_t)c;
    return;
}

void perm_sub(State_t* state){
    //applies the bitsliced implementation from figure 4
    state->words[0] ^= state->words[4]; state->words[4] ^= state->words[3] ; state->words[2] ^= state->words[1];
    uint64_t t0 = state->words[0]; uint64_t t1 = state->words[1] ; uint64_t t2 = state->words[2];
    uint64_t t3 = state->words[3] ; uint64_t t4 = state->words[4];
    t0 =~ t0; t1 =~ t1 ; t2 =~ t2 ; t3 =~ t3 ; t4 =~ t4;
    t0 &= state->words[1] ; t1 &= state->words[2] ; t2 &= state->words[3] ; t3 &= state->words[4] ; t4 &= state->words[0];
    state->words[0] ^= t1 ; state->words[1] ^= t2 ; state->words[2] ^= t3 ; state->words[3] ^= t4 ; state->words[4] ^= t0;
    state->words[1] ^= state->words[0] ; state->words[0] ^= state->words[4];
    state->words[3] ^= state->words[2] ; state->words[2] =~ state->words[2];
    return;
}

uint64_t circular_shift(uint64_t word, int nb){
    uint64_t shifted = word>>nb;
    uint64_t extracted = word<<(RATE-nb);
    return shifted+extracted;
}

void perm_lin(State_t* state){
    //calculates the sigma functions
    state->words[0] = state->words[0]^circular_shift(state->words[0],19)^
     circular_shift(state->words[0],28);
    state->words[1] = state->words[1]^circular_shift(state->words[1],61)^
     circular_shift(state->words[1],39);
    state->words[2] = state->words[2]^circular_shift(state->words[2],1)^
     circular_shift(state->words[2],6);
    state->words[3] = state->words[3]^circular_shift(state->words[3],10)^
     circular_shift(state->words[3],17);
    state->words[4] = state->words[4]^circular_shift(state->words[4],7)^
     circular_shift(state->words[4],41);
    return;
}
```

Listing: Implementation for permutation.c

```c
#include "ascon.h"

#ifndef PERMUTATION_H
#define PERMUTATION_H

//functions for the permutation
```

```c
void permutation(State_t* state, int nb_rounds, bool attack); //modifies the
    current state by applying nb_rounds times the permutation
uint8_t round_const(int nb_rounds, int round); //calculates the constant as
    shown in table 2 of the permutation section
void perm_const(State_t* state, uint8_t c);
void perm_sub(State_t* state);
uint64_t circular_shift(uint64_t word, int nb);//shifts the word nb spaces right
void perm_lin(State_t* state);

#endif
```

Listing: Implementation for permutation.h

```c
#include "ascon.h"
#include "hal/hal.h"
#include "hal/simpleserial.h"
#include <stdint.h>
#include <stdlib.h>

uint8_t key[KEY_LENGTH/BYTE_LENGTH];
uint8_t A[MAX_DATA_SIZE];
uint8_t P[MAX_DATA_SIZE];
uint8_t C[2*MAX_DATA_SIZE];
uint8_t* T = &(C[MAX_DATA_SIZE]);
uint8_t n[NONCE_LENGTH];

uint8_t set_key(uint8_t* k, uint8_t len)
{
    if (len != KEY_LENGTH/BYTE_LENGTH) {
        led_error(1);
        return 0x01;
    }

```

```c
  for(int i = 0; i < KEY_LENGTH/BYTE_LENGTH; i++) {
        key[i] = k[i];
    }
  return 0x00;
}

uint8_t set_assodata(uint8_t* ad, uint8_t len){
    if (len != MAX_DATA_SIZE) {
        led_error(1);
        return 0x01;
    }

  for(int i = 0; i < MAX_DATA_SIZE; i++) {
        A[i] = ad[i];
    }
  return 0x00;
}

uint8_t set_nonce(uint8_t* nonce, uint8_t len){
    if (len != NONCE_LENGTH/BYTE_LENGTH){
        led_error(1);
        return 0x01;
    }

    for(int i=0; i < NONCE_LENGTH/BYTE_LENGTH; i++){
        n[i] = nonce[i];
    }
    return 0x00;
}

uint8_t set_pt(uint8_t* pt, uint8_t len)
{
    if (len != MAX_DATA_SIZE) {
        led_error(1);
        return 0x01;
    }

    for(int i = 0; i < MAX_DATA_SIZE; i++) {
        P[i] = pt[i];
    }

    /*
    Encrypt here using ciphertext, plaintext, tag, associated data and key
     variables.
    */
  ASCONEncrypt(C,T,P,len,A,len,n,key);

  simpleserial_put('r', 16, C);//sends ciphertext
  return 0x00;
}

int main(void)
{
    platform_init();
    init_uart();
    trigger_setup();

    led_ok(1);
    led_error(0);

    simpleserial_init();
    simpleserial_addcmd('k', 16, set_key);
    simpleserial_addcmd('a', 16, set_assodata);
    simpleserial_addcmd('n', 16,  set_nonce);
    simpleserial_addcmd('p', 16,  set_pt);


    while(1) {
        simpleserial_get();// get next command and react to it
    }
```

