Research Institute of Electrical Communication

Tohoku University

ASCON Hardware Implementation

HSU TING YU

Environmentally Conscious Secure Information Systems Laboratory

Prof. Homma Naofumi

Abstract

This paper introduces ASCON, a lightweight authenticated encryption algorithm selected by NIST as the standard for lightweight cryptography. ASCON is designed for both efficiency and security, making it well-suited for resource-constrained devices such as IoT and embedded systems. Its sponge-based construction enables both encryption and authentication, ensuring data integrity and confidentiality.

This study focuses on understanding the ASCON algorithm. The encryption process consists of several key stages, including initialization, associated data processing, encryption, and finalization. Its design prioritizes simplicity and efficiency, making it a strong candidate for hardware acceleration.

The proposed implementation has been verified through RTL-level simulations using Icarus Verilog, with comprehensive waveform analysis conducted in GTKWave. This work provides insights into ASCON's potential for lightweight cryptographic applications and serves as a foundation for future hardware optimizations.

Keywords: Lightweight Cryptography, ASCON, Hardware Security, Authenticated Encryption, AEAD, IoT Security, Embedded Systems

Contents

1 Introduction

With the rapid growth of wearable devices and small-scale IoT applications, the need for lightweight cryptographic algorithms has increased significantly. Traditional encryption methods, originally designed for high-performance systems, must now adapt to environments with limited resources. Many of these applications require encryption for short-term security rather than long-term confidentiality. For example, in One-Time Passwords (OTP), data protection is only necessary for a brief period.

ASCON, first introduced in 2014, was selected as the NIST Lightweight Cryptography Standard in 2023 due to its balance of security, efficiency, and hardware compatibility. Unlike conventional cryptographic algorithms that require substantial computing power, ASCON is specifically designed for embedded systems. It offers high-speed authenticated encryption with associated data (AEAD) and hashing functions while keeping power consumption low.

ASCON is widely applicable in wearable technology, smart home devices, industrial IoT systems, and automotive security. For example, in wireless sensor networks, it enables energy-efficient encryption for data transmission. In secure boot mechanisms, ASCON can verify firmware updates with minimal computational overhead. The algorithm's design includes features such as a non-linear S-box, which provides resistance against side-channel attacks at a theoretical level. However, its actual resistance in hardware implementations depends on specific design techniques. Power consumption and electromagnetic leakage, for instance, can still introduce vulnerabilities not directly mitigated by the algorithm itself. As a result, ongoing research (4) focuses on optimizing hardware implementations to enhance resistance against side-channel attacks.

This study explores ASCON's hardware implementation and examines existing optimization techniques, such as algorithmic restructuring, to improve its resilience against side-channel attacks. The objective is to evaluate its practicality in real-world applications where both lightweight encryption and efficiency are critical.

2 ASCON

2.1 Introduction of ASCON

ASCON operates on a 320-bit internal state, which is updated through permutation functions during encryption, decryption, and hashing. The state is divided into two parts: an outer part S_r (rate) and an inner part S_c (capacity), where $S = S_r \parallel S_c$. The rate r and capacity $c = 320 - r$ vary depending on the ASCON variant. During permutation, the state is further split into five 64-bit words x_0, x_1, x_2, x_3, x_4, formally represented as:

$$S = S_r \parallel S_c = x_0 \parallel x_1 \parallel x_2 \parallel x_3 \parallel x_4 \quad \text{(each } x_i \text{ is 64 bits).}$$

Two permutation functions p^a (permutation with a rounds) and p^b (permutation with b rounds) govern the state updates. For example, ASCON-128 uses p^6 (6 rounds) during initialization and p^{12} (12 rounds) for finalization.

ASCON consists of two primary variants based on functionality: AEAD (Authenticated Encryption with Associated Data) and Hashing.

