ASCON-Based Lightweight Cryptography Standards for Constrained Devices

Authenticated Encryption, Hash and Extendable Output Functions

Michelle Lau
11917662

A paper presented for a Seminar to Security in Software Engineering and Internet Computing

Institute of Information Systems Engineering
University of Technology Vienna
Austria
July 20, 2025

Abstract

Cryptography plays a crucial role in today's technology. Not only does it have to be secure, but also efficient so that devices with limited resources - such as IoT devices - can utilize and implement secure cryptographic algorithms. The family of Ascon algorithms, designed by Dobraunig, Eichlseder, Mendel and Schläffer is a new standard chosen in the NIST Lightweight Cryptography competition (2019–2023) [nist17]. The NIST special publication [asc24], which this paper is based on, includes Authenticated Encryption, Hash, as well as extendable output functions.

1 Introduction

The importance of cryptography is vital in almost all digital ecosystems. Without it, maintaining data security and privacy would not be possible in today's digital world. The practice of cryptography enables different ways of securing communication by encrypting and decrypting data with their corresponding algorithm. There exist many conventional standards of cryptographic algorithms (such as the Secure Hash Standard *SHS* [shs15] and the Advanced Encryption Standard *AES* [aes01]); however, not all of them consider the limited capacity of constrained devices - like Internet of Things (IoT) devices, embedded systems, or low-power sensors.
As a viable alternative for such devices, the Ascon family provides a set of algorithms that use symmetric cryptography and lightweight permutations to achieve secure ciphering of information even in resource-limited environments.
The family introduces new standards including the Authenticated Encryption with Associated Data `Ascon-AEAD128`, the hash function `Ascon-Hash256`, Extendable Output Function (XOF) `Ascon-XOF128` and the customized version of the Extendable Output Function (CXOF) `Ascon-CXOF128`:

1. `Ascon-AEAD128`: This nonce-based algorithm provides a lightweight solution to ciphering a message with associated data and provides 128-bit security in the single-key setting.

2. `Ascon-Hash256`: This hash function converts an input message into a 256-bit digest with a security of 128 bits.

3. `Ascon-XOF128`: This XOF is similar to the Ascon-Hash256 function, with the difference that the output size of the digest can be customized by the user.

4. `Ascon-CXOF128`: This CXOF provides further customization by taking a random input string chosen by the user to be used in the computation of the custom output length.

The family of Ascon algorithms utilizes a symmetric-key cryptography scheme, which means that the same key is used in encryption as well as decryption. Furthermore, a single lightweight permutation using sponge/duplex [spng12] constructions is used, enabling an efficient and simple implementation of the algorithms.

The main features of Ascon include:

- **Multiple functionalities**: All introduced algorithms (`Ascon-AEAD128`, `Ascon-Hash128`, `Ascon-XOF128` and `Ascon-CXOF128`) of the Ascon family utilize the same permutation, which allows a more unified, consistent and compact implementation.
- **Online and single pass**: Ascon is able to start ciphering data immediately, without needing to know the full size of the input in advance. It also only needs to go through the data once to complete encryption and decryption, since there is no need for multiple rounds of reading or buffering. This ensures that the algorithms stay efficient, lightweight and secure.
- **Inverse-free**: Since no inverse permutations are necessary (unlike other cryptographic algorithms like AES that uses inverse cipher [aes01]), implementation costs are reduced significantly compared to other algorithms that require inverse operations for decryption.

These features highlight Ascon's lightweight design and its role in establishing cryptographic standards for constrained devices.

2 Ascon Permutations

As already mentioned in 1, Ascon utilizes lightweight permutations as a core cryptographic function to process the internal state (explained in 2.1). This means that the bits of the internal state (that comprise the message input amongst other data) get shuffled around through multiple rounds, ensuring that each input produces a unique output. This way, it is difficult for any potential adversary to reverse the permuted state.
Ascon permutations act as the main components of the ciphering and hashing algorithms and are iteratively applied rnd-times depending on the algorithm used. They are specified as rnd-round *Ascon-p[rnd]* permutations, where $1 \leq rnd \leq 16$. This means that *Ascon-p[8]* would indicate a permutation of 8 rounds, so the input state gets permuted 8 times. Note that higher rounds can be used to ensure more secure ciphering. Additionally, *Ason-p[8]* and *Ason-p[12]* are exclusively used for the introduced algorithms; however, permutations defined with other numbers may be used to standardize other functionalities.
In Ascon, a permutation is a transformation that is applied to a 320-bit long internal state. The transformation itself consists of three different layers. Each round of the permutation transforms the state as follows:

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

where p_L is the constant-addition layer, p_S is the substitution layer and p_C is the diffusion layer. This means that given an *Ascon-p[8]* permutation as an example, all three layers (constant-addition, substitution and diffusion) are applied iteratively 8 times.

2.1 Internal State

As previously stated, the permutations operate on an internal state. This 320-bit state is divided into five concatenated 64-bit long words, denoted as S_i for $0 \leq i \leq 4$, making up the final state S:

$$S = S_0 \parallel S_1 \parallel S_2 \parallel S_3 \parallel S_4.$$

This means that the Ascon algorithm will apply three layers of transformation (constant-addition, substitution, and diffusion) to the internal state S throughout a single round of permutation, up to 16 rounds, causing the individual words and bits to be shuffled around.
Each Ascon algorithm initializes the internal state differently and will be discussed in further detail in their respective specifications.

2.2 Constant-Addition Layer pC

The Constant-Additional Layer is the first part of the transformation the internal state goes through in a permutation. It aims to break the symmetry of the internal state, allowing unpredictability and therefore better security. This layer takes a constant value c_i and shuffles it into the internal state using the XOR ($\oplus$) operator. The pre-determined constant value is different for each round i of the Ascon permutation *Ascon-p[rnd]*, with $0 \leq i \leq rnd - 1$ and is defined as

$$c_i = const_{16-rnd+i}.$$

The constant variables $const_0, \ldots, const_{15}$ are defined in table 1.

i	$const_i$	i	$const_i$
0	0x000000000000003c	8	0x00000000000000b4
1	0x000000000000002d	9	0x00000000000000a5
2	0x000000000000001e	10	0x0000000000000096
3	0x000000000000000f	11	0x0000000000000087
4	0x00000000000000f0	12	0x0000000000000078
5	0x00000000000000e1	13	0x0000000000000069
6	0x00000000000000d2	14	0x000000000000005a
7	0x00000000000000c3	15	0x000000000000004b

Table 1: The constants $const_i$ to derive round constants of the Ascon permutations

The constant is incorporated into the internal state, however only the word S_2 gets modified:

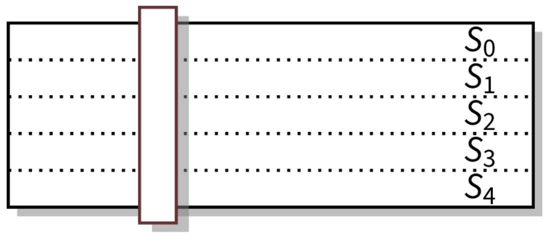
$$S_2 = S_2 \oplus c_i.$$

This means that from the whole internal state S comprising of five 64-bit words, only one singular word S_2 undergoes transformation in this step.
Note that for each constant, the first 56 bits are zero, which means that only the least significant 8 bits (consequently, only one byte) are applied to S_2.

2.3 Substitution Layer pS

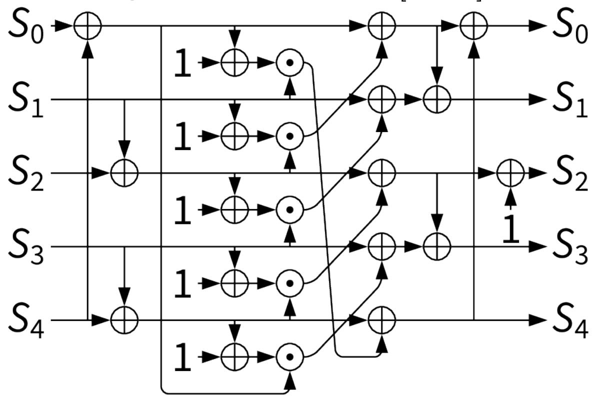
Following the constant-layer transformation, a substitution layer is applied to further modify the internal state. The introduction of this transformation ensures nonlinearity, which contributes to the resistance of cryptanalytic attacks like differential or linear cryptanalysis [crat94]. In this layer, a 5-bit S-Box transformation is applied to each word 64 times in parallel in a bit-sliced fashion as shown in figure 1. A bit-slice is defined as the collection of bits at the same position across all five words of the internal state. This results in a modification of each vertical slice, containing one bit from each word of the state S.

