Hardware Implementation of ASCON

Aneesh Kandi[1], Anubhab Baksi[2], Tomas Gerlich[3*], Sylvain Guilley[4], Peizhou Gan[2], Jakub Breier[5], Anupam Chattopadhyay[2], Ritu Ranjan Shrivastwa[4], Zdenek Martinasek[3*], and Shivam Bhasin[2]

[1] Indian Institute of Technology Madras, India

[2] Nanyang Technological University, Singapore

[3] Brno University of Technology, Brno, Czechia

[4] Télécom Paris, Paris, France; Secure-IC, Cesson-Sévigné, France

[5] Silicon Austria Labs, Graz, Austria

aneeshkandi@smail.iitm.ac.in, anubhab001@e.ntu.edu.sg, xgerli02@vut.cz, sylvain.guilley@telecom-paristech.fr, peizhou.gan@ntu.edu.sg, jbreier@jbreier.com, anupam@ntu.edu.sg, ritu-ranjan.shrivastwa@secure-ic.com, martinasek@vut.cz, sbhasin@ntu.edu.sg

Abstract. In this work, we present various hardware implementation for ASCON. We cover encryption + tag generation as well as decryption + tag verification for ASCON AEAD and also ASCON hash function. On top the usual (unprotected) implementation, we present side channel protection (threshold countermeasure) and triplication/majority based fault protection. The side channel and fault protections work orthogonal to each other (i.e., either one can be turned on/off without affecting the other). We also show ASIC and FPGA benchmarks for our implementations.

Keywords: ASCON · Hardware Implementation · Side Channel Attack · Threshold Implementation · Fault Attack · Countermeasure

1 Introduction

`ASCON` is a lightweight hash function and AEAD (authenticated encryption with associated data) family [DEMS19]. It has recently been selected as the primary choice in the LWC project by NIST [NIS23]. As the main aim of lightweight cryptography are resource-constrained embedded devices (such as Internet-of-Things appliances), one of the main concerns are efficient implementations and protection against physical attacks.

1.1 Our Contributions

In this work, we present various hardware (Verilog) implementations of `ASCON` AEAD and hash. We consider regular (unprotected) implementations as well as side-channel and fault attack protection.

In summary, we implement and benchmark the followings:

(α) Unprotected `ASCON` (encryption + tag generation, decryption + tag verification; and hashing).
(β) Side channel attack protected `ASCON` using *threshold implementation*.
(γ) Fault injection protected `ASCON` using triplication/majority.
(δ) Combined side channel attack and fault injection protected `ASCON` using threshold and triplication/majority.

Our implementations use simple interface. The side channel and fault protections can be turned on/off easily depending on the use-case (it is not necessary that both the countermeasures have to be used all the time) by making minimal adjustment to the interface. Our source-codes are aceesible as an open-source project[1].

[*]The work is supported by the Ministry of Interior of Czech Republic under grant VJ02010010.

[1]https://github.com/aneeshkandi14/ascon-hw-public.

1.2 Previous Works

The hardware implementation of ASCON has been explored before, for instance, [WEHM+22, GMK16]. However, as far as we can tell, no side channel protected (threshold) implementation exists. Also, the common countermeasure against fault (that rely on triplication/majority) has apparently not been explored before.

2 ASCON Description

The ASCON family has 2 variants - ASCON 128 (block size = 64) and ASCON 128a (block size = 128). Both of those take 128-bit key and nonce, and have a 320-bit state. A 128-bit tag is generated after encryption + tag generation and it is verified in decryption + tag verification.

2.1 Permutation

The main strength of ASCON lies in the permutation process. p^n represents the number of rounds of the permutation. There are two types of permutation of *i)* p^a consisting of a rounds (used for initialization and finalization) and *ii)* p^b consisting of b rounds (used for data processing).

The 320-bit state S of the ASCON is divided into 5 registers of 64 bits each. $S = x_0||x_1||x_2||x_3||x_4$. These 5 registers are then sent for further processing. Each permutation round is further divided into three layers – the constant addition layer, the substitution layer, and the linear diffusion layer.

Constant Addition Layer: In this layer, a constant term is added with the x_2 register word. The constant term added depends on the current round number of the permutation. For p^a, round constant c_r is used and for ρ^b, round constant c_{a-b+r} is used, where r is the number of rounds. The number of rounds and related constants are given in Table 1.

Table 1: Round constants and number of rounds for ASCON

ρ^{12}	ρ^8	ρ^6	Constant	ρ^{12}	ρ^8	ρ^6	Constant
0			000000000000000000f0	6	2	0	00000000000000000096
1			000000000000000000e1	7	3	1	00000000000000000087
2			000000000000000000d2	8	4	2	00000000000000000078
3			000000000000000000c3	9	5	3	00000000000000000069
4	0		000000000000000000b4	10	6	4	0000000000000000005a
5	1		000000000000000000a5	11	7	5	0000000000000000004b

Substitution Layer This layer implements the 5-bit SBox operation of the ASCON which is the only non-linear operation within the permutation. The SBox is applied to each bit-slice comprising of 5 bits from all the registers, where x_0 acts as MSB and x_4 acts as LSB.

The ASCON SBox can be given by the following look-up table: (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).

Linear Diffusion Layer This layer is used to shuffle the bits of each register internally with the help of right rotation and XOR. It is performed as per the equations given here:

$$\begin{aligned}
x_0 &= x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28)\\
x_1 &= x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39)\\
x_2 &= x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6)\\
x_3 &= x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17)\\
x_4 &= x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)
\end{aligned}$$

2.2 Encryption + Tag Generation and Decryption + Tag Verification

`ASCON` is a family of authenticated encryption and verified decryption parameterized by 4 variables, key (k), rate (r) and internal number of rounds (a and b) for the permutation computation. The key length is, $k \leq 128$ bits and other parameters vary depending on the type of `ASCON`. Inputs for the authenticated encryption are plaintext P, associated data A, key K, and nonce N (k-bits), and outputs are the ciphertext C and tag T. Inputs for the verified decryption are key k, nonce N (k-bits), ciphertext C, and tag T and output is plaintext P if the tag is successfully verified else $\perp$.