```
}
```

Listing: Implementation for main.c

```python
import chipwhisperer as cw
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm
import time

def reset_target(scope):
    scope.io.pdic = 'low'
    time.sleep(0.05)
    scope.io.pdic = 'high'
    time.sleep(0.05)

def generate_nonce() :
    nonce = np.random.randint(0,high=1<<64-1,size=2)
    nonce[1] = 0
    for i in range(64):
        nonce[1] += ((IV[i//8]>>(i%8))%2)<<i

    nonce = bytearray(nonce)

    return nonce

IV = [0x00,0x00,0x10,0x00,0x80,0x8c,0x00,0x01]

scope = cw.scope()

scope.default_setup()
scope.adc.samples = 2000

```

```python
target = cw.target(scope)

ktp = cw.ktp.Basic()
key, pt = ktp.new_pair()
tag = ktp.next_text()
ad = ktp.next_text()
nonce = generate_nonce()

target.simpleserial_write('k',key)
target.simpleserial_wait_ack()
target.simpleserial_write('a',ad)
target.simpleserial_wait_ack()
target.simpleserial_write('n',nonce)
target.simpleserial_wait_ack()
target.simpleserial_write('t',tag)
target.simpleserial_wait_ack()

nonces = []
traces = []
pts = []
cts = []
tags = []

for i in tqdm(range(10000)):

    ct = ktp.next_text()
    tag = ktp.next_text()
    ad = ktp.next_text()

    if True :
        nonce = generate_nonce()

    target.simpleserial_write('a',ad)
    target.simpleserial_wait_ack()
    target.simpleserial_write('t',tag)
    target.simpleserial_wait_ack()
    target.simpleserial_write('n',nonce)
    target.simpleserial_wait_ack()

    trace = cw.capture_trace(scope,target,pt, key)
    traces.append(trace.wave)
    cts.append(trace.textin)
    pts.append(trace.textout)
    tags.append(tag)
    nonces.append(nonce)

nptraces = np.asarray(traces)
np.save("Mesures_hf/traces.npy", nptraces)

nppts = np.asarray(pts)
np.save("Mesures_hf/pts.npy", nppts)

npcts = np.asarray(cts)
np.save("Mesures_hf/cts.npy", npcts)

npnonces = np.asarray([nonces])
np.save("Mesures_hf/nonces.npy", npnonces)

nptags = np.asarray(tags)
np.save("Mesures_hf/tags.npy", nptags)

plt.plot(traces[0])
plt.show()
```

Listing: Implementation for trace capture for the ChipWhisperer

```
"""Using the equations from Modou Sarry during the decryption with a fixed nonce
    to find the key by finding the constant output of the initialization
```

```
sbox"""

using NPZ
using Statistics
using InformationMeasures
using Plots

plain = npzread("Mesures_hf/pts.npy")
cipher = npzread("Mesures_hf/cts.npy")
tags = npzread("Mesures_hf/tags.npy")
traces = npzread("Mesures_hf/traces.npy")
nonces = npzread("Mesures_hf/nonces.npy")[1,:,:]

function cut(x,len)
    [(x<<(k-1))>>(len-1) for k in 1:len]
end

function assemble(lst,len)
    sum(map(<<,lst,(len-1):-1:0))
end

function to_bits(x,deb,len)
    [(UInt8(x[deb + i])<<(k-1))>>7 for i in 1:len for k in 1:8]
end

function to_bytes(lst,len)
    [sum(map(<<,lst[j:j+7],7:-1:0)) for j in 1:8:len]
end

function sbox_4(k0,k1,n0,n1,v0)
    xor(n0,xor(n1,k0*(xor(1, xor(v0, n1)))))
end

function sbox_3(k0,k1,n0,n1,v0)
    xor(xor(v0,1)*xor(n1, n0), xor(v0, xor(k0,k1)))
```

```
end

function sbox_2(k0,k1,n0,n1,v0)
    xor(n1*xor(n0,1), xor(1, xor(k0,k1)))
end

function sbox_1(k0,k1,n0,n1,v0)
    xor(n1, xor(v0, xor(xor(n0,1)*xor(k0,k1), k0*k1)))
end

function sbox_0(k0,k1,n0,n1,v0)
    xor(n0, xor(v0, xor(k1,k0*(xor(n1, xor(k1, xor(v0,1)))))))
end

function HW(x,nb_bits)
    nb = 0
    for i in 0:(nb_bits-1)
        if (x>>i)%2 == 1
            nb += 1
        end
    end
    nb
end

function find_key(n0,n1,v0,s4)
    if n0 == n1 && n1 == v0
        return s4
    elseif v0 == n1
        return xor(1, s4)
    else
        error("Mauvais nonce")
    end
end