Figure 1: 5-bit S-Box [res23]



The S-Box transformation itself that is applied to each slice is depicted in figure 2. Each slice is treated as a 5-bit input to the S-Box and maps it to a new 5-bit output. The same S-Box is applied in parallel to all 64 slices, resulting in a bit-sliced implementation. The exact transformation is fixed and optimized for lightweight cryptography, a key design point in the Ascon family.

Figure 2: 5-bit S-Box [res23]



To note is that the S-Box logic involves only AND ($\odot$), XOR ($\oplus$) and NOT operations, and it is designed to be bitsliced and efficient for both hardware and software. In hardware, the S-Box may also be implemented with a lookup table (since there are only $2^5 = 32$ possible input variations), though it requires consideration of area cost and side-channel vulnerability. Table 2 represents the lookup table of S-Box, where the 5-bit inputs are represented in hexadecimal numbers.

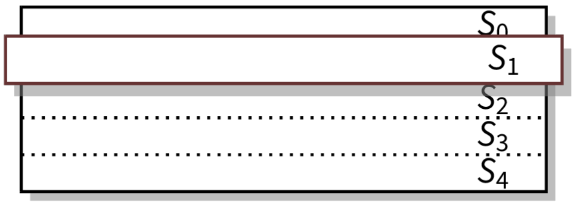
x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
SBox(x)	4	b	if	14	1a	15	9	2	1b	5	8	12	1d	3	6	1c
x	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
SBox(x)	1e	13	7	e	0	d	11	18	10	c	1	19	16	a	f	17

Table 2: S-box values used in the Ascon permutation

2.4 Linear Diffusion Layer pL

Finally, a linear diffusion is applied to the internal state as the last step of the Ascon permutation. It's the final transformation of a round and is responsible for spreading the influence of each bit across the state. Through this transformation, even a slight change in the input ensures a widespread alteration of the output bits. This naturally enhances the cipher's resistance to unwanted attacks.
The Linear Diffusion Layer mixes the bits together in a horizontal fashion, which means that the transformation happens within each 64-bit word, like depicted in figure 3.

Figure 3: Linear Diffusion Layer p_L [res23]



The diffusion layer applies a combination of bitwise rotations ($\ggg$) and XOR operations to each word:

$$\Sigma_0(S_0) = S_0 \oplus (S_0 \ggg 19) \oplus (S_0 \ggg 28) \tag{1}$$
$$\Sigma_1(S_1) = S_1 \oplus (S_1 \ggg 61) \oplus (S_1 \ggg 39) \tag{2}$$
$$\Sigma_2(S_2) = S_2 \oplus (S_2 \ggg 1) \oplus (S_2 \ggg 6) \tag{3}$$
$$\Sigma_3(S_3) = S_3 \oplus (S_3 \ggg 10) \oplus (S_3 \ggg 17) \tag{4}$$
$$\Sigma_4(S_4) = S_4 \oplus (S_4 \ggg 7) \oplus (S_4 \ggg 41), \tag{5}$$

where the function Σ_i is applied to their corresponding words as $S_i \leftarrow \Sigma_i(i)$, where $0 \le i \le 4$. The Linear Diffusion Layer acts as the final step of the state transformation and concludes a permutation round.
To conclude, the three layers of the permutation ensure that the internal state stays resistant to cryptanalytic attacks. This transformation is paramount in Ascon's lightweight and secure design for ciphering, authentication, and hashing.

3 Authenticated Encryption Scheme: `Ascon-AEAD128`

The Authenticated Encryption Scheme: Ascon-AEAD128 is a lightweight Authenticated Encryption with Associated Data, designed for environments with limited resources like IoT devices. This section will elaborate on the specification of this scheme in a detailed manner.

3.1 Specification of Ascon-AEAD128

Ascon-AEAD128 is responsible for encrypting and decrypting data and therefore consists of two algorithms - one for encryption (`Ascon-AEAD128.enc` 3.2) and one for decryption. (`Ascon-AEAD128.dec` 3.3).
Even though Ascon-AEAD128 consists of two algorithms, the processes of encryption and decryption are very similar in design and use the same permutation.

3.2 Encryption

`Ascon-AEAD128.enc` is responsible for encrypting plaintext to ensure data security by converting readable data into an unreadable format (ciphertext). It also offers authentication by verifying the identity of the sender using a cryptographic tag and associated data support. Note that associated data is mainly used to provide integrity protection for non-sensitive metadata (like headers or addresses) that is not encrypted but still needs to be authenticated.
The following describes an `Ascon-AEAD128.enc` function:

$$\texttt{Ascon-AEAD128.enc}(K, N, A, P) = (C, T),$$

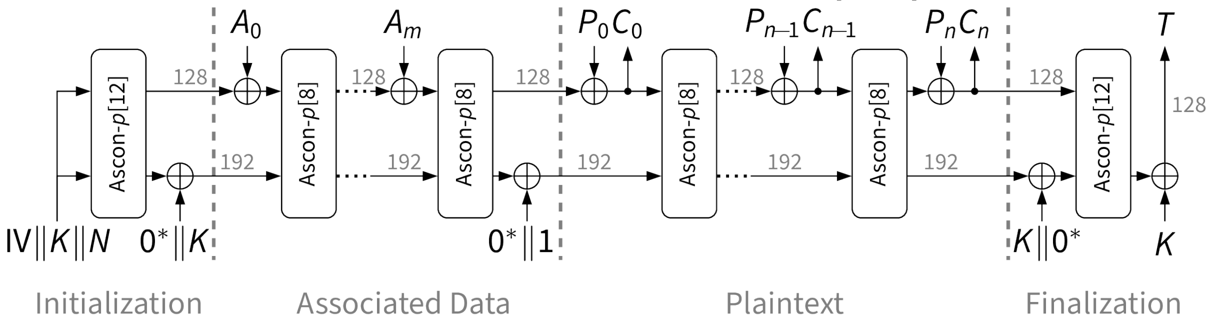
where K is a 128-bit secret key, N is a 128-bit nonce, A is a variable-length associated data and P is a variable-length plaintext. The function outputs the ciphertext C (where $|C| = |P|$) and a 128-bit long authentication tag T.

`Ascon-AEAD128.enc` consists of four main phases:

- **Initialization**: Initialization of the internal state with input key K and nonce N
- **Associated Data Processing**: Absorption of the associated data A into the internal state
- **Plaintext Processing**: Encryption of plaintext into ciphertext C
- **Finalization**: Reinjection of the key K and extraction of an authentication tag T

The encryption process is depicted in figure 4 and showcases the four different phases. Important to note is that the Associated Data A, Plaintext P, and Ciphertext C are divided into different blocks for processing, therefore indicated as $A_0 \dots A_m$, $P_0 \dots P_n$ and $C_0 \dots C_n$ in the figure.
Furthermore, two core functions, namely $parse(X, r)$ and $pad(X, r)$, are used to parse bitstrings into blocks and then padded to a desirable length. These functions are described in Appendix F and will be used frequently.

Figure 4: `Ascon-AEAD128.enc` encryption [res23]



The pseudocode for the whole encryption process (referenced from the original Ascon specification pseudocode [asc24]) is provided in algorithm 1 and will be discussed in detail.