The operation of `ASCON` can be divided into four sub-routines, namely:

1. Initialization
2. Processing associated data
3. Processing plaintext/ciphertext
4. Finalization

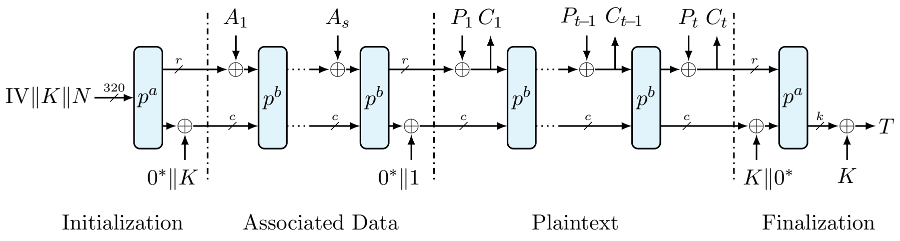

Figure 1: `ASCON` encryption and tag generation

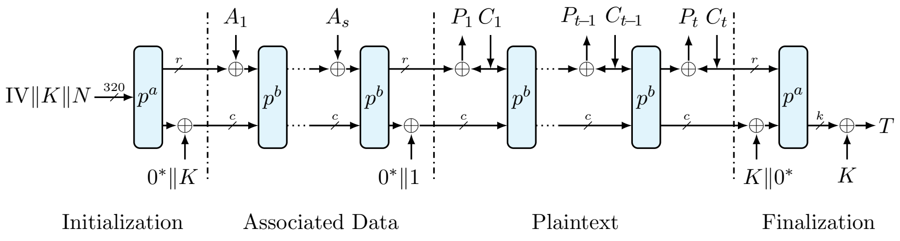

Figure 2: `ASCON` decryption and tag verification

At initialization, the algorithm is initialized by creating a state of 320 bits by concatenating the fixed initialization vector, key, and the nonce, which are then passed through a rounds of permutation and xor operation of the least significant $(320 - r)$ bits with the key padded with 0's (on the left) before proceeding to the next stage.

In the next stage, associated data is absorbed in the algorithm by dividing it into datasets of r bits each and the last dataset is padded with a 1 followed by 0's to make the length equal to r.

The next stage processes the plaintext in a similar way and in addition, it generates ciphertexts in encryption and does the opposite for decryption. Processing of associated data and plaintext both have b rounds of the permutation.

The finalization stage generates a tag in encryption which is used in the finalization stage of decryption to verify if the processed data is correct. Pictorial description can be found in Figures 1 and 2[2].

[2]We acknowledge "TikZ for Cryptographers" [Jea16].

2.3 Hashing

ASCON Hashing is based on Sponges and is parameterized by 4 variables — maximal output length (h), rate (r), and internal number of rounds (a and b) for the permutation computation. The input for the hashing algorithm is message data M and the output is the hash data H.

The operation of ASCON can be divided into four sub-routines, namely:

1. Initialization
2. Absorbing message
3. Squeezing

At the initialization stage, the algorithm is initialized by creating a state of 320 bits by padding the fixed initialization vector with 0's on the LSB side, which is then passed through a rounds of permutation.

In the next stage, the message data is absorbed into the algorithm similar to the plain text processing stage mentioned above. It uses b rounds of permutation.

In the last stage, the ASCON state is first passed through a permutation rounds which generates the first block of hash data. The output is then passed through b rounds of permutations till all the blocks of hash data are generated.

3 (Unprotected) Hardware Implementation

3.1 Substitution Layer

The substitution layer employs a 5-bit SBox (see Section 2). Possibly the most straightforward approach to implementing the SBox is by utilizing a look-up table. However, this method incurs a significant area cost.

An alternative approach involves using the coordinate functions. In general, it can be stated that the coordinate function-based implementation takes much less area than what would be required for a look-up-based implementation. Expressed in the algebraic normal form, the coordinate functions of the ASCON SBox are as given:

$$\begin{aligned}
y_0 &= x_4x_1 \oplus x_3 \oplus x_2x_1 \oplus x_2 \oplus x_1x_0 \oplus x_1 \oplus x_0 \\
y_1 &= x_4 \oplus x_2x_3 \oplus x_3 \oplus x_3x_1 \oplus x_2 \oplus x_1x_2 \oplus x_1 \oplus x_0 \\
y_2 &= x_4x_3 \oplus x_4 \oplus x_2 \oplus x_1 \oplus 1 \\
y_3 &= x_4x_0 \oplus x_3x_0 \oplus x_4 \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0 \\
y_4 &= x_4x_1 \oplus x_4 \oplus x_3 \oplus x_1x_0 \oplus x_1
\end{aligned}$$

3.2 Linear Layer

The linear layer, which is discussed in Section 2, can be realized using right rotation and XOR. In our implementation, we opted to use only XOR operation. This approach requires 2 XOR operations for each row, resulting in a total of 640 XOR operations for the entire layer[3]. Although both methods needs same area, XOR implementation is more flexible in terms of code. Additionally, it is easier to transform it into the threshold implementation of the linear layer using the latter method.

[3]Equivalently, the linear layer can also be implemented using 320 XOR3 operations. The problem of implementation with higher input XOR gates is studied in the literature [BFI21, BDK+21, LWS+22]. Also note that the same binary matrix is considered in [RBC23].

4 Protection Against Side Channel Attacks

Side channel attacks, particularly those relying on information from power consumption or electromagnetic emanation, are of prominent concern while dealing with the physical security of the ciphers [MOP07, Pee13, Bil15, Lom10]. It has been systematically shown that a cipher with sufficient classical security claims falls short against an adversary equipped with a side channel attack set-up. It, therefore, goes without saying, understanding the attacks and finding low-cost countermeasures are among the top research priorities by/for the community.

Side channel attacks are based on the connection between a (a priori or learned) model and any intermediate variable in the implementation that might be leaking. Therefore, the countermeasures attempt to destroy the linkage of the model and the intermediate variables. *Masking* [MOP07, Chapter 9] is considered a prominent countermeasure. A masking scheme randomly distributes each intermediate to introduce randomness in a way that the overall algorithmic flow in the cipher is unchanged, but the randomized operations make the side channel leakage free from the intermediate variables. Depending on the strength of the attacker, various degrees of masking can be adopted.