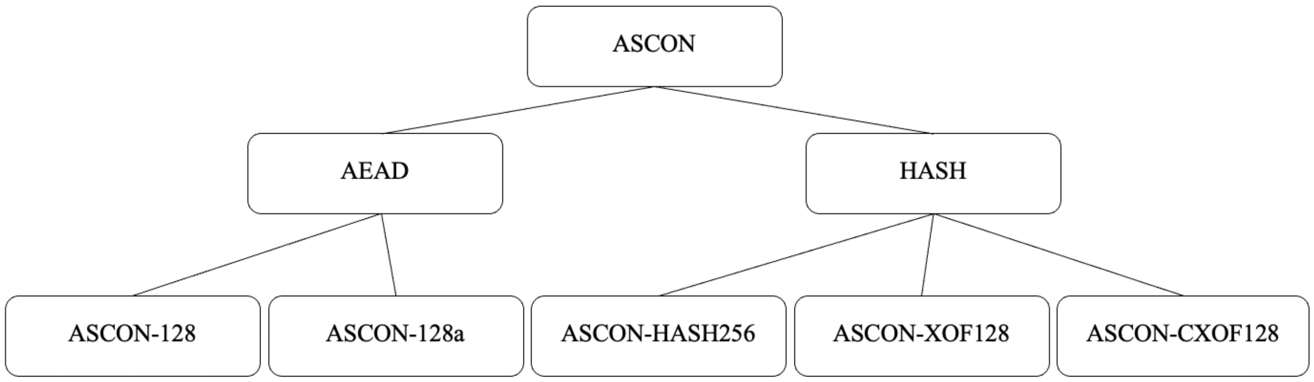


Figure 1: ASCON Variant

ASCON-AEAD. This category provides nonce-based authenticated encryption, ensuring confidentiality, integrity, and authenticity of data. The subvariants include:

- **ASCON-128** A standard AEAD scheme offering 128-bit security strength, suitable for general AEAD applications.
- **ASCON-128a** Compared to ASCON-128, this variant uses a larger block size (rate of 128 bits instead of 64 bits), making it more efficient for applications requiring high throughput but short message lengths, such as low-latency or resource-constrained environments (1).

ASCON-HASH. This category focuses on cryptographic hash functions and extendable-output functions (XOFs). The subvariants include:

- **ASCON-Hash256** A hash function that produces a fixed 256-bit output, providing a security strength of 128 bits.
- **ASCON-XOF128** An extendable-output function (XOF) that allows users to select the output length, making it suitable for applications requiring flexible digest sizes, such as random number generation or variable-length message hashing.
- **ASCON-CXOF128** Similar to ASCON-XOF128 but incorporates a *customization string* as an additional input, enabling domain separation for different application-specific hashing needs.

These variants and their implementations are discussed in detail in (1), with a particular emphasis on the suitability of ASCON-128 and ASCON-128a for various scenarios. This paper specifically focuses on ASCON-AEAD.

2.2 Authenticated Encryption

Authenticated Encryption (AE) is a cryptographic technique that ensures both confidentiality and integrity of data using symmetric-key encryption, where the same key is used for encryption and decryption. Traditional encryption methods only guarantee confidentiality, but AE integrates authentication to prevent unauthorized modifications.

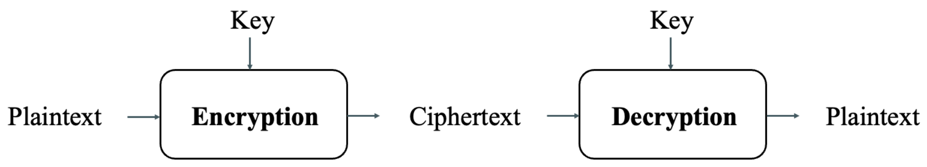


Figure 2: Symmetric Encryption

2.2.1 Authenticated Encryption with Associated Data (AEAD)

Authenticated Encryption with Associated Data (AEAD) extends AE by allowing additional, non-encrypted data to be authenticated. This is particularly useful in applications where metadata, such as packet headers, need authentication but should remain in plaintext. AEAD schemes, such as ASCON-AEAD, are optimized for efficiency and security in lightweight cryptographic applications.

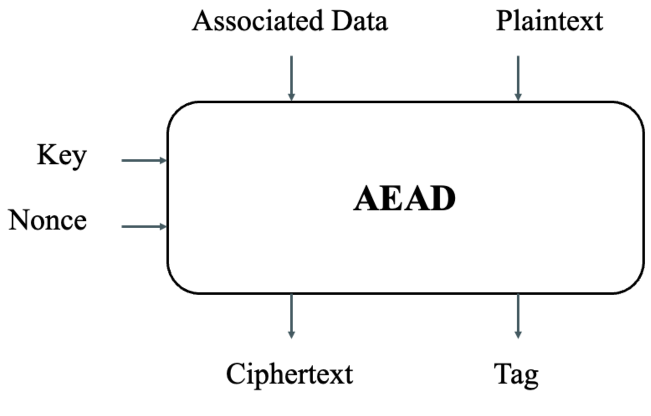


Figure 3: Authenticated Encryption with Associated Data (AEAD)

2.2.2 Interface Specifications and Parameter

The ASCON interface specifications define the necessary input and output parameters for its authenticated encryption and decryption processes. This section presents the key components involved in ASCON's encryption and decryption functions, as well as a comparison of parameter variations between ASCON-128 and ASCON-128a.

