Side-channel attacks on Ascon's S-box

Alexane Boldo
Internship from May to July 2025
Supervisor: Hélène Le Bouder
OCIF, IRISA

***Abstract*—Though side-channel attacks on the National Institute of Standards and Technology winner Ascon have succeeded, simple correlation power analysis attacks targeting the S-box have failed. However, some of its properties can give more information on potential attacks, like the direction of computation or its relatively simple equations. Therefore, this article is interested in finding if those weaknesses can lead to an attack, using multiple traces of the power consumption captured by a ChipWhisperer and analyzing their correlation with the output of the S-box. The theorized attack uses a link between the key and the output, and it was found that computing seems to leak better than writing in the register. However, we were unable to use the relatively easy equations from the S-box to find weaknesses, even while controlling the nonces by attacking the decryption phase.**

I. Introduction

Though mathematically secure, cryptographic algorithms can still be broken. Indeed, leakages happen during the computation of the encryption, which can be observed and analyzed to figure out the key. Such leakages can be the computation time, the power consumption, or the electromagnetic radiations and can lead to attacks recovering the key, called side-channel attacks (SCA). Therefore, a whole field of cybersecurity consists of analyzing which computations leak the most data.

This article focuses on Ascon [1], the winner of the National Institute of Standards and Technology (NIST) competition for Authenticated Encryption with Associated Data (AEAD), which became the standard for lightweight cryptography [2]. Multiple attacks [3], [4] based on Correlation Power Analysis (CPA) [5] have already been performed and analyzed [6] to find a stronger implementation of Ascon, using masking to render SCA harder.

However, to our knowledge, no CPA attack on the non-linear layer of the permutation were successful, although such permutations are known to provide leakage. Furthermore, an analysis provided by Sarry[7] deducts equations to link the output of this layer to the secret key.

Therefore, this article uses a CPA in order to determine where exactly the leakage is the clearer, by attacking the output of the substitution layer of the first-round of the permutation. The goal is to provide an analysis finding the best path to attack it.

This paper first describes Ascon-AEAD then the methodology used to attack it. Next, it presents the best choices for the CPA attack after a small presentation of its concept. Lastly, Sarry's equations [7] are used to try an attack, and its failure is analyzed.

II. Description of Ascon

A. Generalities

This paper focuses on Ascon-AEAD as described in the norm [2]. This paper especially focuses on the decryption phase to use the leaks in order to recover the key.

Data: 128-bit key K and nonce N, associated data A, ciphertext C and 128-bit tag T
Result: checks authentication with T, if it is authenticated returns plaintext P otherwise returns fail

This algorithm uses a 320-bit state, divided into 5 64-bit words and modified through 4 phases: the initialization, the associated data process, the ciphertext process, and the finalization. IV is a known constant given in the norm [2], and in the following, this paper refers to the rate as $r = 128$.

B. Ascon's permutation

This algorithm uses a permutation $p = p_L \circ p_S \circ p_C$, which is used at all phases of the encryption or decryption and is the target of the attack. Each notation B is described in more detail in the annex. This permutation is itself divided into three layers: the constant-addition layer p_C that adds to the end of the third word an 8-bit round-dependent constant; the substitution layer p_S, which is a 5-bit S-box taking the j^{th} bit of each word; and the linear-diffusion layer p_L rotating each word to provide diffusion to the cipher where $\forall i \in [\![1;5]\!], S_i \leftarrow \Sigma_i(S_i)$:

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

The substitution layer is the one attacked in this article. It is a non-linear permutation, providing confusion to the cipher and computing each column of the state separately, using the circuit 1. Formally, let's write $\sigma = p_S(S)$

The substitution layer computes for each $j \in [\![1;64]\!]$:

$$(\sigma_0^j, \sigma_1^j, \sigma_2^j, \sigma_3^j, \sigma_4^j) = S - box(S_0^j, S_1^j, S_2^j, S_3^j, S_4^j)$$