4.1 Threshold Implementation

The threshold implementation (TI) is a form of masking, and is among the top recommendations against the side channel attacks [NRR06, Bil15, KNP12, JGC+20, BNN+15, NRS11, NNR19] specially for protecting the hardware implementation.

Typically, the TI of an affine function is considered straightforward, while that of a non-linear (in most block ciphers, the only non-linear component is the SBox) function is considered a strenuous task to accomplish. The TI of a given SBox can be realized either through *without decomposition* (the SBox is implemented as a combinational circuit) or *with decomposition* (the SBox is implemented as a sequential circuit) [JGC+20, BGST23].

4.2 Hardware Implementation of Threshold

The Threshold Implementation technique is renowned for its hardware implementation simplicity. Each phase in this method is concealed by the utilization of random numbers generated through an external entropy source such as a *True Random Number Generator* (TRNG). In the unprotected `ASCON`, the `ASCON` State is partitioned into three shares, namely S_0, S_1, and S_2, with S representing the `ASCON` State. The three shares must meet the following condition: $S = S_0 \oplus S_1 \oplus S_2$.

All three shares undergo separate processing and are subsequently combined at the end. The `ASCON` permutation, for instance, consists of three stages as mentioned in Section 2.1. Each share has a distinct constant addition layer, substitution layer, and linear layer that are cleverly designed so that the output of all three shares can be merged at the conclusion of the permutation phase to yield the same state value as in unprotected `ASCON`. Figure 3 shows the flow of `ASCON` permutation with threshold implementation; where the 3 individual shares for the each of round constant addition, substitution layer and the linear diffusion layer are shown.

The schematic of the `ASCON` permutation with threshold is presented in Figure 3, which demonstrates how state S is divided into three shares. Each of the shares is then processed with distinct permutation processes, where RCi represents the round constant layer, SBOXi represents the substitution layer, and LLi represents the linear layer for share i. See Figure 7 for the flowchart representation of `ASCON` permutation.

4.3 `ASCON` SBox Sharing (for Threshold)

The SBox, being the only non-linear component, is considered the hardest to implement in threshold. The minimum number of shares needed is 1 more than the algebraic degree of the SBox (thus, we

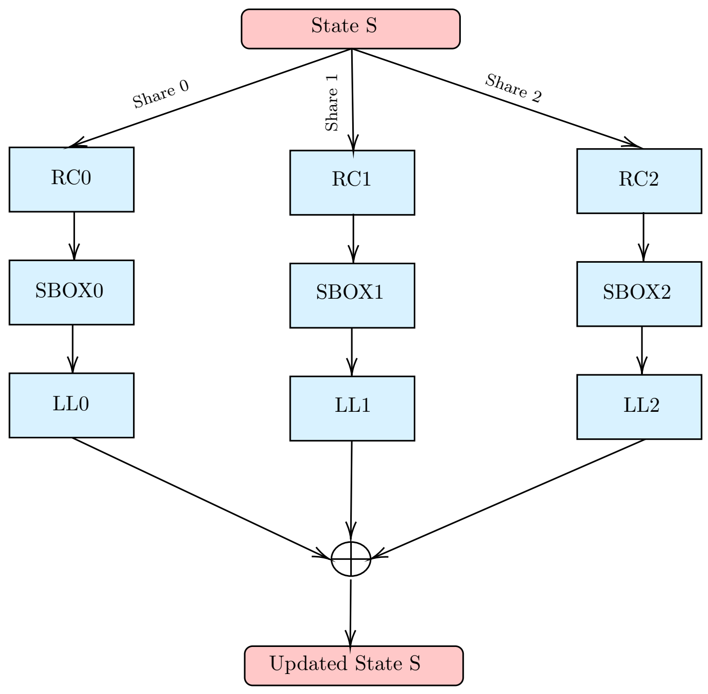

Figure 3: `ASCON` permutation under 3-share threshold (schematic)

need at least 3 shares), and the total number of monomials in the shares is related to the product of the number of shares and total number of monomials in the coordinate functions.

We show two sharing options for the `ASCON` SBox subsequently. Both the options use 3 shares and are available in our implementation (either one can be chosen). Given the coordinate functions (Section 3.1), note that, the following relationships hold: $y_i = \bigoplus_{j=0}^{2} y_{ij}$ given $x_i = \bigoplus_{j=0}^{2} x_{ij}$ for $i = 0, 1, \ldots, 4$.

SBox Threshold 1 The corresponding benchmark is shown in Table 2.

Share 0:

$$\begin{aligned}
y_{00} &= x_{00} \oplus x_{01}x_{11} \oplus x_{01}x_{12} \oplus x_{01} \oplus x_{11}x_{21} \oplus x_{11}x_{41} \oplus x_{11}x_{02} \oplus x_{11}x_{22} \oplus x_{11}x_{42} \\
&\quad \oplus x_{11} \oplus x_{21}x_{12} \oplus x_{21} \oplus x_{31} \oplus x_{41}x_{12} \oplus x_{02}x_{12} \oplus x_{12}x_{22} \oplus x_{12}x_{42} \oplus x_{12} \oplus x_{22} \oplus x_{32} \\
y_{10} &= x_{10} \oplus x_{01} \oplus x_{11}x_{21} \oplus x_{11}x_{31} \oplus x_{11}x_{22} \oplus x_{11}x_{32} \oplus x_{11} \oplus x_{21}x_{31} \oplus x_{21}x_{12} \\
&\quad \oplus x_{21}x_{32} \oplus x_{21} \oplus x_{31}x_{12} \oplus x_{31}x_{22} \oplus x_{31} \oplus x_{41} \oplus x_{02} \oplus x_{12}x_{22} \oplus x_{12}x_{32} \oplus x_{22}x_{32} \\
&\quad \oplus x_{22} \oplus x_{32} \oplus x_{42} \\
y_{20} &= x_{20} \oplus x_{11} \oplus x_{21} \oplus x_{31}x_{41} \oplus x_{31}x_{42} \oplus x_{41}x_{32} \oplus x_{41} \oplus x_{12} \oplus x_{32}x_{42} \oplus x_{42} \oplus 1 \\
y_{30} &= x_{30} \oplus x_{01}x_{31} \oplus x_{01}x_{41} \oplus x_{01}x_{32} \oplus x_{01}x_{42} \oplus x_{01} \oplus x_{11} \oplus x_{21} \oplus x_{31}x_{02} \\
&\quad \oplus x_{31} \oplus x_{41}x_{02} \oplus x_{41} \oplus x_{02}x_{32} \oplus x_{02}x_{42} \oplus x_{02} \oplus x_{12} \oplus x_{22} \oplus x_{42} \\
y_{40} &= x_{40} \oplus x_{01}x_{11} \oplus x_{01}x_{12} \oplus x_{11}x_{41} \oplus x_{11}x_{02} \oplus x_{11}x_{42} \oplus x_{11} \oplus x_{31} \oplus x_{41}x_{12} \\
&\quad \oplus x_{41} \oplus x_{02}x_{12} \oplus x_{12}x_{42} \oplus x_{12} \oplus x_{32}
\end{aligned}$$