Input.

- K: A 128-bit secret key.
- N: A 128-bit nonce, which must be unique for each encryption operation.
- A: Associated data, divided into r-bit blocks A_i, which remains unencrypted but contributes to authentication.
- P: Plaintext, divided into r-bit blocks P_i.

Output.

- C: Ciphertext, consisting of r-bit blocks C_i, corresponding to the encrypted plaintext.
- T: A 128-bit authentication tag that ensures data integrity and authenticity.

Encryption Function.

$$\text{ASCON-AEAD.enc}(K, N, A, P) = (C, T) \tag{1}$$

Decryption Function.

$$\text{ASCON-AEAD.dec}(K, N, A, C, T) = \begin{cases} P, & \text{if } T \text{ is valid} \\ \perp, & \text{otherwise} \end{cases} \tag{2}$$

where $\perp$ denotes an invalid output in the case of authentication failure.

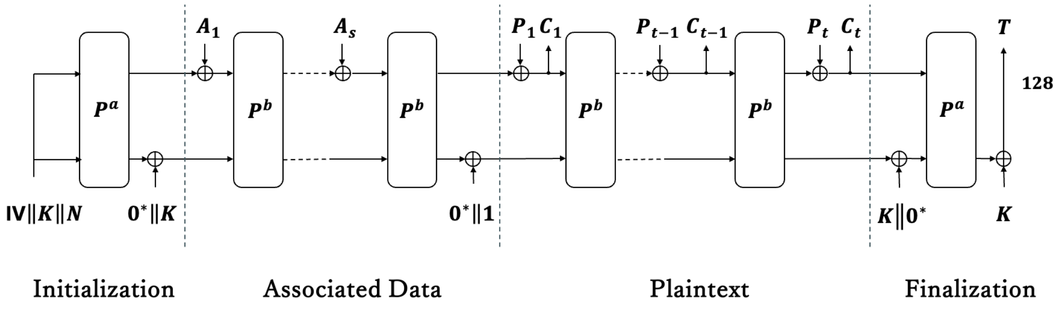
The primary difference between ASCON-128 and ASCON-128a lies in the data block size and the number of permutation rounds used in the processing phase (p^b).

Name	**Bit size of**				**Rounds**	
	Key	**Nonce**	**Tag**	**Data block**	p^a	p^b
ASCON-128	128	128	128	64	12	6
ASCON-128a	128	128	128	128	12	8

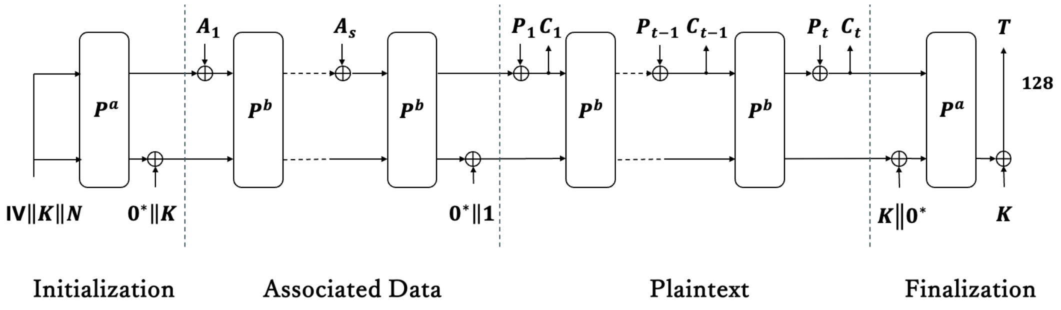
Table 1: Parameters for ASCON-AEAD algorithm

2.2.3 Process

ASCON encryption and decryption follow a structured process consisting of four main steps: initialization, processing associated data, processing plaintext/ciphertext, and finalization. Below, we describe each step in detail.