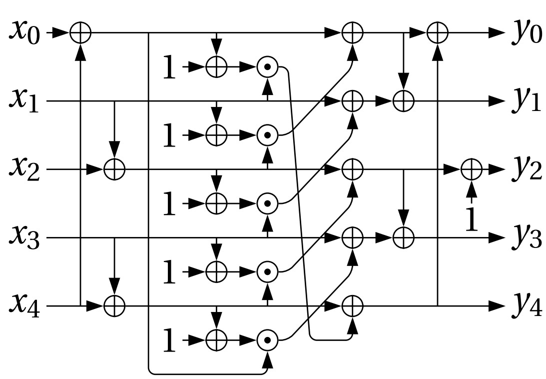


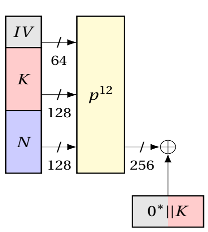
Fig. 1: Circuit to compute the S-box, from [1]

C. Ascon-AEAD's algorithm

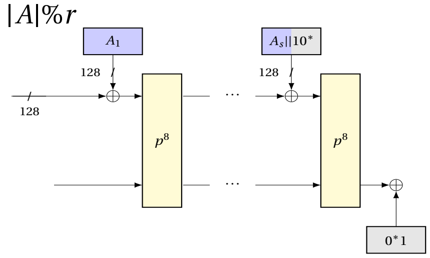
Each step of the algorithm is described here, and the complete figure from [8] for the encryption can be found in the annex 13.

1) Initialization: state creation and modification

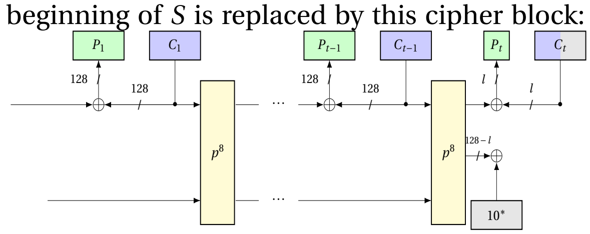
$$S \leftarrow p^{12}(\underbrace{IV}_{S_0}||\underbrace{K}_{S_1,S_2}||\underbrace{n}_{S_3,S_4})$$



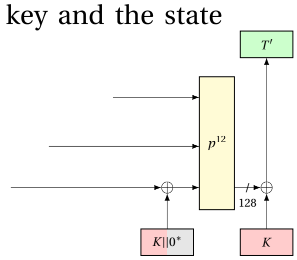
2) Associated data process: for the authentication of A, updates S using $A = (A_1||...||A_s)$ parsed into blocks of r-bits, except for the last of size $|A|\%r$



3) Ciphertext process: $C = (C_1||...||C_t)$ is parsed with C_i of size r except for C_t of size $l = |C|\%r$. Each block of the ciphertext is then XORed to S to find the associated plaintext, and then the beginning of S is replaced by this cipher block:



4) Finalization: computes the tag thanks to the key and the state



If $T' = T$, returns $P \leftarrow P_1||...||P_t$, otherwise returns `fail`

III. Methodology

We implemented our version of Ascon, given in this repository[2] to visualize the possible implementation weaknesses, which is later compared to the reference implementation[3] from the authors of Ascon [1]. Our version has the advantage to be able to follow other constants than those given by the norm.

In order to attack the implementations, a ChipWhisperer-Lite board with an XMEGA target microcontroller (CW1173) is used to capture the power consumption during the execution of the decryption, as it gives clearer traces than in real-life scenarios.

To facilitate the capture, triggers were added in the first round of the permutation, during the initialization stage, in order to start the recording of the power consumption.