Share 1:

$$\begin{aligned}
y_{01} &= x_{00}x_{10} \oplus x_{00}x_{11} \oplus x_{00}x_{12} \oplus x_{10}x_{20} \oplus x_{10}x_{40} \oplus x_{10}x_{01} \oplus x_{10}x_{21} \oplus x_{10}x_{41} \\
&\quad \oplus x_{10}x_{02} \oplus x_{10}x_{22} \oplus x_{10}x_{42} \oplus x_{10} \oplus y_{20}x_{11} \oplus x_{20}x_{12} \oplus x_{20} \oplus x_{30} \oplus x_{40}x_{11} \oplus x_{40}x_{12} \\
y_{11} &= x_{00} \oplus x_{10}x_{20} \oplus x_{10}x_{30} \oplus x_{10}x_{21} \oplus x_{10}x_{31} \oplus x_{10}x_{22} \oplus x_{10}x_{32} \oplus x_{20}x_{30} \oplus x_{20}x_{11} \\
&\quad \oplus x_{20}x_{31} \oplus x_{20}x_{12} \oplus x_{20}x_{32} \oplus x_{20} \oplus x_{30}x_{11} \oplus x_{30}x_{21} \oplus x_{30}x_{12} \oplus x_{30}x_{22} \oplus x_{30} \oplus x_{40} \\
y_{21} &= x_{10} \oplus x_{30}x_{40} \oplus x_{30}x_{41} \oplus x_{30}x_{42} \oplus x_{40}x_{31} \oplus x_{40}x_{32} \oplus x_{40} \\
y_{31} &= x_{00}x_{30} \oplus x_{00}x_{40} \oplus x_{00}x_{31} \oplus x_{00}x_{41} \oplus x_{00}x_{32} \oplus x_{00}x_{42} \oplus x_{00} \oplus x_{10} \oplus x_{20} \\
&\quad \oplus x_{30}x_{01} \oplus x_{30}x_{02} \oplus x_{40}x_{01} \oplus x_{40}x_{02} \oplus x_{40} \\
y_{41} &= x_{00}x_{10} \oplus x_{00}x_{11} \oplus x_{00}x_{12} \oplus x_{10}x_{40} \oplus x_{10}x_{01} \oplus x_{10}x_{41} \oplus x_{10}x_{02} \oplus x_{10}x_{42} \\
&\quad \oplus x_{10} \oplus x_{30} \oplus x_{40}x_{11} \oplus x_{40}x_{12}
\end{aligned}$$

Share 2:

$$\begin{aligned}
y_{02} &= x_{02} \\
y_{12} &= x_{12} \\
y_{22} &= x_{22} \\
y_{32} &= x_{32} \\
y_{42} &= x_{42}
\end{aligned}$$

SBox Threshold 2

Share 0:

$$\begin{aligned}
y_{00} &= x_{01}x_{30} \oplus x_{01}x_{31} \oplus x_{01}x_{32} \oplus x_{01} \oplus x_{02}x_{30} \oplus x_{02}x_{31} \oplus x_{02}x_{32} \oplus x_{02} \oplus x_{11} \oplus x_{12} \\
&\quad \oplus x_{30}x_{40} \oplus x_{30}x_{41} \oplus x_{30}x_{42} \oplus x_{30} \oplus x_{31}x_{40} \oplus x_{31}x_{41} \oplus x_{31}x_{42} \oplus x_{31} \oplus x_{32}x_{40} \\
&\quad \oplus x_{32}x_{41} \oplus x_{32}x_{42} \oplus x_{32} \\
y_{10} &= x_{01}x_{40} \oplus x_{01}x_{41} \oplus x_{01}x_{42} \oplus x_{01} \oplus x_{02}x_{40} \oplus x_{02}x_{41} \oplus x_{02}x_{42} \oplus x_{02} \oplus x_{11}x_{40} \\
&\quad \oplus x_{11}x_{41} \oplus x_{11}x_{42} \oplus x_{11} \oplus x_{12}x_{40} \oplus x_{12}x_{41} \oplus x_{12}x_{42} \oplus x_{12} \oplus x_{21} \oplus x_{22} \oplus x_{30} \oplus x_{31} \\
&\quad \oplus x_{32} \oplus x_{40} \oplus x_{41} \oplus x_{42} \\
y_{20} &= x_{01}x_{11} \oplus x_{01}x_{12} \oplus x_{01} \oplus x_{02}x_{11} \oplus x_{02}x_{12} \oplus x_{02} \oplus x_{21} \oplus x_{22} \oplus x_{30} \oplus x_{31} \oplus x_{32} \oplus 1 \\
y_{30} &= x_{01} \oplus x_{02} \oplus x_{11}x_{21} \oplus x_{11}x_{22} \oplus x_{11}x_{30} \oplus x_{11}x_{31} \oplus x_{11}x_{32} \oplus x_{11} \oplus x_{12}x_{21} \\
&\quad \oplus x_{12}x_{22} \oplus x_{12}x_{30} \oplus x_{12}x_{31} \oplus x_{12}x_{32} \oplus x_{12} \oplus x_{21}x_{30} \oplus x_{21}x_{31} \oplus x_{21}x_{32} \oplus x_{21} \oplus x_{22}x_{30} \\
&\quad \oplus x_{22}x_{31} \oplus x_{22}x_{32} \oplus x_{22} \oplus x_{30} \oplus x_{31} \oplus x_{32} \oplus x_{40} \oplus x_{41} \oplus x_{42} \\
y_{40} &= x_{01}x_{30} \oplus x_{01}x_{31} \oplus x_{01}x_{32} \oplus x_{02}x_{30} \oplus x_{02}x_{31} \oplus x_{02}x_{32} \oplus x_{11} \oplus x_{12} \oplus x_{21}x_{30} \\
&\quad \oplus x_{21}x_{31} \oplus x_{21}x_{32} \oplus x_{21} \oplus x_{22}x_{30} \oplus x_{22}x_{31} \oplus x_{22}x_{32} \oplus x_{22} \oplus x_{30}x_{40} \oplus x_{30}x_{41} \oplus x_{30}x_{42} \\
&\quad \oplus x_{30} \oplus x_{31}x_{40} \oplus x_{31}x_{41} \oplus x_{31}x_{42} \oplus x_{31} \oplus x_{32}x_{40} \oplus x_{32}x_{41} \oplus x_{32}x_{42} \oplus x_{32} \oplus x_{40} \oplus x_{41} \oplus x_{42}
\end{aligned}$$