(a) Encryption



(b) Decryption

Figure 4: ASCON Operation

Initialization. The process begins by setting up the internal state using a **secret key** K, a **nonce** N, and an **initialization vector (IV)**. The IV is defined based on the key size k, the rate

r, the initialization and finalization round number a, and the intermediate round number b, each represented as an 8-bit integer. The IV is constructed as follows:

$$\mathrm{IV}_{k,r,a,b} \leftarrow k \parallel r \parallel a \parallel b \parallel 0^{160-k}$$

For different ASCON variants, the IV values are:

$$\mathrm{IV}_{k,r,a,b} = \begin{cases} 80400c0600000000000 & \text{for ASCON-128} \\ 80800c0800000000000 & \text{for ASCON-128a} \end{cases}$$

Using this IV, the initial 320-bit state S is formed by concatenating the IV with the secret key and nonce:

$$S \leftarrow \mathrm{IV}_{k,r,a,b} \parallel K \parallel N$$

Then, a rounds of the permutation function p are applied to the initial state, followed by an XOR operation with the secret key:

$$S \leftarrow p^a(S) \oplus (0^{320-k} \parallel K)$$

Processing Associated Data. The associated data (AD) is processed to ensure authentication while remaining unencrypted. This step is particularly useful for securing metadata in communication protocols.

ASCON handles AD by first padding it to a multiple of r bits by appending a 1 followed by the minimum required 0s. The padded data is then divided into s blocks of r bits:

$$A_1, \ldots, A_s \leftarrow \begin{cases} r\text{-bit blocks of } A \| 1 \| 0^{r-1-(|A| \bmod r)} & \text{if } |A| > 0 \\ \emptyset & \text{if } |A| = 0 \end{cases}$$

Each block A_i (for $i = 1, \ldots, s$) is XORed with the rate portion S_r of the internal state S, and the updated state undergoes the b-round permutation p^b:

$$S \leftarrow p^b((S_r \oplus A_i) \| S_c), \quad 1 \leq i \leq s$$

Finally, a domain separation bit is XORed into S to distinguish this phase from subsequent operations:

$$S \leftarrow S \oplus (0^{319} \| 1)$$

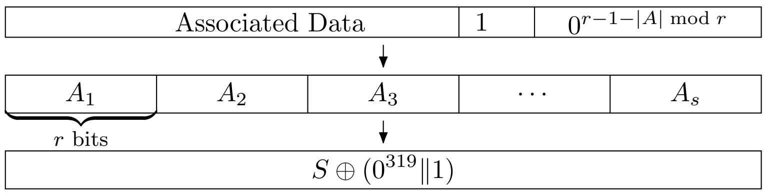


Figure 5: Processing Associated Data

Processing Plaintext/Ciphertext. The encryption and decryption processes involve splitting the plaintext or ciphertext into blocks of fixed size and padding them as necessary to match the required length. The padding process appends a single '1' bit followed by the minimum number of '0' bits such that the total length becomes a multiple of the block size r. The padded plaintext is represented as $P_1 \| \cdots \| P_t$, where each P_i is an r-bit block:

$$P_1, \ldots, P_t \leftarrow r\text{-bit blocks of } P \| 1 \| 0^{r-1-(|P| \mod r)}.$$

1. **Encryption.** In each iteration, a padded plaintext block P_i is XORed with the first r bits of the internal state S_r, producing the ciphertext block C_i:

$$C_i \leftarrow S_r \oplus P_i.$$

For $1 \leq i < t$, the internal state S is updated by applying a b-round permutation to the concatenated state $C_i \| S_c$:

$$S \leftarrow p^b(C_i \| S_c).$$

For the last block C_t, it is truncated to match the length of the unpadded plaintext's last fragment, ensuring the ciphertext length matches the plaintext:

$$\tilde{C}_t \leftarrow [C_t]_{|P| \mod r}.$$

2. **Decryption.** During decryption, each ciphertext block C_i is XORed with the first r bits of the internal state S_r to recover the plaintext block P_i:

$$P_i \leftarrow S_r \oplus C_i.$$

The internal state is updated similarly as in encryption, using the permutation p^b for $1 \leq i < t$:

$$S \leftarrow p^b(C_i \| S_c).$$

For the truncated last block $\tilde{C}_t$, the plaintext fragment is recovered by XORing it with the corresponding bits of S_r, followed by restoring the internal state:

$$\tilde{P}_t \leftarrow [S_r]_\ell \oplus \tilde{C}_t,$$

where $\ell = |P| \mod r$.