Algorithm 1 `Ascon-AEAD128.enc`(K, N, A, P)

Input: 128-bit key K; 128-bit nonce N; Associated data A; Plaintext P
Output: Ciphertext C; 128-bit tag

```
 1: $IV \leftarrow$ 0x00001000808c0001                         ▷ Initialization
 2: $S \leftarrow IV \| K \| N$
 3: $S \leftarrow$ Ascon-p[12]$(S)$
 4: $S \leftarrow S \oplus (0^{192} \| K)$

 5: if $|A| > 0$ then                                        ▷ Processing Associated Data
 6:     $A_0, \ldots, A_{m-1}, \widetilde{A_m} \leftarrow$ parse$(A, 128)$
 7:     $A_m \leftarrow$ pad$(\widetilde{A_m}, 128)$
 8:     for $i = 0$ to $m$ do
 9:         $S \leftarrow$ Ascon-p[8]$((S_{[0:127]} \oplus A_i) \| S_{[128:319]})$
10:     end for
11: end if
12: $S \leftarrow S \oplus (0^{319} \| 1)$

13: $P_0, \ldots, P_{n-1}, \widetilde{P_n} \leftarrow$ parse$(P, 128)$        ▷ Processing Plaintext
14: $\ell \leftarrow |\widetilde{P_n}|$
15: for $i = 0$ to $n - 1$ do
16:     $S_{[0:127]} \leftarrow S_{[0:127]} \oplus P_i$
17:     $C_i \leftarrow S_{[0:127]}$
18:     $S \leftarrow$ Ascon-p[8]$(S)$
19: end for
20: $S_{[0:127]} \leftarrow S_{[0:127]} \oplus$ pad$(\widetilde{P_n}, 128)$
21: $\widetilde{C_n} \leftarrow S_{[0:\ell-1]}$
22: $C \leftarrow C_0 \| \ldots \| C_{n-1} \| \widetilde{C_n}$

23: $S \leftarrow$ Ascon-p[12]$(S \oplus (0^{128} \| K \| 0^{64}))$          ▷ Finalization
24: $T \leftarrow S_{[192:319]} \oplus K$
25: return $C, T$
```

As stated previously, the encryption process consists of four phases: Initialization 3.2.1, Associated Data Processing 3.2.2, Plaintext Processing 3.2.3 and Finalization 3.2.4. Each phase will be described in detail in the following sections, with reference to the official Ascon specification pseudocode 1.

3.2.1 Initialization

The encryption process starts with the initialization phase. This phase sets up the internal state of the Ascon cipher before any data is processed.
First, a fixed 64-bit initial value IV (refer to Appendix G for the values) is defined, before concatenating it with the 128-bit secret key and the 128-bit public nonce, making up the initial internal state:

$$S \leftarrow IV \| K \| N.$$

This state is then permuted using the *Ascon-p[12]* permutation, which applies 12 rounds of cryptographic transformation to ensure diffusion and non-linearity as explained in 2:

$$S \leftarrow \texttt{Ascon-p}[12](S).$$

Note that only in the initialization and finalization phases, a 12-round permutation *Ascon-p[12]* is used. All the other phases use an eight-round permutation *Ascon-p[8]* in this algorithm.
After the 12 round permutation, the input key K is then incorporated into the internal state by XORing it with the last 128 bits of S. Since the state contains 320 bits and the key is only 128 bits long, the remaining 192 bits are first concatenated into the state as zeros:

$$S \leftarrow S \oplus (0^{192} \| K)$$

The above step ensures that the secret key is well integrated into the state and therefore helps prevent unwanted malicious attacks. This concludes the initialization phase of the encryption process. As a result, the internal state is securely initialized and ready for processing the associated data and plaintext.

3.2.2 Associated Data Processing

After the initialization phase, the input associated data A is processed by injecting it into the cipher's internal state. This step absorbs the associated data into the state for authentication during the finalization phase in 3.2.4; however, no encryption is taking place. This stage ensures that any change in A will be detected by the verification process, even though no ciphertext is produced.
The following section will explain in detail the processing of associated data A with references to the official Ascon specification pseudocode 1.

First of all, the associated data plaintext needs to be prepared for processing, since the Ascon specification operates block-wise. The functions $\texttt{parse}(X,\ r)$ and $\texttt{pad}(X,\ r)$ will be used to process the associated data A in blocks of r bits (where $r = 128$ in Ascon-AEAD128 as indicated in $\texttt{parse}(A, 128)$). If the length of A is not a multiple of r, it is then padded by appending a single 1 bit followed by the minimum number of 0 bits needed to reach a full 128-bit block:

$$A_0, \dots, A_{m-1}, \widetilde{A_m} \leftarrow \texttt{parse}(A, 128)$$
$$A_m \leftarrow \texttt{pad}(\widetilde{A_m}, 128)$$

If A is empty, no associated data blocks are processed, and the padding step is skipped entirely.
After the processing of the plaintext, each block $A_0, \dots, A_m$ gets absorbed into the cipher's internal state. This is done by XORing each block A_i (in a loop) into the first 128 bits of the state:

$$((S_{[0:127]} \oplus A_i) \| S_{[128:319]}),$$

The updated state is formed by XORing A_i into the first 128 bits of the state, while the remaining bits are left unchanged. After incorporating the associated data block, the algorithm applies the Ascon permutation with 8 rounds to the entire state:

$$S \leftarrow \texttt{Ascon-p}[8]((S_{[0:127]} \oplus A_i) \| S_{[128:319]}).$$

This approach of permuting after each block gets absorbed, mixes the associated data into all parts of the state before the next block is processed. These two steps (XOR then permutation) are repeated for each AD block in sequence. Notably, this phase does not produce any output — it mainly makes sure that the associated data influences the state (and later authentication tag) but does not itself appear in the ciphertext.
After all AD blocks are processed, a domain separation constant is XORed into the state to mark the transition into the next phase: Plaintext Processing 3.2.3. In this algorithm, the constant is indicated by a single bit (1 bit with 319 zeros) and will be XORed into the state:

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

Notably, this domain separation step is essential to distinguish the AD phase from the plaintext processing phase, by ensuring that there is no ambiguity in how data is interpreted during processing.

3.2.3 Plaintext Processing

The third stage of the encryption algorithm is the plaintext processing. The purpose of this phase is to encrypt the input plaintext message P and produce the output ciphertext C, while ensuring that the message is also authenticated (since its influence propagates into the finalization stage 3.2.4 used to generate the tag). Similar to the associated data processing 3.2.2, blocks of plaintext are absorbed into the state. This phase uses a duplex sponge construction, where each plaintext block is first XORed into the state (absorbed), and the updated state immediately produces the corresponding ciphertext block (squeezed). Since this process of absorbing input and then squeezing output resembles the usage of a sponge, it is referred to as a duplex sponge construction.
The following section provides a detailed explanation of how the plaintext input P is processed to produce the ciphertext C output.

Similar to the previous stage, the plaintext P is split into different blocks for processing. This divides the input P into a sequence of 128-bit blocks $P_0 \dots \widetilde{P_n}$. This ensures that each block can be processed uniformly by the sponge:

$$P_0, \dots, P_{n-1}, \widetilde{P_n} \leftarrow \texttt{parse}(P, 128)$$

The variable assignment of ℓ in line 14 as part of pseudocode 1 computes the bit-length of the final plaintext block and has a variable length $0 \leq \ell \leq 128$. It will be used to extract exactly ℓ bits of ciphertext from the state for the last cipher block, since the encryption function specifies that $|C| = |P|$:

$$\ell \leftarrow |\widetilde{P_n}|$$

After the plaintext is processed into different blocks, the algorithm starts processing each full block (except for the last block $\widetilde{P_n}$; it will be processed after the loop) by looping over them. Each block gets absorbed into the state and produces a corresponding cipher block C_i.
The absorption takes the plaintext block P_i and XORs it into the first 128 bits of the internal state. This means that the old state and P_i are combined so that the state now depends on the plaintext block. Note that the remaining 192 bits remain as before:

$$S_{[0:127]} \leftarrow S_{[0:127]} \oplus P_i$$

After the absorption step, a ciphertext block C_i is immediately produced (”squeezed out”) from the first 128 bits of the state. This means that the portion of the state mixed with P_i becomes part of the ciphertext output:

$$C_i \leftarrow S_{[0:127]}$$

The internal state S is then permuted with an 8-round Ascon permutation. This step provides diffusion and scrambles S, so that the effects of the plaintext block P_i and every previous block are spread throughout the state. The permutation also prepares the state for the following block's absorption and prevents patterns, since between every block absorption, `Ascon-p[8]` is applied.