Share 1:

$$\begin{aligned}
y_{01} &= x_{00}x_{30} \oplus x_{00}x_{31} \oplus x_{00}x_{32} \oplus x_{00} \\
y_{11} &= x_{00}x_{40} \oplus x_{00}x_{41} \oplus x_{00}x_{42} \oplus x_{00} \oplus x_{10}x_{40} \oplus x_{10}x_{41} \oplus x_{10}x_{42} \oplus x_{10} \\
y_{21} &= x_{00}x_{10} \oplus x_{00}x_{12} \oplus x_{00} \oplus x_{02}x_{10} \oplus x_{20} \\
y_{31} &= x_{00} \oplus x_{10}x_{20} \oplus x_{10}x_{22} \oplus x_{10}x_{30} \oplus x_{10}x_{31} \oplus x_{10}x_{32} \\
&\quad \oplus x_{10} \oplus x_{12}x_{20} \oplus x_{20}x_{30} \oplus x_{20}x_{31} \oplus x_{20}x_{32} \oplus x_{20} \\
y_{41} &= x_{00}x_{30} \oplus x_{00}x_{31} \oplus x_{00}x_{32} \oplus x_{10} \oplus x_{20}x_{30} \oplus x_{20}x_{31} \oplus x_{20}x_{32}
\end{aligned}$$

Share 2:

$$
\begin{aligned}
y_{02} &= x_{10}\\
y_{12} &= x_{20}\\
y_{22} &= x_{00}x_{11} \oplus x_{01}x_{10}\\
y_{32} &= x_{10}x_{21} \oplus x_{11}x_{20}\\
y_{42} &= x_{20}
\end{aligned}
$$

5 Protection Against Fault Injection Attacks

Since fault injection attacks [BBB+23] rely on some form of error propagation, the idea is to use redundancy. The same circuit is replicated (could be in the temporal domain or in the spatial domain) and depending on the power, we may need to duplicate or triplicate:

- Duplicate and compare works against *Differential Fault Attack* (DFA).
- Triplicate and take majority works against *Statistical Ineffective Fault Attack* (SIFA), although duplication-based SIFA countermeasures do exist.

In our implementation, we employed triplication and majority-based countermeasure techniques. Specifically, all the procedures are executed thrice, and the final output is determined by selecting the majority output from the three. In cases where all three outputs differ, a random number is produced as the output. Figure 4 shows the illustrates the working of the triplication countermeasure. F1, F2 and F3 are the three instances of `ASCON` whose output is finally combined with the majority operation.

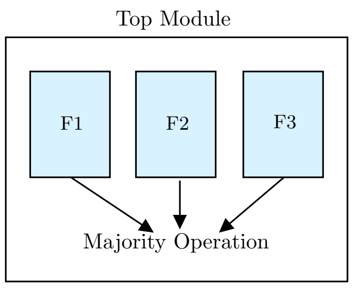

Figure 4: Triplicattion based countermeasure (schematic)

Note that we considered triplication-based SIFA countermeasure for the interest of simplicity. The overall area can be reduced by using a more complicated duplication-based SIFA countermeasure as explained in [BKK+20, BBB+20].

6 Architecture and Interface

The input data consist of the key, 128-bit nonce, associated data (AD), plain text (PT), control signals and random numbers (generated using an external entropy source, which is not considered within the scope). The output data consists of the cipher text (CT), 128-bit tag and ready signals to indicate the end of the processing. Some of the important signals are explained below:

- `keyxSI` signal is of width 3 bits. The LSB bit carries the key information, and the other 2 bits carry random numbers, which is utilised for threshold implementation.