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

[2] https://github.com/ABo-projets/Stage_L3

[3] https://github.com/ascon/ascon-c/blob/main/crypto_aead/asconaead128/ref

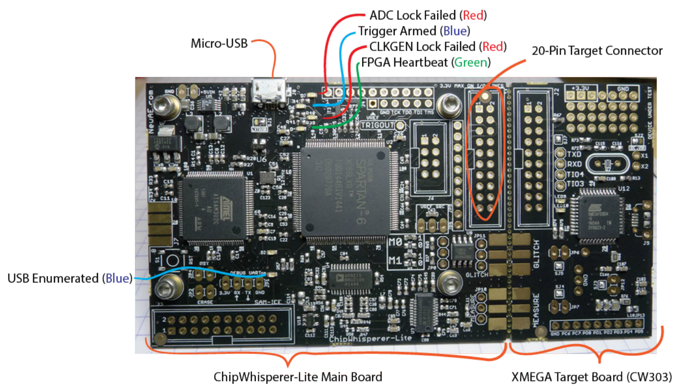


Fig. 2: ChipWhisperer from [9]

10K traces with a fixed key and variable nonces, 10K traces with variable keys and nonces, 10K traces with variable keys and a fixed nonce all during the p_S layer were recorded on both the reference implementation and our own. The following section explains these choices.

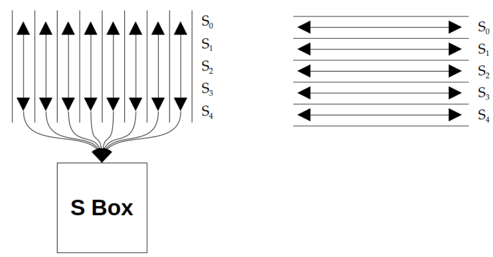
IV. Observations for CPA attacks

A. Correlation Power Analysis (CPA) definition

Correlation Power Analysis[5] is a type of attack using the recording of multiple traces of the execution of a known cryptographic algorithm to recover the private key. It can be blind (the attacker knows neither the plaintext nor the ciphertext) or not. It is often based on the divide-and-conquer approach, where the attacker recovers the key portion-by-portion (often byte-by-byte or bit-by-bit). The following explanation takes the notations used in the course [10].

1) **Campaign:**
   - Let's first define the target k that will take all its possible values (e.g., for one byte of the key, its 256 values)
   - Then the first step is to compute multiple traces of the targeted algorithm to gain observables (e.g., the plaintext and the power consumption as a function of time)
   - Find an attack path, i.e., a relation between observables and the target: $\mathscr{R}(k, O_S) = O_R$. It depends on physical functions, called leak functions, and cannot be computed
2) **Prediction:** Because the exact leak functions cannot be computed, the attacker has to find a model to approximate it (e.g., the Hamming Weight (HW) of the byte for the power consumption, as the consumption of writing in a register depends on whether it is a 1 or a 0 that is written), it gives a new relation model, called a theoretical attack path: $\mathscr{R}_m(k, O_S) = P_{m,k}$
3) **Confrontation:** The attacker has to find a distinguisher, i.e., a statistical function that puts a hypothesis k_d upfront of the others by confronting O_R and $P_{m,k}$ (e.g., the Pearson correlation between the vector of HW and the power consumptions, at each timestamp t). This also gives points of interest, i.e., timestamps where the target is used.

This attack is well established on the previous encryption standard AES [5], [11] by attacking the S-box; however in Ascon the S-box is computed vertically and not horizontally, whereas the results are written in the words, therefore horizontally.



(a) Computing (b) Writing in the register

Fig. 3: Comparison of the direction for computing the S-box and writing in the register for the first byte of the five words

Therefore it is pertinent to wonder, like Sarry in [7], whether the leak is horizontal or vertical, in order to determine if the S-box leaks because of the computation or of the writing in the register and to find better strategies of masking. Hereafter, this article refers to attacks on one byte of the last word of the state as horizontal attacks and to the ones on the concatenation of the 5 bits at position j on each word of the state as vertical attacks.

B. Choosing a theoretical attack path and a distinguisher

1) Horizontal attack on the S-box: In this part, the traces used are those measured on our implementation, with variable keys and a fixed nonce, in order to see what functions show the most influence from the key.

The goal is to see if there is an adequate distinguisher for the key hypothesis, using the HW of the first horizontal byte. Two statistical functions seem promising: the Pearson correlation, usually used in CPA attacks, which failed to conclude for more than 40K traces in [4], and the mutual information.