Finalization. In the finalization step, an authentication tag is generated to ensure data integrity and authenticity. The secret key K is XORed with the internal state S, and the state is transformed using a a-round permutation p^a. The tag T is derived from the last 128 bits of the updated state XORed with the last 128 bits of the key K:

$$S \leftarrow p^a(S \oplus (0^r \| K \| 0^{c-k})),$$
$$T \leftarrow [S]^{128} \oplus [K]^{128}.$$

The encryption algorithm outputs the tag T concatenated with the ciphertext $C_1 \| \cdots \| \tilde{C}_t$. During decryption, the plaintext $P_1 \| \cdots \| \tilde{P}_t$ is returned only if the calculated tag matches the received tag value.

2.3 Permutation

The ASCON permutation is a fundamental building block of the ASCON family, including ASCON-AEAD and ASCON-HASH. It is based on a sponge construction, which processes input data by absorbing it into an internal state and then producing output through a squeezing phase. The MonkeyDuplex mode extends this structure, allowing interleaved absorption and squeezing, making it particularly efficient for authenticated encryption and hashing applications.

The permutation function, denoted as p, operates on a 320-bit state split into five 64-bit words:

$$S = x_0 \parallel x_1 \parallel x_2 \parallel x_3 \parallel x_4$$

The transformation consists of three sequential steps:

$$p = p_L \circ p_S \circ p_C$$

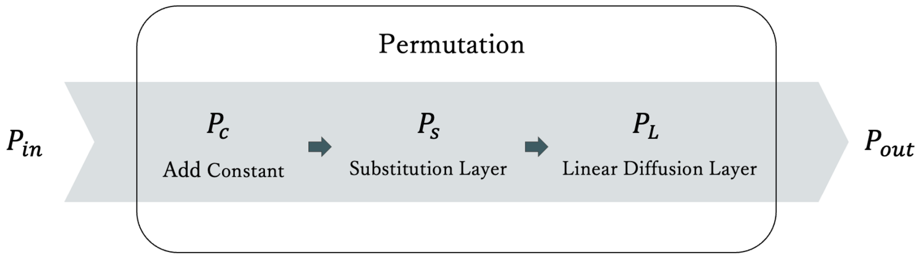


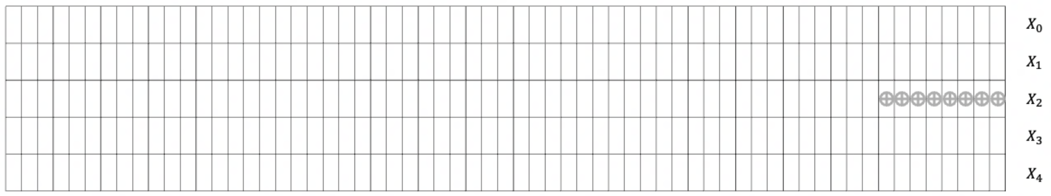
Figure 6: Permutation Process

2.3.1 Round Constant Addition

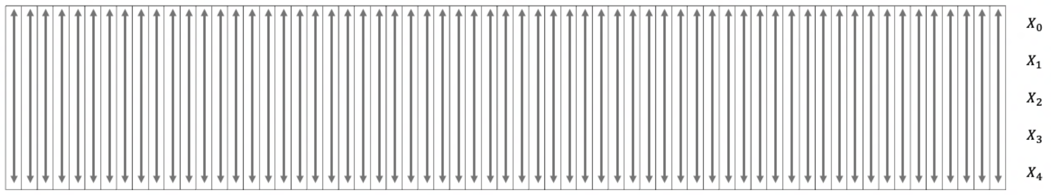
In this layer, only x_2 is changing. The round constant c_r is added to x_2 in each round i, as shown in Fig. 7. The indices r and i both start from zero, and the mapping follows:

$$x_2 \leftarrow x_2 \oplus c_r.$$

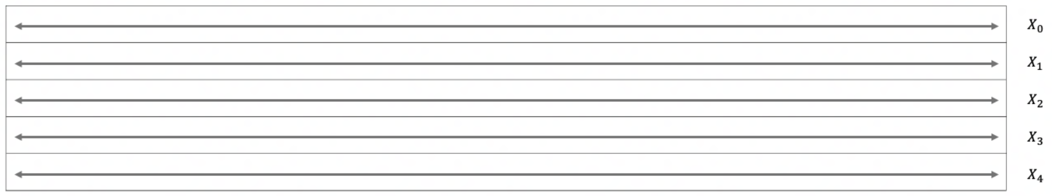
The values of the round constants c_r for different rounds are listed in Table 2.