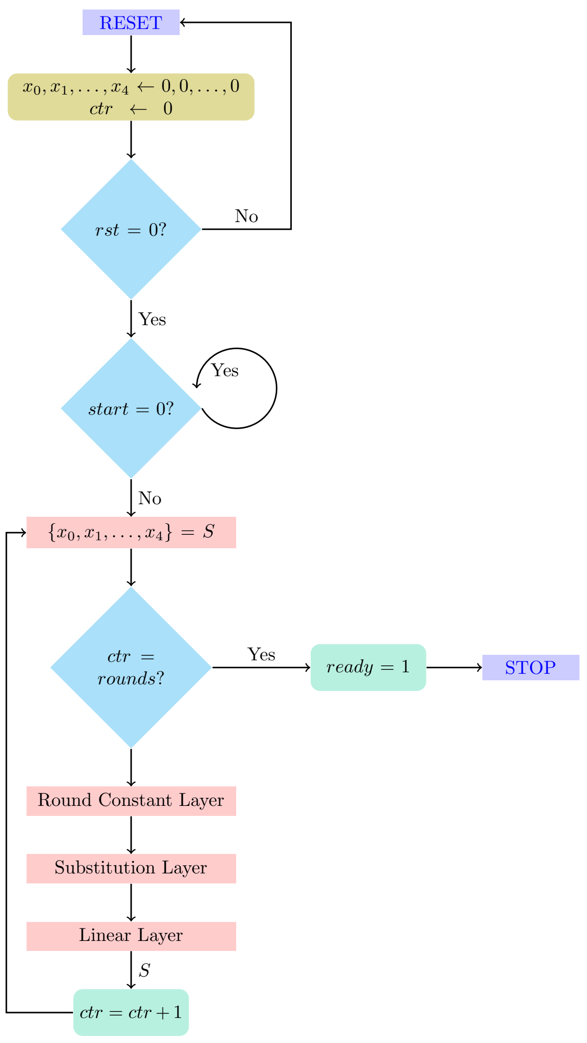

Figure 5: Finite state diagram for `ASCON` permutation (with round counter)

- `noncexSI, plain_textxSI, associated_dataxSI` signals are of width 3 bits, and the distribution of the bits is similar to `key`.
- `encryption_startxSI` and `decryption_startxSI` are 1-bit control pulses that signal the start of encryption/decryption, respectively.
- `rxSI` signals is of width 7 bits carrying random numbers which is utilized for threshold implementation.

Figure 6 represents the top-level diagram of the proposed `ASCON` architecture, which includes all the signals mentioned above.

The followings parameters can be tuned to any specific configuration:

- `k` is the key size
- `r` is the rate or the block size
- `a` and `b` internal number of rounds which vary based on the `ASCON` variant
- `l` is the length of associated data
- `y` is the length of plain text

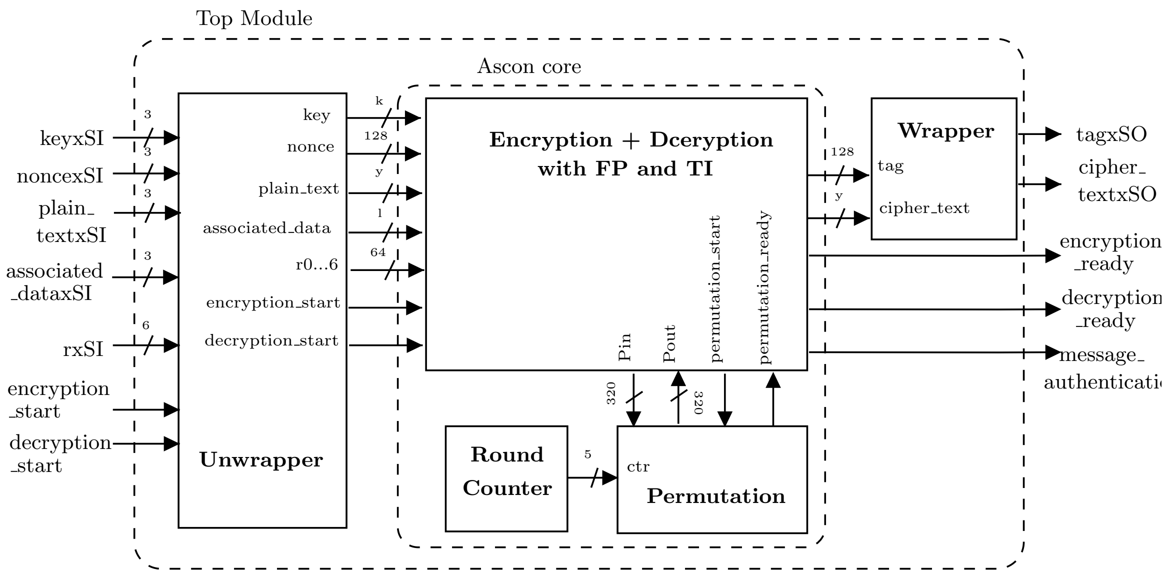

Figure 6: Top level diagram of ASCON

- TI is set to 1 for threshold implementation; else 0
- FP is set 1 for fault protection; else 0

Figure 5 shows a flowchart representation of the ASCON Permutation, which begins with a reset state where the round counter is set to 0. The state then waits for the `permutation_start` signal to be activated before proceeding to divide the ASCON state S into five registers, which are updated after every round. Each round consists of three stages, namely round constant addition, substitution layer, and linear layer, as described in section 2. At the end of each round, the counter variable `ctr` is incremented by 1.

Figure 7 depicts the finite state diagram for the ASCON encryption process, which begins with the RESET state, where all signals are reset to 0. The process then proceeds to the IDLE state, where the ASCON state is initialized based on the key, nonce, and cipher configuration. The system remains in this state until the `encryption_start` pulse is activated.

Upon receiving the start pulse, the system enters the INITIALIZE state, where the initialization process occurs (as shown in Figure 1). The next state, ASSOCIATED_DATA, is where the associated data is processed. The associated data is processed in multiple blocks, and the permutation process runs on each block one after the other. Once all the blocks are processed, the system enters the PTCT state, where the plain text is processed, and cipher text is generated. This stage is similar to the ASSOCIATED_DATA stage. The final stage is the FINALIZE state, where the tag is generated. After this, the system enters the DONE state, where it waits for new data and the next start signal.

7 ASIC and FPGA Benchmarks

In Table 2, we show the ASIC (STM 130nm) benchmark for the ASCON SBox with threshold implementation and the linear layer. The corresponding 3 shares of the SBox is given in Section 4.3 ('SBox Threshold 1').

In Table 3a, we show the benchmark results for both unprotected and protected ASCON encryption and decryption on an ASIC (STM 130nm) platform and in Table 3b on Xilinx based Kintex-7 FPGA. The results indicate that the threshold implementation version occupies approximately 3.7 times the area of (α) in ASIC, while in FPGA, it occupies approximately 4.2 times the area of (α).