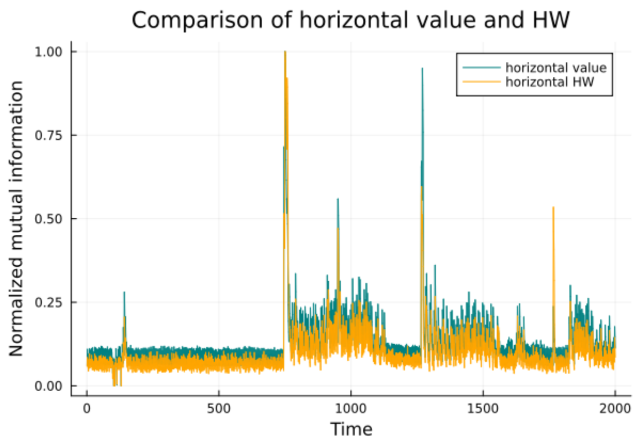


Fig. 4: Normalized mutual information between the power consumption and the Hamming weight or the value of the first byte of S_4

In the graph 4, there are a few points of interest that seem to show an influence from the key on the power consumption on a few instances, which is a good sign to be able to find the key later. To visualize how the points of interest for each byte correlate, the graph 5 shows that the use of each byte of key is successive, which is logical.

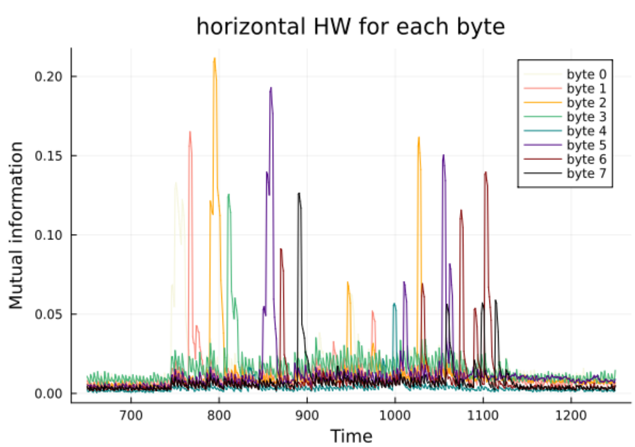


Fig. 5: Mutual information between power consumption and Hamming weight of each of the 8 bytes

To check if our implementation is worse than the reference one, the graph 6 is useful and shows that both of them have clear leaks. However, these leaks don't happen at the same time, since the S-box is not written exactly the same.

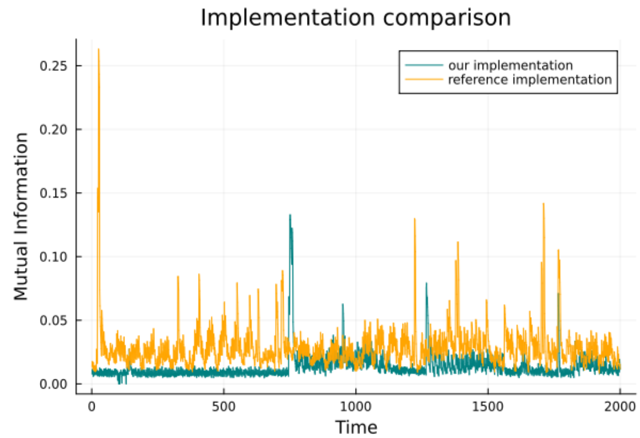


Fig. 6: Mutual information between power consumption and horizontal Hamming Weight on the reference implementation and on our own