$$S \leftarrow \texttt{Ascon-p}[8](S).$$

The sponge mechanism of absorbing and squeezing blocks concludes after the loop has iterated through every plaintext block (apart from $\widetilde{P_n}$).
After the iteration, the remaining partial block $\widetilde{P_n}$ is then padded and absorbed into the state. The padding is essential, as the last block does not necessarily have a length of 128 bits but instead a variable length ℓ, which means that the block needs to be padded to 128 bits. ($\widetilde{P_n}$ becomes $\widetilde{P_n} \,\|\, 1 \| \, 0 \ldots 0$ with 128 bits in length after the padding). This padded block is then XORed into the first 128 bits of S:

$$S_{[0:127]} \leftarrow S_{[0:127]} \oplus \texttt{pad}(\widetilde{P_n}, 128)$$

After the final bits have been absorbed into the state, the corresponding cipher block needs to be squeezed out. In this case, the first ℓ bits of the state are output as the last ciphertext segment $\widetilde{C_n}$. This “squeezes” the exact number of bits corresponding to $\widetilde{P_n}$ (if $\ell = 0$, this step produces no bits):

$$\widetilde{C_n} \leftarrow S_{[0:\ell-1]}$$

Finally, all the ciphertext blocks are concatenated into the full ciphertext output C:

$$C \leftarrow C_0 \| \ldots \| C_{n-1} \| \widetilde{C_n}$$

At this point, the whole plaintext input has been completely encrypted and the final digest constructed. The internal state holds all absorbed data, which will be used in the next step: Finalization 3.2.4

3.2.4 Finalization

After the plaintext has been absorbed and the ciphertext created, the algorithm enters the finalization stage to produce a 128-bit authentication tag T. The authentication tag ensures the integrity of both the plaintext as well as the associated data input during the previous stages. It verifies that the message has not been tampered with and is therefore paramount for secure communication. It summarizes all data that has been absorbed into the state (Nonce and Key during the initialization phase, Associated Data during AD Processing, and Plaintext input during Plaintext Processing) and produces a unique 128-bit long authentication tag.

During the decryption 3.3 process, the recipient can then ensure that the message is valid by comparing their tag (created in the decryption process) with the one produced during the encryption phase.
The finalization stage starts by absorbing the secret key K back into the state and then applying a 12-round Ascon permutation:

$$S \leftarrow \texttt{Ascon-p}[12](S \oplus (0^{128} \parallel K \parallel 0^{64}))$$

The notation $S \oplus (0^{128} \| K \| 0^{64})$ indicates that the first 128 bits of the XOR mask are zeros, the next 128 bits are the key K, and the last 64 bits are zeros, making up 320 bits in total for the entire internal state. This means that the key is injected into the middle of the state, while the rest stays the same. The permutation that follows diffuses the entire state further and provides strong mixing before extracting the authentication tag T:

$$T \leftarrow S_{[192:319]} \oplus K$$

Noticeably, the authentication tag is extracted by taking only the last 128 bits of the state and XORing them with the key K. Here, $S_{[192:319]}$ denotes the bits from index 192 until 319 of the state – which correspond to the last two 64-bit words of S: S_4 and S_5. By incorporating the secret key directly into the authentication tag ensures that even if an attacker sees the tag and ciphertext, they cannot predict nor forge tags without the key.
Finally, the ciphertext and authentication tag are returned:

$$\textbf{return}\, C, T$$

This completes the encryption algorithm `Ascon-AEAD128.enc`.

3.3 Decryption

`Ascon-AEAD128.dec` is responsible for decrypting ciphertext by converting it back to plaintext. Similar to the encryption algorithm `Ascon-AEAD128.enc` 3.2, this algorithm also offers authentication by verifying the identity of the sender using a cryptographic tag.
The following describes an `Ascon-AEAD128.dec` function:

$$\texttt{Ascon-AEAD128.dec}(K, N, A, C, T) = \begin{cases} P & \text{if the tag } T \text{ is valid} \\ \text{fail} & \text{otherwise} \end{cases}$$

where K is a 128-bit secret key, N is a 128-bit nonce, A is a variable length associated data, C is a variable-length ciphertext and T the authentication tag used to verify data integrity. The function outputs the plaintext P if its own produced 128-bit authentication tag matches the input authentication tag T; otherwise, the algorithm fails in case the tags do not match and therefore no integrity is guaranteed.

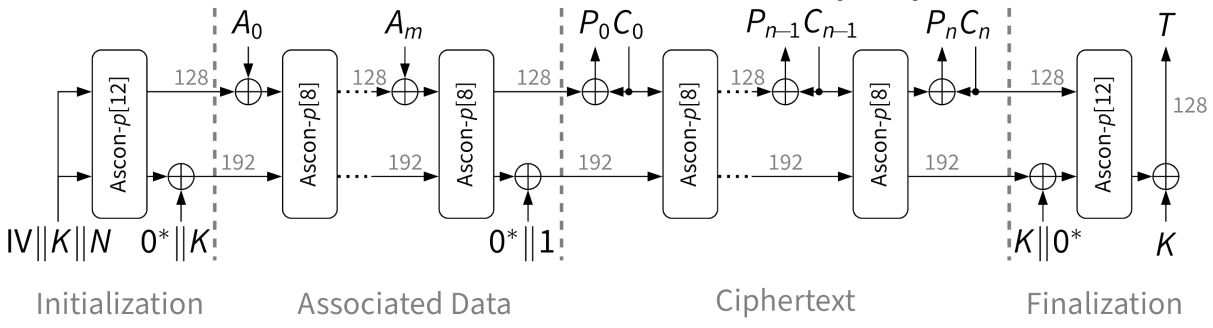
`Ascon-AEAD128.dec` consists of four main phases:

- **Initialization**: Initialization of the internal state with input key K and nonce N

- **Associated Data Processing**: Absorption of the associated data A into the internal state
- **Ciphertext Processing**: Decryption of ciphertext C into plaintext P
- **Finalization**: Reinjection of the key K, extraction and validation check of an authentication tag T.

The decryption process is depicted in figure 5 and showcases the four different phases. It is also worth noting that the first two phases (Initialization and Associated Data Processing) of the decryption and encryption algorithms are identical. The detailed explanations are therefore omitted to avoid repetition; however, the specification can be found in the section 3.2.1 and 3.2.2.

Figure 5: `Ascon-AEAD128.dec` decryption [res23]



The pseudocode for the decryption process (with reference to the official Ascon specification) is provided in algorithm 2 and the last two phases will be explained in detail in the following sections.

Algorithm 2 `Ascon-AEAD128.dec`(K, N, A, C, T)

```
Input: 128-bit key $K$; 128-bit nonce $N$; Associated data $A$; Ciphertext $C$; 128-bit tag $T$
Output: Plaintext $P$ or fail
1: $IV \leftarrow$ 0x00001000808c0001                          ▷ Initialization
2: $S \leftarrow IV \| K \| N$
3: $S \leftarrow$ Ascon-p$[12](S)$
4: $S \leftarrow S \oplus (0^{192} \| K)$

5: if $|A| > 0$ then                                          ▷ Processing Associated Data
6:     $A_0, \dots, A_{m-1}, \widetilde{A_m} \leftarrow$ parse$(A, 128)$
7:     $A_m \leftarrow$ pad$(\widetilde{A_m}, 128)$
8:     for $i = 0$ to $m$ do
9:         $S_{[0:127]} \leftarrow S_{[0:127]} \oplus A_i$
10:        $S \leftarrow$ Ascon-p$[8](S)$
11:    end for
12: end if
13: $S \leftarrow S \oplus (0^{319} \| 1)$

14: $C_0, \dots, C_{n-1}, \widetilde{C_n} \leftarrow$ parse$(C, 128)$            ▷ Processing Ciphertext
15: for $i = 0$ to $n - 1$ do
16:     $P_i \leftarrow S_{[0:127]} \oplus C_i$
17:     $S_{[0:127]} \leftarrow C_i$
18:     $S \leftarrow$ Ascon-p$[8](S)$
19: end for
20: $\ell \leftarrow |\widetilde{C_n}|$
21: $\widetilde{P_n} \leftarrow S_{[0:\ell-1]} \oplus \widetilde{C_n}$
22: $S_{[\ell:127]} \leftarrow S_{[\ell:127]} \oplus (1 \| 0^{127-\ell})$
23: $S_{[0:\ell-1]} \leftarrow \widetilde{C_n}$

24: $S \leftarrow$ Ascon-p$[12](S \oplus (0^{128} \| K \| 0^{64}))$               ▷ Finalization
25: $T' \leftarrow S_{[192:319]} \oplus K$
26: if $T' == T$ then
27:     $P \leftarrow P_0 \| \dots \| P_{n-1} \| \widetilde{P_n}$
28:     return $P$
29: else
30:     return fail
31: end if
```