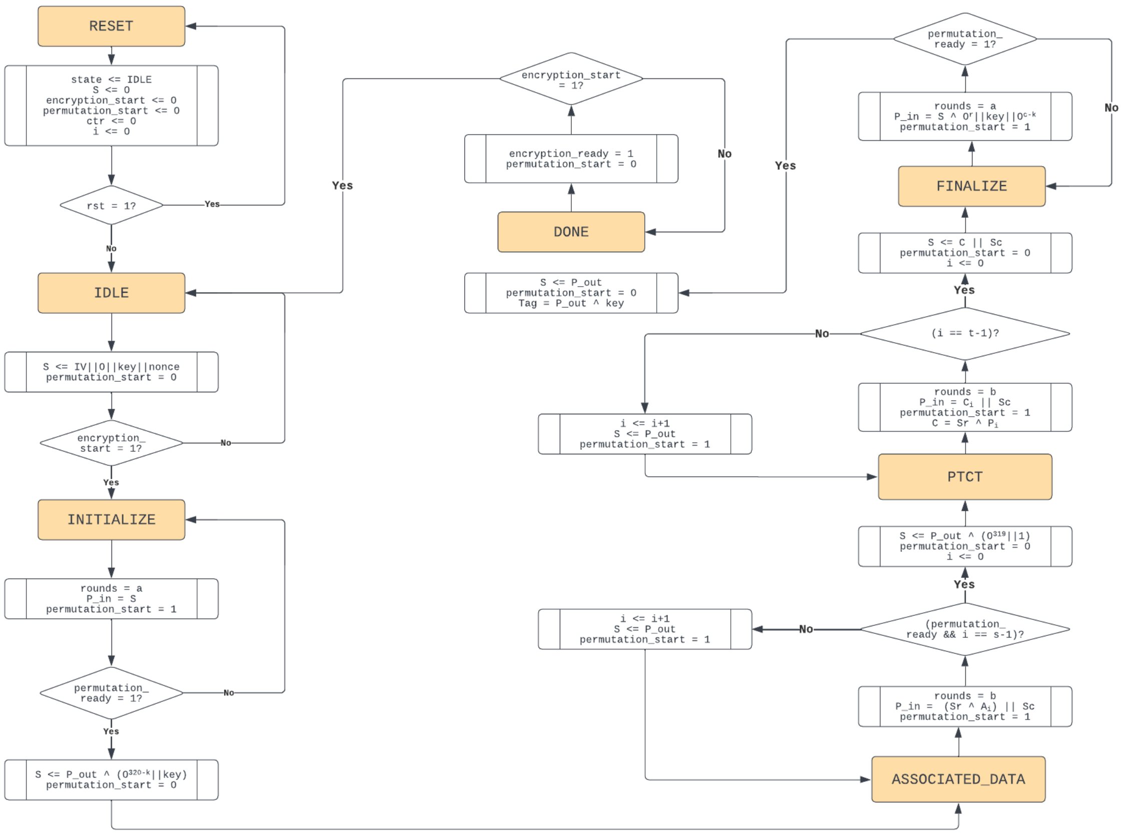

Figure 7: Finite state diagram for ASCON hardware

In Table 4 we show the benchmark results for the following configurations on ASIC (STM 130nm) platform: (α) unprotected ASCON, (β) ASCON with TI, (γ) ASCON with fault protection, (δ) ASCON with TI and fault protection. The results indicate that the implementation of ASCON with threshold requires approximately 3.7× the area of (α), while the implementation with fault protection requires approximately 2.7× the area of (α), and the implementation with both threshold and fault protection requires approximately 9.7× the area (α).

In Table 5, we show the benchmark for ASCON hash on Kintex-7 FPGA. We include unprotected hash and hash with threshold in the benchmark. The results indicate that the threshold implementation leads to a significant increase in the number of registers, with a 196.7% increase compared to the unprotected hash.

8 Conclusion and Outlook

This work presents a full-stack hardware suite for ASCON hash and AEAD [DEMS19]. There seems to be no comprehensive side channel and fault attack protected hardware implementation of this cipher, so we expect our work would become useful in the forthcoming future.

We only use triplication-majority based fault countermeasure in this work for simplicity, duplication-based fault countermeasure does exist [BKK+20]. This can be covered in a future scope.

A hardware interface for LWC is proposed by GMU [KDT+22]. It is possible to make our code compliant to the API (somewhat comparable to [SRBP22]), and this task is left as a future work.

Table 2: ASIC benchmarks (STM 130nm)

Design	**SBox TI**	**Linear Layer**
Gates	56	320
Surface (μm^2)	524.472	9037.056
Delay (fs)	1686	424

Table 3: Benchmarks for protected and unprotected `ASCON`

(a) ASIC (STM 130nm)

Design	**Cells**	**Area** (μm^2)	**Critical Path (ps)**	**Power (nW)**	
				Leakage	**Dynamic**
Encryption + Tag generation	5426	73803	8595	0	827861.117
Decryption + Tag verification	5025	71873	8586	0	644860.995
Encryption + Tag generation TI	20364	273857	10001	0	3522435.310
Decryption + Tag verification TI	20191	274688	9981	0	3647338.988

(b) FPGA (Kintex-7)

Design	**LUT**	**F/F**	**Max. Freq. (MHz)**
Encryption + Tag generation	944	734	181
Decryption + Tag verification	1058	735	181
Encryption + Tag generation TI	3977	2174	166
Decryption + Tag verification TI	3795	2179	156

Finally, one may be interested in evaluating the effect of side channel attack on the unprotected implementation and how the protected implementation protects against it.