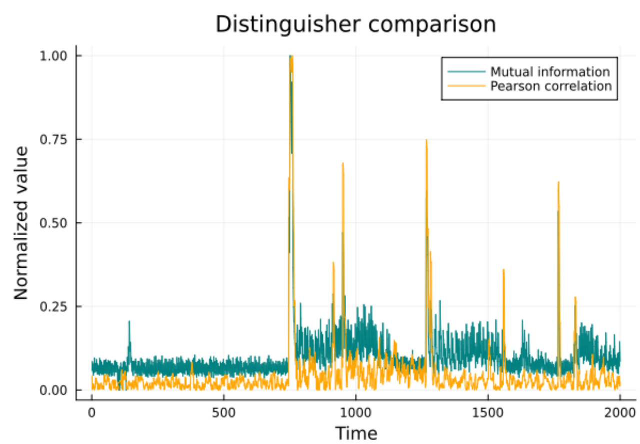


Fig. 7: Normalized mutual information and absolute Pearson correlation for a horizontal attack on the reference implementation

Let's now compare the mutual information with the Pearson correlation. In the graph 7, a reassuring observation is that points of interest are the same no matter the distinguisher. Furthermore, though the absolute value is greater for the Pearson correlation, points of interest appear slightly clearer after normalization with the mutual information, which can help on the attack.

2) Vertical attack on the S-box: First, let's use the same traces as before and compare to see if there is an adequate distinguisher using the HW of the first output of the S-box.

From the graph 8, it seems that there are also vertical leaks, which are fewer in Hamming weight than in value, which probably includes false ones. Furthermore, studying variable nonces also illus-

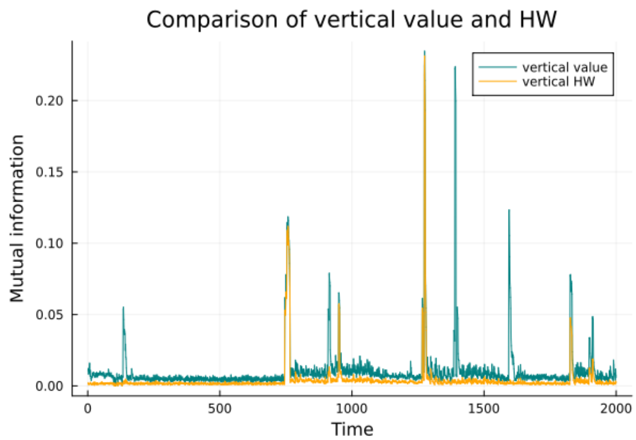


Fig. 8: Mutual information between power consumption and Hamming weight of the concatenation of the first bit of each of the word of S and its value

trates the leaks, which are, however, logically more noisy with variable nonces (see in annex 14).

Finally, let's compare horizontal and vertical leaks with graph 9. Counterintuitively, horizontal and vertical points of interest seem to coincide, which refutes the hypothesis that there is first a leak because of the computation and then another one because of writing in the register. A possible explanation would be that the mutual information is induced by the value of the bit that is both there horizontally and vertically, which could explain why correlations and mutual information are so low. However, the vertical leaks seem to be clearer, which seems to indicate that the computation leaks a lot, and it gives the attacker a clearer theoretical attack path. Furthermore, the chosen distinguisher for the following analysis is the mutual information, which looks slightly clearer.

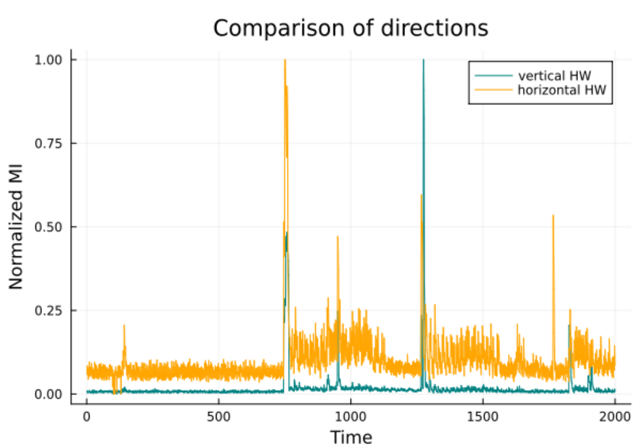


Fig. 9: Normalized mutual information for the horizontal and vertical Hamming weight

C. Working CPA attack with another attack path

Let's note that there is a working simple CPA attack, as proven by Daemen et al. in [3]. The idea is to first put the trigger before the linear-diffusion layer then have for key hypothesis the three bits of the key, which are XORed to have the output S_0^0, and simplifying equations thanks to their observations.