3.3.1 Ciphertext Processing

After initializing the internal state and processing the associated data in the first two phases, the algorithm is prepared to process the ciphertext and decrypt it into plaintext.
First, the ciphertext is parsed into 128-bit long blocks, where the final block $\widetilde{C_n}$ may be partially filled, depending on the size of the input ciphertext C:

$$A_0, \dots, A_{m-1}, \widetilde{A_m} \leftarrow \texttt{parse}(A, 128)$$

The algorithm then loops over every block apart from the last one $\widetilde{C_n}$, as this block contains a variable length ℓ that will be processed after the loop. For each block, the corresponding plaintext block P_i is recovered by XORing the current state's first 128 bits with the cipher block C_i:

$$P_i \leftarrow S_{[0:127]} \oplus C_i$$

This step principally inverts the encryption step, where the original encrypted block had computed $C_i = S_{[0:127]} \oplus P_i$. And because XOR is reversible, $P_i = S_{[0:127]} \oplus C_i$ recovers the original plaintext.
After recovering the plaintext block P_i, the state is then overridden with the ciphertext block C_i:

$$S_{[0:127]} \leftarrow C_i$$

This mirrors how encryption had injected the plaintext into the state, which means that by storing C_i back into the state, an identical reflection of the encryption process is created.
Finally, an eight-round permutation will be applied to the state:

$$S \leftarrow \texttt{Ascon-p}[8](S)$$

The permutation diffuses the injected ciphertext block throughout the internal state, just as it did for the plaintext blocks during the encryption process in 3.2.3.
After the loop, the final ciphertext block $\widetilde{C_n}$ needs to be processed. Since the block can have a variable length ℓ where $0 \leq \ell \leq 128$, it needs to be handled differently:

$$\ell \leftarrow |\widetilde{C_n}|$$
$$\widetilde{P_n} \leftarrow S_{[0:\ell-1]} \oplus \widetilde{C_n}$$

As one can see, the last partial plaintext block $\widetilde{P_n}$ is recovered by XORing the first ℓ bits of the state with $\widetilde{C_n}$.
The bits from index ℓ to 128 of the internal state is then updated with a single `1` bit followed by zeros:

$$S_{[\ell:127]} \leftarrow S_{[\ell:127]} \oplus (1\|0^{127-\ell})$$

This step acts as domain separation and also aligns with the encryption process, ensuring symmetry. Finally, the last ℓ bits of the state are set to the last ciphertext block bits:

$$S_{[0:\ell-1]} \leftarrow \widetilde{C_n}$$

This injection mirrors the encryption algorithm, where the last padded plaintext block was injected into the state. After this step, the state has absorbed all ciphertext blocks and extracted the plaintext.
Note that the plaintext/ciphertext processing steps in both encryption and decryption algorithms in `Ascon-AEAD128` are mirrored:

$$\begin{aligned}
&\text{Encryption:} \quad C_i \leftarrow S_{[0:127]} \oplus P_i, \quad S_{[0:127]} \leftarrow P_i, \quad S \leftarrow \texttt{Ascon-p}[8](S)\\
&\text{Decryption:} \quad P_i \leftarrow S_{[0:127]} \oplus C_i, \quad S_{[0:127]} \leftarrow C_i, \quad S \leftarrow \texttt{Ascon-p}[8](S).
\end{aligned}$$

Noticeably, aside from swapping P_i and C_i in the XOR, the state update is identical. Both parts absorb their respective blocks before applying an eight-round permutation. The mirroring ensures that each encryption step can be "replayed" in decryption, which means that both sides produce the same final internal state and therefore also the same authentication tag for verification.

3.3.2 Finalization

The finalization step occurs after all ciphertext has been processed and the plaintext blocks reconstructed. In this stage, the algorithm injects the secret key back into the state and computes the authentication tag.
The state is XORed with a constant containing the key K and permuted 12 times:

$$S \leftarrow \texttt{Ascon-p}[12](S \oplus (0^{128} \| K \| 0^{64}))$$

This step prepares the internal state for tag extraction. The new authentication tag T' is computed by taking the last 128 bits of the state and XORing it with the key K:

$$T' \leftarrow S_{[192:319]} \oplus K$$

Finally, the newly computed tag T' is compared to the input authentication tag T. If they match, authenticity is confirmed and the plaintext is returned by concatenating all the computed blocks together:

$$P \leftarrow P_0 \| \dots \| P_{n-1} \| \widetilde{P_n}$$

If there is any mismatch, the decryption process fails and returns `fail`, indicating an authentication error. This final tag comparison ensures that the data has not been tampered with by any unwanted adversaries:

$$\textbf{return}\, \texttt{fail}$$

Each step above mirrors the respective encryption sections in reverse, which means that the internal state is updated coherently and ultimately used to verify the tag to authenticate the plaintext. This completes the decryption algorithm `Ascon-AEAD128.dec`.

4 Hash and Extendable Output Functions

The Ascon family includes cryptographic hash and extendable output functions designed for lightweight applications. These algorithms are built upon the sponge construction and leverage the same permutation as the `Ascon-AEAD128` functions:

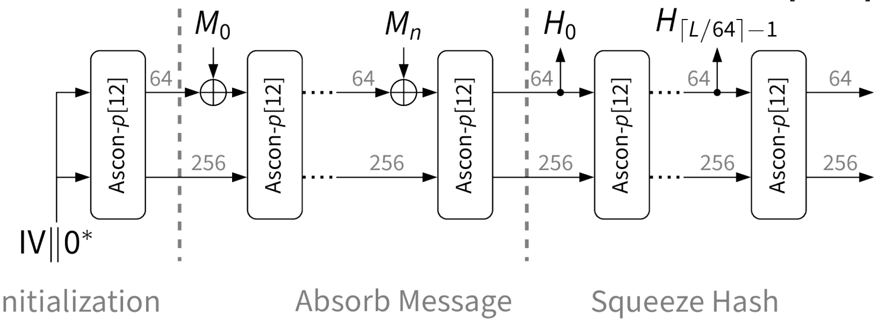
- `Ascon-Hash256`: This hash function converts a variable length message M and produces a 256-bit long digest.
- `Ascon-XOF128`: This extendable output function (XOF) produces an output of any desired length, unlike `Ascon-Hash256` with a fixed digest length.
- `Ascon-CXOF128`: This customized XOF extends the functionality of `Ascon-XOF128` by incorporating a customization string into the computation.

The following sections present the specifications of these functions in detail, as defined in the official Ason specification [asc24].

4.1 Specification of Ascon-Hash256

`Ascon-Hash256` is a sponge-based hash function that takes an input message M and produces a 256-bit long digest with 128-bit security. It operates on a 320-bit internal state split into five 64-bit words, similar to the previously specified `Ascon-AEAD128` functions.
Figure 6 depicts the structure of the `Ascon-Hash256` and `Ascoon-XOF128` algorithms. Note that both share the same structure, though they still differ in design (elaborated in section 4.2).

Figure 6: Structure of `Ascon-Hash256` and `Ascon-XOF128` [res23]



As shown in figure 6, `Ascon-Hash256` consists of three phases: Initialization, Message Absorption and Hash Squeezing. The pseudocode for the `Ascon-Hash256` algorithm is provided in algorithm 3 and will be described in detail in the following sections, with reference to the official Ascon specification pseudocode [asc24].