(a) Round Constant Addition p_C



(b) Substitution Layer p_S



(c) Linear Diffusion Layer p_L

Figure 7: Permutation

p^{12}	p^8	p^6	**Constant** c_r	p^{12}	p^8	p^6	**Constant** c_r
0	-	-	`00000000000000f0`	6	2	0	`0000000000000096`
1	-	-	`00000000000000e1`	7	3	1	`0000000000000087`
2	-	-	`00000000000000d2`	8	4	2	`0000000000000078`
3	-	-	`00000000000000c3`	9	5	3	`0000000000000069`
4	0	-	`00000000000000b4`	10	6	4	`000000000000005a`
5	1	-	`00000000000000a5`	11	7	5	`000000000000004b`

Table 2: Round constants c_r used in the round constant addition step.

2.3.2 Substitution Layer

The substitution layer p_S updates the state S by applying the 5-bit S-box $S(x)$ to each bit-slice of the five registers x_0, x_1, x_2, x_3, x_4, as shown in Fig. 7. This transformation operates on the entire 64-bit words, allowing efficient implementation in a bitsliced manner. The S-box lookup table is given in Table 3, where x_0 is the most significant bit (MSB) and x_4 is the least significant bit (LSB).

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

Table 3: S-box $S(x)$ for Substitution Layer

2.3.3 Linear Diffusion Layer

The linear diffusion layer p_L enhances diffusion within each 64-bit register word x_i, as illustrated in Fig. 7.

$$\begin{aligned}
x_0 &\leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28) \\
x_1 &\leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39) \\
x_2 &\leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6) \\
x_3 &\leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17) \\
x_4 &\leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41)
\end{aligned}$$

Table 4: Linear Diffusion Layer

3 Hardware Implementation

3.1 Test Vector

To verify the correctness of this implementation, an official test vector was first sought. One common approach is to use a widely validated ASCON Python implementation (3) as a reliable reference.

However, this Python implementation differs from the paper in several aspects. For instance, the initialization vector (IV) in (1) is different from the standardized version proposed by NIST (2). Additionally, there may be differences in bitwise operations and other details. These differences indicate that the Python implementation follows a different methodology than the referenced paper, potentially leading to variations in test results.

3.2 Block Diagram

The hardware implementation consists of two main modules: the encryption module and the permutation module. The permutation module acts as a submodule within the encryption module, performing repeated transformations essential for the encryption process.

3.2.1 Encryption

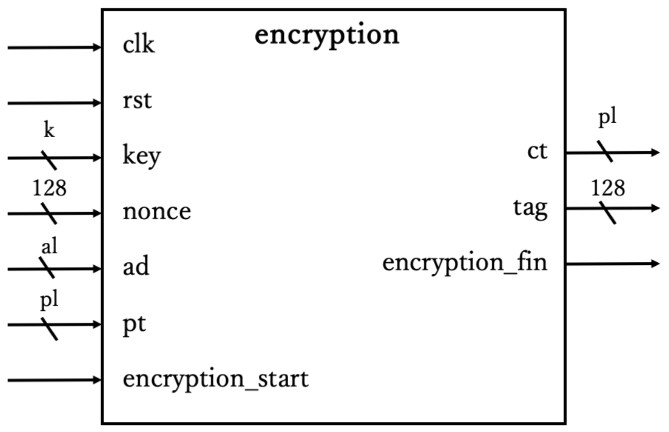


Figure 8: Encryption Block Diagram

Signal	I/O	Bit Width	Description
clk	I	1	Positive edge-triggered clock signal
rst	I	1	Active-high reset signal
key	I	k	Encryption key
nonce	I	128	Nonce
ad	I	al	Associated data
pt	I	pl	Plaintext
encryption_start	I	1	Encryption start signal
ct	O	pl	Ciphertext
tag	O	128	Authentication tag
encryption_fin	O	1	Encryption done signal