V. Using equations binding the input and output of the S-box

A. Description of the equation

Sarry [7] describes in his thesis equations linking the output of the S-box to the value of the key, depending on the value of the nonce and of the IV 10. These equations are described in the annex D.

(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

Fig. 10: Link between k_0^j and S_4^j depending on IV and N

B. Attack path

According to [6], attacking the S-box by distinguishing key hypotheses is very difficult and needs numerous traces, as there are multiple keys that give strong correlations to their vertical HW. We reproduced the experiment on 40K traces from the ChipWhisperer and were still unable to recover the key, as multiple false key bit hypotheses gave good results.

The goal is to use the equations described in the previous section V-A to recover the key thanks to the value of S_4. So the problem is reduced to directly finding the output of the S-box in certain conditions (i.e., convenient values of nonces that are chosen by the attacker when asking for decryption).

Therefore, an other idea would be to directly have hypotheses for the output of the S-box, and to later find the associated key thanks to table 10. To recover these traces, the attacker asks the device to decrypt random ciphertexts with random tags 10K times with chosen convenient nonces, and no matter the failure to authenticate, the algorithm goes through the attacked permutation from the initialization process.

Let's formalize our attack: there are 64 attacks for each column j of the state where the target t_j is $S-box(S_0^j||S_1^j||S_2^j \oplus \texttt{const}_{16-a}^j||S_3^j||S_4^j)$, the vertical output of the S-box in the first trace. To find the corresponding outputs, the attacker deducts the key and then finds the associated outputs for the other traces. The attack path links these values to the measured power consumption. The theoretical attack path selected in IV-B is the Hamming weight of the target t_j, and the distinguisher is the maximum of mutual information.

Once an output t_j is found, the bit t_j^4 is used to deduce k_0^j and this algorithm entirely recovers k_0. Once that is done, k_1 is recovered using the equations 2, where S_3^j has already been found as t_j^3.

$$S_3^j = (IV^j \oplus 1) \times (n_1^j \oplus n_0^j) \oplus IV^j \oplus k_0^j \oplus k_1^j \quad (1)$$

$$k_1^j = (IV^j \oplus 1) \times (n_1^j \oplus n_0^j) \oplus IV^j \oplus k_0^j \oplus S_3^j \quad (2)$$

C. Results

To try to find the key, the attack was repeated with more or less varying nonces.

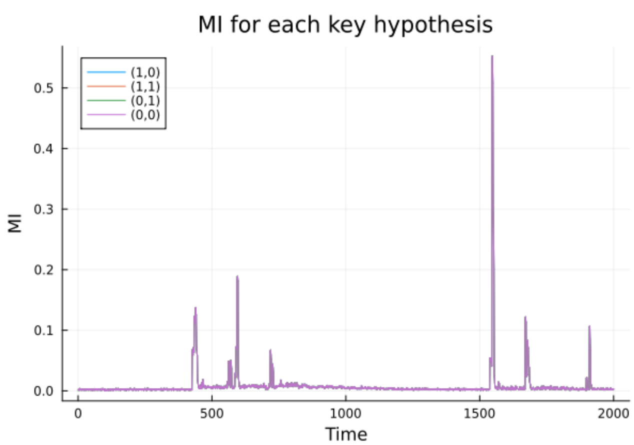


Fig. 11: MI between the HW of the outputs (associated key in the legend) and the power consumption, for each of the possible outputs for the first nonce with half of its bits fixed

Following our attacks, the graphs seem to show points of interest, however, having half the nonce fixed gives inconclusive results as all key hypotheses have the same mutual information with the power consumption 11. To compare, full random nonces were used and seemed to show more points of interest 12, though the key giving the best results was not the real key used by the target.