Algorithm 3 `Ascon-Hash256`(M)

```
Input: Message $M \in \{0,1\}^*$
Output: Hash output $H \in \{0,1\}^{256}$
 1: $IV \leftarrow$ 0x0000080100cc0002                                ▷ Initialization
 2: $S \leftarrow$ Ascon-p[12]($IV \| 0^{256}$)

 3: $M_0, \ldots, M_{n-1}, \widetilde{M_n} \leftarrow$ parse($M$, 64)          ▷ Absorbing phase
 4: $M_n \leftarrow$ pad($\widetilde{M_n}$, 64)
 5: for $i = 0$ to $n-1$ do
 6:     $S_{[0:63]} \leftarrow S_{[0:63]} \oplus M_i$
 7:     $S \leftarrow$ Ascon-p[12]($S$)
 8: end for
 9: $S_{[0:63]} \leftarrow S_{[0:63]} \oplus M_n$

10: $S \leftarrow$ Ascon-p[12]($S$)                                   ▷ Squeezing phase
11: for $i = 0$ to 2 do
12:     $H_i \leftarrow S_{[0:63]}$
13:     $S \leftarrow$ Ascon-p[12]($S$)
14: end for
15: $H_3 \leftarrow S_{[0:63]}$
16: $H \leftarrow H_0 \| H_1 \| H_2 \| H_3$
17: return $H$
```

4.1.1 Initialization

This phase is responsible for initializing the internal state with a constant initial value IV. The IV is incorporated into the first 64 bits of the state, which then gets permuted 12 times:

$$IV \leftarrow \texttt{0x0000080100cc0002}$$
$$S \leftarrow \texttt{Ascon-p}[12](IV \| 0^{256})$$

4.1.2 Message Absorption

After the initialization phase, the state is ready to absorb the input message M. First, the message is split and padded into different 64-bit long blocks for processing:

$$M_0, \ldots, M_{n-1}, \widetilde{M_n} \leftarrow \texttt{parse}(M, 64)$$
$$M_n \leftarrow \texttt{pad}(\widetilde{M_n}, 64)$$

Unlike the `Ascon-AEAD128` functions 3, all blocks are padded fully to 64 bits for processing, including the final block M_n.
Each block is then absorbed by XORing it into the first 64 bits of the internal state. After the absorption follows a 12-round Ascon permutation, ensuring that the state is prepared for the next message block:

$$S_{[0:63]} \leftarrow S_{[0:63]} \oplus M_i$$
$$S \leftarrow \texttt{Ascon-p}[12](S)$$

Note that the loop does not cover the last message block M_n, as the final permutation does not take place in the absorbing phase, but at the start of the squeezing phase:

$$S_{[0:63]} \leftarrow S_{[0:63]} \oplus M_n$$

The integration of the last message block concludes the absorbing phase of `Ascon-Hash256`.

4.1.3 Hash Squeezing

After all blocks are absorbed into the state, the state is ready to start squeezing out the hash blocks. First, a permutation takes place at the beginning of the squeezing phase. This makes sure that after the last message block is absorbed (from the absorption phase), the state is further diffused:

$$S \leftarrow \texttt{Ascon-p}[12](S)$$

Since `Ascon-Hash256` outputs a 256-bit long digest, only four 64-bit long hash blocks need to be digested ($4 \times 64 = 256$ bits). Here, the first three blocks (H_0, H_1, H_2) are squeezed out and the state undergoes a 12-round permutation for each hash block processed:

$$H_i \leftarrow S_{[0:63]}$$
$$S \leftarrow \texttt{Ascon-p}[12](S)$$

This step produces the hash blocks H_0, H_1 and H_2, with the final block squeezed out after the loop:

$$H_3 \leftarrow S_{[0:63]}.$$

Note that after the processing of H_3, no permutation takes place as the algorithm has finished processing the output digest at this point. After all blocks are extracted, the final hash digest is constructed by concatenating the hash blocks:

$$H \leftarrow H_0 \| H_1 \| H_2 \| H_3,$$

where H is the final 256-bit long digest.

4.2 Specification of Ascon-XOF128

The Extendable Output Function `Ascon-XOF128` acts as an extension of `Ascon-Hash256` that allows the user to specify the output's desired length, in addition to the message input M. As depicted in figure 6, the structures of these two algorithms are equal, though fundamentally, `Ascon-XOF128` differs to `Ascon-Hash256` in multiple ways:

1. Since `Ascon-XOF128` can have an arbitrary output length, the length $L > 0$ needs to be specified as an input.

2. `Ascon-Hash256` has a fixed 256-bit digest. Since the output length is specified in `Ascon-XOF128`, the number of blocks that are processed during the squeezing phase is equal to $\lceil L/64 \rceil$.

3. The initial value IV differs from the `Ascon-Hash256` algorithm by one bit.

Similar to `Ascon-Hash256`, the `Ascon-XOF128` function consists of three phases: Initialization, Message Absorption and Hash Squeezing. The pseudocode is provided in algorithm 4, with reference to the original Ascon specification pseudocode and will be explained in detail in the following sections.

Algorithm 4 `Ascon-XOF128`(M, L)

Input: Bitstring $M \in \{0,1\}^*$; Output length $L > 0$
Output: Digest $H \in \{0,1\}^L$

```
 1: $IV \leftarrow$ 0x0000080000cc0003                          ▷ Initialization
 2: $S \leftarrow \texttt{Ascon-p}[12](IV \| 0^{256})$

 3: $M_0, \ldots, M_{n-1}, \widetilde{M_n} \leftarrow \texttt{parse}(M, 64)$     ▷ Absorbing phase
 4: $M_n \leftarrow \texttt{pad}(\widetilde{M_n}, 64)$
 5: for $i = 0$ to $n - 1$ do
 6:     $S_{[0:63]} \leftarrow S_{[0:63]} \oplus M_i$
 7:     $S \leftarrow \texttt{Ascon-p}[12](S)$
 8: end for
 9: $S_{[0:63]} \leftarrow S_{[0:63]} \oplus M_n$

10: $S \leftarrow \texttt{Ascon-p}[12](S)$                          ▷ Squeezing phase
11: $h \leftarrow \lceil L/64 \rceil - 1$
12: for $i = 0$ to $h - 1$ do
13:     $H_i \leftarrow S_{[0:63]}$
14:     $S \leftarrow \texttt{Ascon-p}[12](S)$
15: end for
16: $H_h \leftarrow S_{[0:63]}$
17: $H' \leftarrow H_0 \| H_1 \| \ldots \| H_h$
18: $H \leftarrow H'_{[0:L-1]}$
19: return $H$
```

4.2.1 Initialization and Message Absorption

The initialization and message absorption phases in `Ascon-XOF128` are mostly identical to the ones in the `Ascon-Hash256`. The only difference lies in the initialization phase, where the initial value IV only differ by one bit:

$$\begin{aligned} \texttt{Ascon-Hash256}: &\quad 0x0000080100cc0002 \\ \texttt{Ascon-XOF128}: &\quad 0x0000080000cc0003. \end{aligned}$$

Since both phases are explained in sections 4.1.1 and 4.1.2 with reference to the pseudocode, a detailed explanation will be omitted in this algorithm in order to avoid repetition.

4.2.2 Hash Squeezing

After the state has been initialized and the message blocks absorbed, the algorithm enters the squeezing phase, where a variable-length hash output is produced. This phase differs from the `Ascon-Hash256` algorithm, as the digest length needs to be adjusted to the length of the input variable L. First, a 12-round permutation is applied to the state before processing the hash blocks in order to ensure that the state is diffused:

$$S \leftarrow \texttt{Ascon-p}[12](S)$$

The number of 64-bit hash blocks (denoted by h) that need to be squeezed out is then computed by dividing L with 64:

$$h \leftarrow \lceil L/64 \rceil - 1.$$

Note that this step subtracts 1 to h, as this gives the number of full iterations in the loop for hash block processing. The last block is handled separately, outside of the loop.