References

BBB$^+$20. Anubhab Baksi, Shivam Bhasin, Jakub Breier, Anupam Chattopadhyay, and Vinay B. Y. Kumar. Feeding three birds with one scone: A generic duplication based countermeasure to fault attacks (extended version). Cryptology ePrint Archive, Paper 2020/1542, 2020. https://eprint.iacr.org/2020/1542. 8

BBB$^+$23. Anubhab Baksi, Shivam Bhasin, Jakub Breier, Dirmanto Jap, and Dhiman Saha. A survey on fault attacks on symmetric key cryptosystems. *ACM Comput. Surv.*, 55(4):86:1–86:34, 2023. 8

BDK$^+$21. Anubhab Baksi, Vishnu Asutosh Dasu, Banashri Karmakar, Anupam Chattopadhyay, and Takanori Isobe. Three input exclusive-or gate support for boyar-peralta's algorithm. In Avishek Adhikari, Ralf Küsters, and Bart Preneel, editors, *Progress in Cryptology - INDOCRYPT 2021 - 22nd International Conference on Cryptology in India, Jaipur, India, December 12-15, 2021, Proceedings*, volume 13143 of *Lecture Notes in Computer Science*, pages 141–158. Springer, 2021. 4

BFI21. Subhadeep Banik, Yuki Funabiki, and Takanori Isobe. Further results on efficient implementations of block cipher linear layers. *IEICE Trans. Fundam. Electron. Commun. Comput. Sci.*, 104-A(1):213–225, 2021. 4

BGST23. Anubhab Baksi, Sylvain Guilley, Ritu-Ranjan Shrivastwa, and Sofiane Takarabt. From substitution box to threshold. Cryptology ePrint Archive, Paper 2023/633, 2023. https://eprint.iacr.org/2023/633. 5

Bil15. Begül Bilgin. *Threshold Implementations As Countermeasure Against Higher-Order Differential Power Analysis*. PhD thesis, Katholieke Universiteit Leuven and University of Twente, 2015. https://www.esat.kuleuven.be/cosic/publications/thesis-256.pdf. 5

BKK$^+$20. Anubhab Baksi, Vinay B. Y. Kumar, Banashri Karmakar, Shivam Bhasin, Dhiman Saha, and Anupam Chattopadhyay. A novel duplication based countermeasure to statistical ineffective fault

Table 4: ASIC benchmarks for protected and unprotected ASCON (STM 130nm)

Design	Cells	Area (μm^2)	Critical Path (ps)	Power (nW)	
				Leakage	Dynamic
Unprotected ASCON	7157	98524	8520	0	762520.083
ASCON with Fault protection	19150	258224	8518	0	2346943.378
ASCON with TI	26248	364320	9830	0	4369303.426
ASCON with Fault protection and TI	69692	948544	9832	0	11124794.73

Table 5: FPGA benchmarks for unprotected and protected versions of ASCON hash (Kintex-7)

Design	LUT	Registers	Slice	LUT Logic	Max. Freq. (MHz)
Hash	870	912	313	870	203
Hash TI	3930	2706	1169	3930	171

analysis. Cryptology ePrint Archive, Report 2020/1268, 2020. https://eprint.iacr.org/2020/1268. 8, 11

BNN+15. Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, Natalia Tokareva, and Valeriya Vitkup. Threshold Implementations of Small S-boxes. *Cryptography and Communications*, 7(1):3–33, 2015. 5

DEMS19. Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. Ascon v1.2. Submission to NIST, 2019. https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/round-2/spec-doc-rnd2/ascon-spec-round2.pdf. 1, 11

GMK16. Hannes Gross, Stefan Mangard, and Thomas Korak. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. Cryptology ePrint Archive, Paper 2016/486, 2016. https://eprint.iacr.org/2016/486. 2

Jea16. Jérémy Jean. TikZ for Cryptographers. https://www.iacr.org/authors/tikz/, 2016. 3

JGC+20. Arpan Jati, Naina Gupta, Anupam Chattopadhyay, Somitra Kumar Sanadhya, and Donghoon Chang. Threshold implementations of gift: A trade-off analysis. *IEEE Trans. Inf. Forensics Secur.*, 15:2110–2120, 2020. 5

KDT+22. Jens-Peter Kaps, William Diehl, Michael Tempelmeier, Ekawat Homsirikamol, , and Kris Gaj. Hardware api for lightweight cryptography v1.1 (with support for sca-protected implementations). GMU ATHENa website, 2022. https://cryptography.gmu.edu/athena/LWC/LWC_HW_API_v1_1.pdf. 11

KNP12. Sebastian Kutzner, Phuong Ha Nguyen, and Axel Poschmann. Enabling 3-share threshold implementations for any 4-bit s-box. Cryptology ePrint Archive, Report 2012/510, 2012. https://eprint.iacr.org/2012/510. 5

Lom10. Victor Lomné. *Power and Electro-Magnetic Side-Channel Attacks: Threats and Countermeasures*. PhD thesis, Docteur de l'Université Montpellier II, 2010. https://sites.google.com/site/victorlomne/research. 5

LWS+22. Qun Liu, Weijia Wang, Ling Sun, Yanhong Fan, Lixuan Wu, and Meiqin Wang. More inputs makes difference: Implementations of linear layers using gates with more than two inputs. *IACR Cryptol. ePrint Arch.*, page 747, 2022. 4

MOP07. Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - Revealing the secrets of smart cards*. Springer, 2007. 5

NIS23. NIST. Lightweight Cryptography Standardization Process: NIST Selects Ascon, February 7 2023. https://csrc.nist.gov/News/2023/lightweight-cryptography-nist-selects-ascon. 1

NNR19. Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Decomposition of permutations in a finite field. *Cryptogr. Commun.*, 11(3):379–384, 2019. 5

NRR06. Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *International conference on information and communications security*, pages 529–545. Springer, 2006. 5

NRS11. Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *Journal of Cryptology*, 24(2):292–321, 2011. 5

Pee13. Eric Peeters. *Advanced DPA Theory and Practice: Towards the Security Limits of Secure Embedded Circuits*. Springer-Verlag New York, 1 edition, 2013. 5

RBC23. Soham Roy, Anubhab Baksi, and Anupam Chattopadhyay. Quantum implementation of ascon linear layer. Cryptology ePrint Archive, Paper 2023/617, 2023. `https://eprint.iacr.org/2023/617`. 4

SRBP22. Ambati Sathvik, Tirunagari Rahul, Anubhab Baksi, and Vikramkumar Pudi. Hardware implementation of spoc-128. Cryptology ePrint Archive, Paper 2022/119, 2022. `https://eprint.iacr.org/2022/119`. 11

WEHM$^+$22. Xiangdong Wei, Mohamed El-Hadedy, Sergiu Mosanu, Zhengping Zhu, Wen-Mei Hwu, and Xinfei Guo. Reco-hcon: A high-throughput reconfigurable compact ascon processor for trusted iot. In *2022 IEEE 35th International System-on-Chip Conference (SOCC)*, pages 1–6, 2022. 2