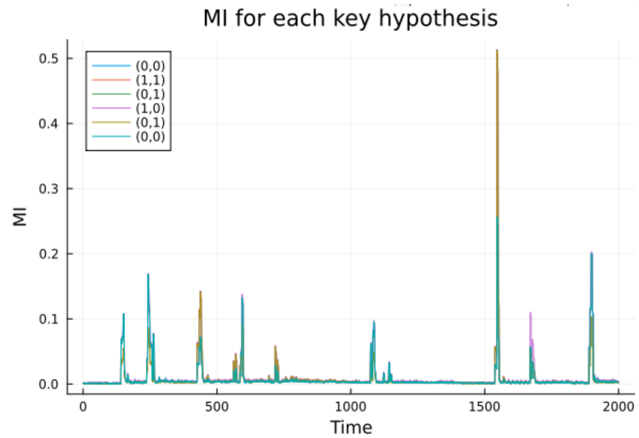


Fig. 12: Same as 11 but nonce with all its bits random

Though it is easy to distinguish between the correct HW and random ones (see annex 15), and though the S-box is supposed to provide confusion, false keys still give false points of interest. It therefore seems that false keys don't give enough random outputs. It stems from the fact that only two bits can change in the input, so there are very few possible outputs. The theorized attack is probably still too close to the attack from [4], proven difficult by [6].

VI. Conclusion

As the non-linear function of the Ascon protocol follows simple equations, it seemed to lead to easier leaks to use. Furthermore, the attacker has control over the nonces used in the decryption part of the algorithm, which seemed to be another weakness. However, we were unable to use those weaknesses to build a successful CPA attack on the S-box. Therefore, the S-box seems more secure in Ascon than in AES, as found by Nguyen et al. in [6], and having power over the nonce doesn't seem to be useful. Another idea would be to use a learning phase to be able to detect the output computed through the power consumption with a single trace and then use the given equations [7] to deduce the key. A belief propagation algorithm, first used on AES for side-channel attacks by Veyrat-Charvillon et al. in [12], like the ones proposed against other NIST competitors like Elephant by Sarry [7], could be interesting to later focus on.

REFERENCES

[1] D. C., E. M., M. F., and S. M., “Ascon v1.2: Lightweight authenticated encryption and hashing.” *Journal of Cryptology*, vol. 34, p. 33, 2021. [Online]. Available: https://doi.org/10.1007/s00145-021-09398-9

[2] 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

[3] D. J. and S. N., “Dpa on hardware implementations of ascon and keyak.” [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3075564.3079067

[4] R. K., A. A., D. W., K. J., and A. P., “Active and passive side-channel key recovery attacks on ascon.” [Online]. Available: https://csrc.nist.gov/CSRC/media/Events/lightweight-cryptography-workshop-2020/documents/papers/active-passive-reco/very-attacks-ascon-lwc2020.pdf

[5] B. E., C. C., and O. F., *Correlation Power Analysis with a Leakage Model*. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-28632-5_2

[6] N. V. S., G. V., and C. P., “Practical second-order cpa attack on ascon with proper selection function.” [Online]. Available: https://cascade-conference.org/Paper/CASCADE25/final-versions/cascade2025-cycleB/cascade2025b-final31.pdf

[7] S. M., “Side channel analysis against aead.” [Online]. Available: https://theses.hal.science/tel-04816066v1

[8] L. B. H., T. G., and A. D., “Théorie de la cryptographie.”

[9] Chipwhisperer documentation. [Online]. Available: https://chipwhisperer.readthedocs.io/en/latest/getting-started.html

[10] L. B. H. and L. R., “Introduction aux attaques physiques de systèmes électroniques.”

[11] D. M., B. E., N. J., F. J., B. L., R. E., and D. J. Jr., “Advanced encryption standard (aes).” [Online]. Available: https://www.nist.gov/publications/advanced-encryption-standard-aes

[12] V.-C. N., G. B., and S. F., *Soft Analytical Side-Channel Attacks*.