In a loop, the hash blocks are then computed by taking the first 64 bits of the internal state. This step is similar to the `Ascon-Hash256` algorithm, as the only difference lies in the length of the loop (fixed loop length in `Ascon-Hash256` and variable loop length dependent on L in `Ascon-XOF128`):

$$H_i \leftarrow S_{[0:63]}$$
$$S \leftarrow \texttt{Ascon-p}[12](S)$$

Finally, the last hash block H_h is produced after the loop:

$$H_h \leftarrow S_{[0:63]}$$

Note that H_h has a fixed length of 64 bits at this point, since Ascon operates on 64-bit blocks at a time. Since the final digest should have the length of L, the final concatenated hash H' is truncated, so that it has the correct length:

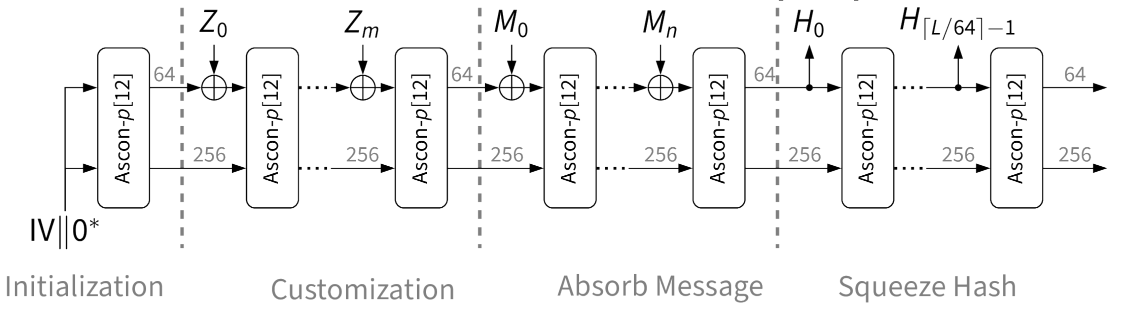
$$H' \leftarrow H_0 \| H_1 \| \dots \| H_h$$
$$H \leftarrow H'_{[0:L-1]}$$

H is then returned and this step concludes the `Ascon-XOF128` algorithm.

4.3 Specification of Ascon-CXOF128

`Ascon-CXOF128` is a customized version of `Ascon-XOF128` and allows the user to add a customization string as input. This input is then used in the computation of the final digest, providing further customization. Note that an input of two different customization strings and the same message produces two different digests. Figure 7 illustrates the structure of `Ascon-CXOF128`.

Figure 7: Structure of `Ascon-CXOF128` [res23]



Noticeably, `Ascon-CXOF128` contains the same phases (Initialization, Message Absorption, and Hash Squeezing) as the previously specified hash and extendable output functions; however, a new phase - a customization phase - is added before absorbing

the message.
The algorithm is defined as follows:

$$\texttt{Ascon-CXOF128}(M, L, Z) = H,$$

where M is the message, L the desired length of the digest, Z the customization string, and H the output digest.
`Ascon-CXOF128` is very similar to the non-customizable version `Ascon-XOF128` and varies in the following ways:

1. The initial value IV differs from the `Ascon-XOF128` algorithm by one bit.
2. In addition to the message M and digest length L, `Ascon-CXOF128` takes as an additional input a customization string Z, where Z can be at most 2048 bits long.

For comparison, the Initial Values IV of the hash and extendable output functions are as follows:

$$\begin{aligned} \texttt{Ascon-Hash256}: &\quad 0x0000080100cc0002 \\ \texttt{Ascon-XOF128}: &\quad 0x0000080000cc0003 \\ \texttt{Ascon-CXOF128}: &\quad 0x0000080000cc0004. \end{aligned}$$

The pseudocode for `Ascon-CXOF128` is provided in algorithm 5, with reference to the original Ascon specification pseudocode. Since the initialization, message absorption, and hash squeezing phases are the same as the ones already specified in `Ascon-COF128` 4.2, only the customization phase will be explained in detail and the other phases omitted.

Algorithm 5 `Ascon-CXOF128`(M, L, Z)

```
Input: Bitstring $M \in \{0,1\}^*$; Output length $L > 0$; Customization string $Z \in \{0,1\}^*$,
    where $|Z| \le 2048$
Output: Digest $H \in \{0,1\}^L$

1: $IV \leftarrow$ 0x0000080000cc0004                          ▷ Initialization
2: $S \leftarrow$ Ascon-p$[12](IV \| 0^{256})$

3: $Z_0 \leftarrow$ int64$(|Z|)$                                ▷ Customization
4: $Z_1, \ldots, Z_{m-1}, \widetilde{Z}_m \leftarrow$ parse$(Z, 64)$
5: $Z_m \leftarrow$ pad$(\widetilde{Z}_m, 64)$
6: for $i = 0$ to $m$ do
7:     $S_{[0:63]} \leftarrow S_{[0:63]} \oplus Z_i$
8:     $S \leftarrow$ Ascon-p$[12](S)$
9: end for

10: $M_0, \ldots, M_{n-1}, \widetilde{M}_n \leftarrow$ parse$(M, 64)$     ▷ Absorbing Message
11: $M_n \leftarrow$ pad$(\widetilde{M}_n, 64)$
12: for $i = 0$ to $n - 1$ do
13:     $S_{[0:63]} \leftarrow S_{[0:63]} \oplus M_i$
14:     $S \leftarrow$ Ascon-p$[12](S)$
15: end for
16: $S_{[0:63]} \leftarrow S_{[0:63]} \oplus M_n$

17: $S \leftarrow$ Ascon-p$[12](S)$                                ▷ Squeezing
18: $h \leftarrow \lceil L/64 \rceil - 1$
19: for $i = 0$ to $h - 1$ do
20:     $H_i \leftarrow S_{[0:63]}$
21:     $S \leftarrow$ Ascon-p$[12](S)$
22: end for
23: $H_h \leftarrow S_{[0:63]}$
24: $H' \leftarrow H_0 \| \ldots \| H_h$
25: $H \leftarrow H'_{[0:L-1]}$
26: return $H$
```

4.3.1 Customization

The customization phase starts after the state has been initialized with the IV during the initialization phase. First, the whole length of the customization string is encoded as a 64-bit integer block Z_0. This block acts as a domain separation, ensuring that strings with different lengths all map to different outputs:

$$Z_0 \leftarrow \texttt{int64}(|Z|)$$

After this step, the whole customization string gets parsed and padded into 64-bit blocks:

$$Z_1, \ldots, Z_{m-1}, \widetilde{Z}_m \leftarrow \texttt{parse}(Z, 64)$$
$$Z_m \leftarrow \texttt{pad}(\widetilde{Z}_m, 64)$$

Each block Z_i is then injected into the internal state, followed by a 12-round permutation, similar to the absorption phase:

$$S_{[0:63]} \leftarrow S_{[0:63]} \oplus Z_i$$
$$S \leftarrow \texttt{Ascon-p}[12](S)$$

Once all blocks have been injected into the internal state, the algorithm transitions into the message absorption phase and finally squeezes the hash digest H.

5 Conclusion

Ascon offers a family of lightweight cryptographic functions for resource-constrained environments. Along with the Ascon permutation, the functions `Ascon-AEAD128`, `Ascon-Hash256`, `Ascon-XOF128`, and `Ascon-CXOF128` deliver strong security against known cryptanalytic attacks.
Despite these strengths, Ascon does not specifically target every use case. For example, it omits digital signature schemes and key-exchange protocols. After all, its design prioritizes simplicity and minimal footprint. Hence, some heavyweight ciphers like AES or SHA may be better for high-bandwidth and high-latency environments - especially as AES provides higher bit security [aes01].
Furthermore, as a relatively newer algorithm in the cryptographic field, Ascon has not undergone the same extensive analysis compared to more mature algorithms like AES. This means that there's a possibility of future vulnerabilities that are to be discovered that were not apparent during its initial evaluation.

To conclude, while Ascon excels in lightweight cryptographic applications with its efficient design, it is still best viewed as a complement rather than a replacement for other well-established algorithms like AES. Hence, the choice between Ascon and other heavier ciphers should be made with consideration of the specific constraints and requirements of the environment.

A Appendix

The acronyms and terms that are used are referenced from the official Ascon publication [asc24] this paper is based on.