IV = cut(0x00001000808c0001,64)
```

```
real_key = to_bits([0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7,
    0x15, 0x88, 09, 0xcf, 0x4f, 0x3c],0,16)
lst_n0 = [to_bits(nonces[i,:],0,8) for i in 1:size(nonces,1)]
lst_n1 = [to_bits(nonces[i,:],8,8) for i in 1:size(nonces,1)]

function outputs(output,l,jump)
    outputs = zeros(UInt8,size(traces,1))
    bit_output = cut(UInt8(output),8)
    for i in 1:jump
        outputs[i] = output
    end
    m = 0
    for i in 1:size(traces,1)
        if IV[l] == lst_n1[i][l]
            m = i
            break
        end
    end
    n0 = lst_n0[m][l]
    n1 = lst_n1[m][l]
    v0 = IV[l]
    s4 = bit_output[8]
    k0 = find_key(n0,n1,v0,s4)
    k1 = xor(xor(v0, 1)*xor(n0, n1), xor(v0, xor(k0, bit_output[7])))
    for i in (jump+1):jump:(size(traces,1))
        n0 = lst_n0[i][l]
        n1 = lst_n1[i][l]
        s0 = sbox_0(k0,k1,n0,n1,v0)
        s1 = sbox_1(k0,k1,n0,n1,v0)
        s2 = sbox_2(k0,xor(k1,((l in [57,58,59,60]) ? 1 : 0),n0,n1,v0))
        s3 = sbox_3(k0,k1,n0,n1,v0)
        s4 = sbox_4(k0,k1,n0,n1,v0)
        output_i = assemble([s0,s1,s2,s3,s4],5)
        for j in 0:(jump-1)
            outputs[i+j] = output_i
        end
    end
    outputs
end

outs = Array{Dict{Any,Any}}(undef,64)
for l in 1:64
    outs[l] = Dict()
    for output in 0:31
        out = outputs(output,l,1)
        occ = Dict()
        for i in 1:size(traces,1)
            if !(out[i] in keys(occ))
                occ[out[i]] = 1
            else
                occ[out[i]] += 1
            end
        end
        if length(keys(occ)) == 2
            outs[l][output]  = out
        end
    end
end


IM_hw = [[[get_mutual_information(map(x -> HW(x,5),outs[l][output]),traces[:,t])
      for t in 1:size(traces,2)] for output in keys(outs[l])] for l in 1:1]
IM_val = [[[get_mutual_information(outs[l][output],traces[:,t]) for t in 1:size(
     traces,2)] for output in keys(outs)] for l in 1:64]


plot()
for k in 1:length(keys(outs[1]))
    cles = [output for output in keys(outs[1])]
    out = cles[k]
```

```julia
    m = 0
    for i in 1:size(traces,1)
        if IV[1] == lst_n1[i][1]
            m = i
            break
        end
    end
    bit_output = cut(UInt8(out),8)
    n0 = lst_n0[m][1]
    n1 = lst_n1[m][1]
    v0 = IV[1]
    s4 = bit_output[8]
    k0 = find_key(n0,n1,v0,s4)
    k1 = xor(xor(v0, 1)*xor(n0, n1), xor(v0, xor(k0, bit_output[7])))
    plot!(IM_hw[1][k], label = "("*string(k0)*","*string(k1)*")")
end

key_guess = Array{UInt8}(undef,128)
for l in 1:1
    possible_s = [i for i in keys(outs[l])]
    s_max = 0
    maxi = 0
    for hyp_s in 1:length(IM_hw[l])
        val_max = maximum(IM_hw[l][hyp_s])
        if val_max > maxi
            s_max = possible_s[hyp_s]
            maxi = val_max
        end
    end
    println(s_max)
    m = 0
    for i in 1:size(traces,1)
        if IV[l] == lst_n1[i][l]
            m = i
            break
        end
    end
    bit_output = cut(UInt8(s_max),8)
    n0 = lst_n0[m][l]
    n1 = lst_n1[m][l]
    v0 = IV[l]
    s4 = bit_output[8]
    key_guess[l] = find_key(n0,n1,v0,s4)
    key_guess[l+64] = xor(xor(v0, 1)*xor(n0,n1),xor(v0, xor(key_guess[l],
     bit_output[7])))
end
```

Listing: Analysis in Julia of the traces following the established attack