APPENDIX A
ABBREVIATIONS

Abbreviation	Meaning
SCA	Side-channel attacks
NIST	National Institute of Standards and Technology
AEAD	Authenticated encryption with associated data
CPA	Correlation Power Analysis
MI	Mutual information
HW	Hamming weight
IV	Initialization vector
S-box	Substitution box

APPENDIX B
NOTATIONS

Notation	Definition
K	Secret key
k_0,k_1	first and last 64 bits of K
N	Public nonce (i.e. changes at each encryption)
n_0,n_1	first and last 64 bits of N
S	320-bit state
IV	64-bit initialization vector of value `0x00001000808c0001`
p	The permutation for Ascon $p = p_L \circ p_S \circ p_C$, see II-B
S-box	Non-linear permutation function
`const`	round-constants for p_c indexed by $i \in [\![0;15]\!]$
$x\|\|y$	Concatenation of bitstrings x and y
$x >>> k$	Circular shift to the right of the word x by k bits
$\oplus$	Bitwise xor
$S_i, \forall i \in [\![0;4]\!]$	i^{th} 64-bit word of the state S
w^j	j^{th} bit of the word w (e.g., S_i^j)
p^k	$\underbrace{p \circ p \circ \ldots \circ p}_{k\times}$
0^k	Concatenation of k zeros $\underbrace{(0\|\|0\|\|\ldots\|0)}_{k\times}$
$S_{<=k}$	For the bits numbered from 1 to 320, bits S 1 through k
$S_{>k}$	Bits of S from k+1 to 320

APPENDIX C
ASCON-AEAD MODE

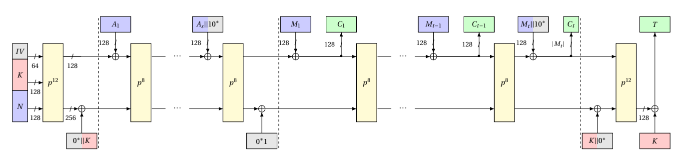


Fig. 13: Ascon Encryption process from [8]

APPENDIX D
EQUATIONS TO LINK THE OUTPUT OF THE S-BOX TO THE KEY

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

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

Then if $IV^j = 0$:

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

Otherwise, if $IV^j = 1$:

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

APPENDIX E
COMPLEMENTARY GRAPHS

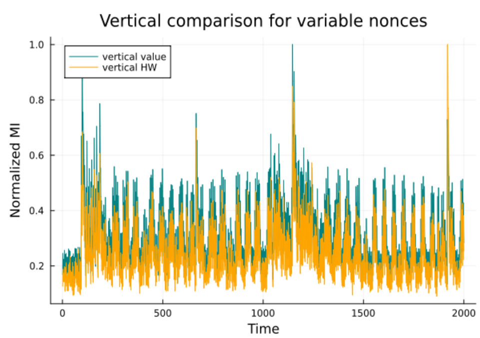


Fig. 14: Mutual information between power consumption and Hamming weight of the concatenation of the first bit of each of the word of S and its value like 8 but for random nonces

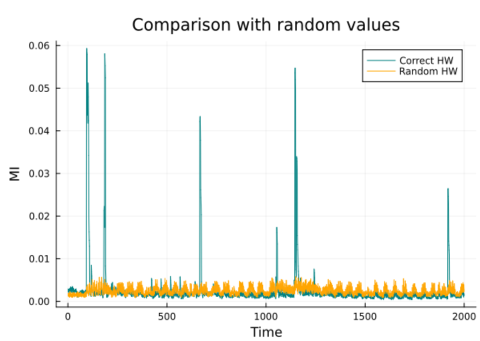


Fig. 15: Mutual information between power consumption and vertical HW or random possible HW

APPENDIX F
MAIN CODE

```
#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,
            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);
    }

```

```c
    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++){
        //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 1: 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

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);
```

```c
//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
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 2: 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 3: Implementation for permutation.c

```c
#include "ascon.h"

#ifndef PERMUTATION_H
#define PERMUTATION_H

//functions for the permutation
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 4: 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;
    }

  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;
```

```c
}

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 5: 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

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 6: Implementation for trace capture for the ChipWhisperer

```julia
"""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]
    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 7: Analysis in Julia of the traces following the established attack