B Acronyms

Acronym	Definition
AD	Associated Data
AE	Authenticated Encryption
AEAD	Authenticated Encryption with Associated Data
AES	Advanced Encryption Standard
NIST	National Institute of Standards and Technology
SHA	Secure Hash Algorithm
SPN	Substitution-Permutation Network
XOF	eXtendable-Output Function
XOR	Exclusive OR

Table 3: Acronyms

C Terms

Term	Definition
associated data	Input data that is authenticated, but not encrypted.
bit	A binary digit, 0 or 1. In this standard bits are indicated in the Courier New font.
bit string	A finite, ordered sequence of bits.
digest	Hash value.
eXtendable Output Function (XOF)	A function on bit strings in which the output can be extended to any desired length.
hash function	A mathematical function that maps a string of arbitrary length to a fixed-length string.
message	Input to the hash function.
nonce	An input value to the authenticated encryption algorithm that is used only once for encryption performed under a given key.
rate	The number of input bits processed or output bits generated per invocation of the underlying permutation.
secret key	A cryptographic key used by a secret-key (i.e., symmetric) cryptographic algorithm and that is not made public.
truncation	A process that shortens an input bitstring, preserving only a substring of a specified length.

Table 4: Acronyms

D Notations

Notation	Definition
K	128-bit secret key
N	128-bit nonce
A	Associated data
A_i	ith block of associated data A
P	Plaintext
P_i	ith block of plaintext P
C	Ciphertext
C_i	ith block of ciphertext C
Z	Customization string
Z_i	ith block of customization string Z
T	128-bit authentication tag
IV	64-bit constant initial value
$fail$	Error message to indicate that the verification of authenticated ciphertext failed
M	Message
M_i	ith block of message M
H	Hash value
H_i	ith block of hash value H
S	320-bit internal state of the underlying permutation
$S_0, \ldots, S_4$	The five 64-bit words of the internal state S, where $S = S_0 \parallel S_1 \parallel \cdots \parallel S_4$
$s(i, j)$	jth bit of S_i, for $0 \leq i \leq 4$, $0 \leq j \leq 63$
$S_i[j]$	jth byte of state word S_i, for $0 \leq i \leq 4$, $0 \leq j \leq 7$
λ	Length of the truncated tag in bits
r	The rate of an algorithm
c_i	The constant value for round i of the Ascon permutation
p_C, p_S, p_L	Constant-addition, substitution, and linear layers of the round function p

Table 5: Notation used throughout the cryptographic algorithm specification

E Basic Operations and Functions

Functions	**Definition**
$\{0,1\}^*$	The set of all finite bit strings, including the empty string
$\{0,1\}^s$	The set of all bit strings of length s
0^s	When $s \geq 0$, the bit string of s consecutive 0s. When $s = 0$, it is the empty string.
$\|X\|$	Length of the bitstring X in bits
$X \parallel Y$	Concatenation of bitstrings X and Y
$x \times y$	Multiplication of integers x and y
$x + y$	Addition of integers x and y
$x - y$	Subtraction of integers x and y
x/y	Division of integer x by non-zero integer y
$x \bmod y$	Remainder in integer division of x by y
$\lceil x \rceil$	The smallest integer greater than or equal to real number x
$\lfloor x \rfloor$	The largest integer less than or equal to real number x
$f \circ g$	Composition of functions f and g, i.e., $f(g(x))$
$\odot$	Bitwise AND operation
$\oplus$	Bitwise XOR operation
$X \ggg i$	Right rotation (circular shift) of 64-bit word X by i bits
$X \ll i$	Left shift of X by i bits
$X[i:j]$	Substring of X from index i to j, inclusive. If $i > j$, result is empty string. If $i = j$, result is a single bit.
$x == y$	Boolean equality check; true if x equals y, false otherwise

Table 6: Functions and operations used in the specification

F Auxiliary Functions

Auxiliary functions are essential components of the Ascon algorithm and act as core components in its permutation and processing steps. Two such functions are used throughout the specification: the *parse* function and the *pad* function.

F.1 parse(X, r)

The **parse** function (Algorithm 6) is used to divide an input bitstring X into blocks with a fixed size r. This is necessary because Ascon processes data in blocks, and splitting the input accordingly ensures that the data gets processed consistently. The function returns all full-size blocks, as well as a final partial block containing any leftover bits that do not fit into the fixed size r.

Algorithm 6 parse(X, r)

```
Input: bitstring $X$, rate $r$
Output: bitstrings $X_0, \dots, X_{\ell-1}, \widetilde{X_\ell}$
1: $\ell \leftarrow \left\lfloor \frac{|X|}{r} \right\rfloor$        ▷ Compute how many full chunks of size $r$ fit into $X$
2: for $i = 0$ to $\ell - 1$ do
3:     $X_i \leftarrow X[i \times r : (i+1) \times r - 1]$        ▷ Extract the $i$-th chunk of $r$ bits
4: end for
5: $\widetilde{X_\ell} \leftarrow X[\ell \times r : |X| - 1]$        ▷ Get the remaining bits that don't form a full chunk
6: return $X_0, \dots, X_{\ell-1}, \widetilde{X_\ell}$        ▷ Return all full chunks and the leftover bits
```

Example: Let $X = 1100110010101110$ (16 bits), and let $r = 6$. Then:

$$X_0 = 110011$$
$$X_1 = 001010$$
$$\widetilde{X_2} = 1110$$

F.2 pad(X, r)

The **pad** function (Algorithm 7) ensures that the bitstring X is extended so that its total length becomes a multiple of the block size r. This is done by appending a single '1' bit followed by the minimum number of '0' bits required to reach the desired block size r — this is required in Ascon for proper block processing.

Algorithm 7 pad(X, r)

```
Input: bitstring $X$, rate $r$
Output: padded bitstring $X'$
1: $j \leftarrow (-|X| - 1) \bmod r$   ▷ Compute how many zeros are needed to make the length
   divisible by $r$ after adding a 1
2: $X' \leftarrow X \parallel 1 \parallel 0^j$        ▷ Append a single 1 bit and then $j$ zero bits
3: return $X'$        ▷ Return the padded bitstring
```

Example: Let $X = 11001100101$ (11 bits), and let $r = 8$. We compute j:

$$j = (-11 - 1) \bmod 8 = -12 \bmod 8 = 4$$

So:

$$X' = 11001100101\mathbf{1}0000$$

Now, $|X'| = 16$, which is divisible by 8.

G Determination of the Initial Values

Each Ascon algorithm mentioned in this paper has a fixed 64-bit Initial Value IV specified as:

Ascon Variant	**Initial Value (IV)**
Ascon-AEAD128	0x00001000808c0001
Ascon-Hash256	0x0000080100cc0002
Ascon-XOF128	0x0000080000cc0003
Ascon-CXOF128	0x0000080000cc0004

Table 7: Initial values (IVs) for different Ascon variants

References

[aes01] National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*. Tech. rep. Federal Information Processing Standards Publication 197. U.S. Department of Commerce, 2001. DOI: 10.6028/NIST.FIPS.197-upd1. URL: https://doi.org/10.6028/NIST.FIPS.197-upd1.

[asc24] Meltem Sönmez Turanet al. *Ascon-Based Lightweight Cryptography Standards for Constrained Devices*. Tech. rep. NIST Special Publication 800-232 (Initial Public Draft). National Institute of Standards and Technology, 2024. DOI: 10.6028/NIST.SP.800-232.ipd. URL: https://doi.org/10.6028/NIST.SP.800-232.ipd.

[crat94] Eli Biham. *New Types of Cryptanalytic Attacks Using Related Keys*. Tech. rep. 1994. DOI: 10.1007/BF00203965. URL: https://doi.org/10.1007/BF00203965.

[nist17] NIST. *Finalists - lightweight cryptography*. 2017. URL: https://csrc.nist.gov/projects/lightweight-cryptography/finalists.

[res23] Ascon Team. *Ascon – Resources*. https://ascon.isec.tugraz.at/resources.html. Accessed: 2025-07-30. 2023.

[shs15] National Institute of Standards and Technology. *Secure Hash Standard (SHS)*. Tech. rep. Federal Information Processing Standards Publication 180-4. Gaithersburg, MD: U.S. Department of Commerce, 2015. DOI: 10.6028/NIST.FIPS.180-4. URL: https://doi.org/10.6028/NIST.FIPS.180-4.

[spng12] Guido Bertoni et al. "Duplexing the Sponge: Single-Pass Authenticated Encryption and Other Applications". In: *Selected Areas in Cryptography (SAC 2011)*. Vol. 7118. Lecture Notes in Computer Science. Springer, 2012, pp. 320–337. DOI: 10.1007/978-3-642-28496-0_19. URL: https://doi.org/10.1007/978-3-642-28496-0_19.