Table 5: Encryption Module Signal Definitions

3.2.2 Permutation

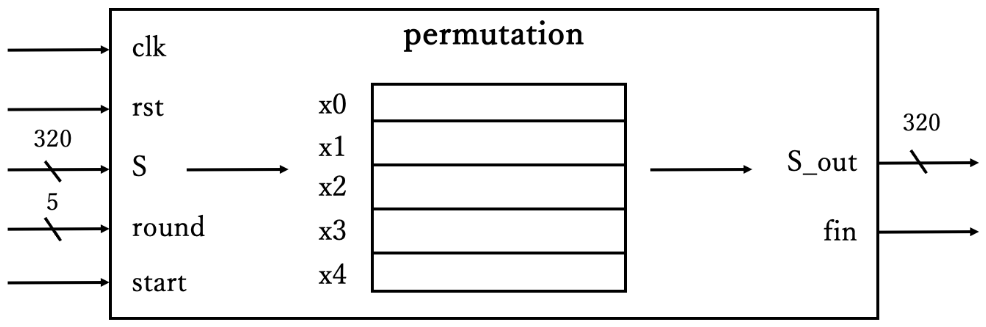


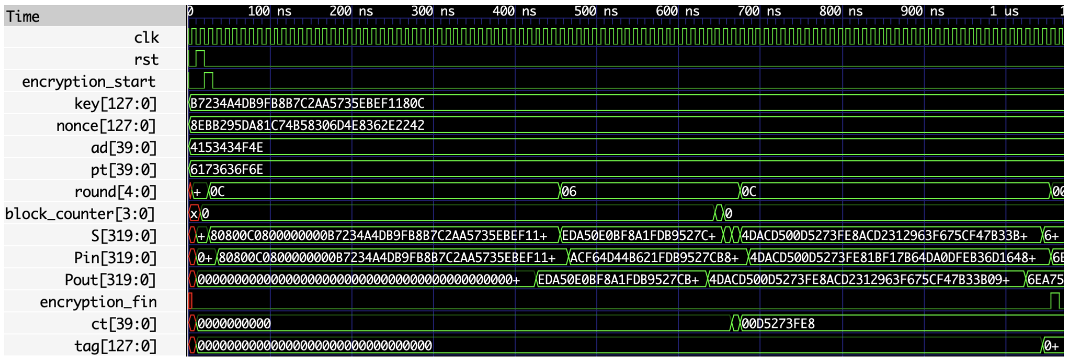
Figure 9: Permutation Block Diagram

Signal	I/O	Bit Width	Description
clk	I	1	Positive edge-triggered clock signal
rst	I	1	Active-high reset signal
S	I	320	State register
round	I	5	Round counter
start	I	1	Permutation start signal
S_out	O	320	State register
fin	O	1	Permutation done signal

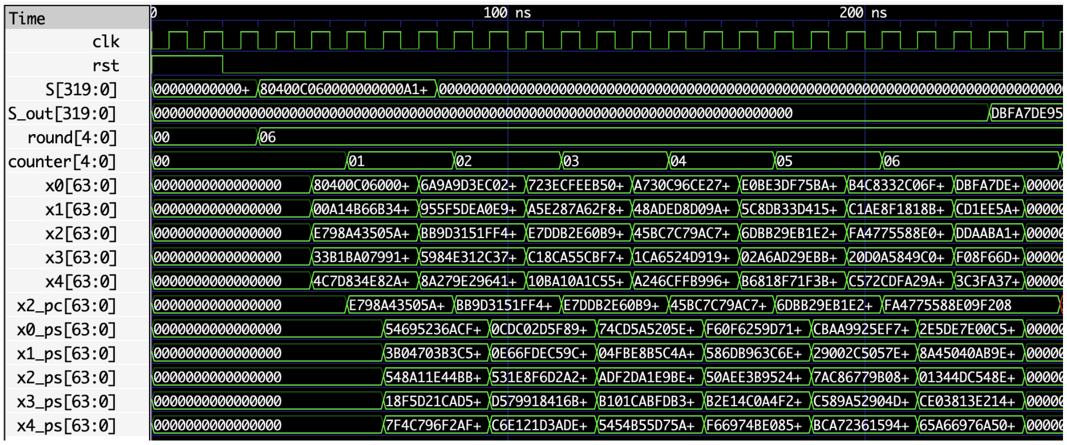
Table 6: Permutation Module Signal Definitions

3.3 Synthesis

The code was implemented using Icarus Verilog and simulated with GTKWave. The simulation verified the hardware design, including the encryption processes and permutation. Below is an example of a waveform from the simulation, showcasing its operation.



(a) Encryption


Time
clk
rst
S[319:0]
S_out[319:0]
round[4:0]
counter[4:0]
x0[63:0]
x1[63:0]
x2[63:0]
x3[63:0]
x4[63:0]
x2_pc[63:0]
x0_ps[63:0]
x1_ps[63:0]
x2_ps[63:0]
x3_ps[63:0]
x4_ps[63:0]
100 ns
200 ns
80400C060000000000A1+
DBFA7DE95
80400C06000+ 6A9A9D3EC02+ 723ECFEEB50+ A730C96CE27+ E0BE3DF75BA+ B4C8332C06F+ DBFA7DE+
00A14B66B34+ 955F5DEA0E9+ A5E287A62F8+ 48ADED8D09A+ 5C8DB33D415+ C1AE8F1818B+ CD1EE5A+
E798A43505A+ BB9D3151FF4+ E7DDB2E60B9+ 45BC7C79AC7+ 6DBB29EB1E2+ FA4775588E0+ DDAABA1+
33B1BA07991+ 5984E312C37+ C18CA55CBF7+ 1CA6524D919+ 02A6AD29EBB+ 20D0A5849C0+ F08F66D+
4C7D834E82A+ 8A279E29641+ 10BA10A1C55+ A246CFFB996+ B6818F71F3B+ C572CDFA29A+ 3C3FA37+
E798A43505A+ BB9D3151FF4+ E7DDB2E60B9+ 45BC7C79AC7+ 6DBB29EB1E2+ FA4775588E09F208
54695236ACF+ 0CDC02D5F89+ 74CD5A5205E+ F60F6259D71+ CBAA9925EF7+ 2E5DE7E00C5+
3B04703B3C5+ 0E66FDEC59C+ 04FBE8B5C4A+ 586DB963C6E+ 29002C5057E+ 8A45040AB9E+
548A11E44BB+ 531E8F6D2A2+ ADF2DA1E9BE+ 50AEE3B9524+ 7AC86779B08+ 01344DC548E+
18F5D21CAD5+ D579918416B+ B101CABFDB3+ B2E14C0A4F2+ C589A52904D+ CE03813E214+
7F4C796F2AF+ C6E121D3ADE+ 5454B55D75A+ F66974BE085+ BCA72361594+ 65A66976A50+

(b) Permutation

Figure 10: Waveform Generated by GTKWave

4 Impact and Future Work

ASCON's lightweight design allows for lower power consumption and faster processing when implemented in hardware. This makes it ideal for battery-powered devices, such as wearable technology and sensors, which contribute to sustainable and energy-efficient security solutions.

Future work will focus on implementing the verified ASCON algorithm on FPGA platforms to further optimize its performance and resource utilization. Although ASCON theoretically resists side-channel attacks at the algorithmic level, practical hardware implementations require additional safeguards to address vulnerabilities arising from power consumption and electromagnetic leakage.

Recent research by (4) proposes a shared hardware architecture to mitigate side channel attack vulnerabilities in ASCON implementations, demonstrating promising results through the concept of sharing. Building on this work, future efforts could explore hybrid approaches that combine algorithmic optimizations with physical layer countermeasures to enhance robustness against advanced side-channel and fault injection attacks.

References

[1] C. Dobraunig, M. Eichlseder, F. Mendel, and M. Schläffer, "Ascon v1.2: Lightweight Authenticated Encryption and Hashing," Journal of Cryptology, vol. 34, no. 3, Jun. 2021. doi: `https://doi.org/10.1007/s00145-021-09398-9`.

[2] S. Meltem, K. Turan, D. Mckay, J. Chang, J. Kang, and J. Kelsey, "NIST Special Publication 800-232 ipd: Ascon-Based Lightweight Cryptography Standards for Constrained Devices – Authenticated Encryption, Hash, and Extendable Output Functions (Initial Public Draft)," NIST SP 800-232 ipd. doi: `https://doi.org/10.6028/NIST.SP.800-232.ipd`.

[3] M. Eichlseder, "Python implementation of Ascon," GitHub, Apr. 2, 2023. [Online]. Available: `https://github.com/meichlseder/pyascon`

[4] A. Kandi, A. Baksi, P. Gan, S. Guilley, T. Gerlich, J. Breier, A. Chattopadhyay, R. R. Shrivastwa, Z. Martinásek, and S. Bhasin, "Side-Channel and Fault Resistant ASCON Implementation: A Detailed Hardware Evaluation (Extended Version)," *Cryptology ePrint Archive, Paper 2024/984*, 2024. [Online]. Available: `https://eprint.iacr.org/2024/984`

AI Usage Disclaimer

In this study, I used ChatGPT by OpenAI to refine wording and correct grammar. However, the logic, structure, and overall narrative were entirely written by me, HSU TING YU. Additionally, all figures and diagrams were created by me using PowerPoint and Overleaf's built-in drawing tools. Some algorithm-related diagrams were inspired by other papers, but I adapted them based on a full understanding of the concepts and tailored them to fit the needs of this research. I affirm that this paper **adheres to AI ethical guidelines** and is held to the academic standards.