NIST Special Publication 800
NIST SP 800-232

Ascon-Based Lightweight Cryptography Standards for Constrained Devices

Authenticated Encryption, Hash, and Extendable Output Functions

Meltem Sönmez Turan
Kerry A. McKay
Donghoon Chang
Jinkeon Kang
John Kelsey

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-232

NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
U.S. DEPARTMENT OF COMMERCE

NIST Special Publication 800
NIST SP 800-232

Ascon-Based Lightweight Cryptography Standards for Constrained Devices

Authenticated Encryption, Hash, and Extendable Output Functions

Meltem Sönmez Turan
Kerry A. McKay
Jinkeon Kang*
John Kelsey

Computer Security Division
Information Technology Laboratory

* Former Foreign Guest Researcher; all work for this publication was done while at NIST.

Donghoon Chang

Strativia

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-232

August 2025

U.S. Department of Commerce
Howard Lutnick, Secretary

National Institute of Standards and Technology
Craig Burkhardt, Acting Under Secretary of Commerce for Standards and Technology and Acting NIST Director

Authority

NIST Technical Series Policies
Copyright, Use, and Licensing Statements
NIST Technical Series Publication Identifier Syntax

Publication History
Approved by the NIST Editorial Review Board on 2025-07-09

How to cite this NIST Technical Series Publication:
Meltem Sönmez Turan, Kerry A. McKay, Donghoon Chang, Jinkeon Kang, John Kelsey (2025) Ascon-Based Lightweight Cryptography Standards for Constrained Devices. (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) NIST SP 800-232. https://doi.org/10.6028/NIST.SP.800-232

Author ORCID iDs
Meltem Sönmez Turan: 0000-0002-1950-7130
Kerry A. McKay: 0000-0002-5956-587X
Donghoon Chang: 0000-0003-1249-2869
Jinkeon Kang: 0000-0003-2142-8236
John Kelsey: 0000-0002-3427-1744

Contact Information
SP800-232-comments@list.nist.gov

Additional Information
Additional information about this publication is available at https://csrc.nist.gov/pubs/sp/800/232/final, including related content, potential updates, and document history.

Abstract

In 2023, the National Institute of Standards and Technology (NIST) announced the selection of the Ascon family of algorithms designed by Dobraunig, Eichlseder, Mendel, and Schläffer to provide efficient cryptographic solutions for resource-constrained devices. This decision emerged from a rigorous, multi-round lightweight cryptography standardization process. The Ascon family includes a suite of cryptographic primitives that provide Authenticated Encryption with Associated Data (AEAD), hash function, and eXtendable Output Function (XOF) capabilities. The Ascon family is characterized by lightweight, permutation-based primitives and provides robust security, efficiency, and flexibility, making it ideal for resource-constrained environments, such as Internet of Things (IoT) devices, embedded systems, and low-power sensors. The family is developed to offer a viable alternative when the Advanced Encryption Standard (AES) may not perform optimally. This standard outlines the technical specifications and security properties of `Ascon-AEAD128`, `Ascon-Hash256`, `Ascon-XOF128`, and `Ascon-CXOF128`.

Keywords

Ascon; authenticated encryption; constrained devices; eXtendable Output Function (XOF); hash function; lightweight cryptography; permutation-based cryptography; standardization.

Reports on Computer Systems Technology

Patent Disclosure Notice

Table of Contents

List of Tables

List of Figures

Acknowledgments

The authors of the standard express their gratitude to the Ascon designers — Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer — for their valuable comments and suggestions during the standardization process.

The authors also acknowledge and appreciate contributions from their colleagues at NIST during the selection process, including Lawrence Bassham, Çağdaş Çalık, Deukjo Hong, and Noah Waller. The authors also thank Elaine Barker, Luís T. A. N. Brandão, Lily Chen, Andrew Regenscheid, Noah Ross, Sara Kerman, Yu Long Chen, and Yu Sasaki for their technical and administrative support.

1. Introduction

This standard specifies the Ascon family of algorithms to provide Authenticated Encryption with Associated Data (AEAD), a hash function, and two eXtendable Output Functions (XOFs). The Ascon family is designed to be efficient in constrained environments. The algorithms included in this standard are as follows:

1. `Ascon-AEAD128` is a nonce-based AEAD scheme, offering 128-bit security strength in the single-key setting.
2. `Ascon-Hash256` is a cryptographic hash function that produces a 256-bit hash of the input messages, offering a security strength of 128 bits.
3. `Ascon-XOF128` is a XOF, where the output size of the hash of the message can be selected by the user, and the supported security strength is up to 128 bits.
4. `Ascon-CXOF128` is a customized XOF that allows users to specify a customization string and choose the output size of the message hash. It supports a security strength of up to 128 bits.

Development of the Ascon family. Ascon (version v1) [1] was first submitted to CAESAR (*Competition for Authenticated Encryption: Security, Applicability, and Robustness*) [1] in 2014. The submission included two AEAD algorithms: a primary recommendation, Ascon-128, with a 128-bit key and the secondary recommendation, Ascon-96, with a 96-bit key. Updated versions v1.1 [2] for Round 2 and v1.2 [3] for Round 3 included minor tweaks, such as reordering the round constants, and the secondary recommendation was updated to Ascon-128a. In 2019, Ascon-128 and Ascon-128a were selected as the first choice for the lightweight authenticated encryption use case in the final portfolio of the CAESAR competition.

NIST Lightweight Cryptography Standardization Process. In 2015, the National Institute of Standards and Technology (NIST) initiated the Lightweight Cryptography Standardization Process to develop cryptographic standards that are suitable for constrained environments in which conventional cryptographic standards (e.g., AES-GCM [4, 5], SHA-2 [6] and SHA-3 [7] hash function families) may be resource-intensive. In February 2023, NIST announced the decision to standardize the Ascon family [8] for lightweight cryptography applications. For more information, refer to NIST Internal Report (IR) 8268 [9], IR 8369 [10], and IR 8454 [11].

Differences from the Ascon submission v1.2. The technical differences between this standard and the Ascon submission [8] are provided below:

1. **Permutations.** The Ascon submission defined three Ascon permutations with 6, 8, and 12 rounds. This standard specifies additional Ascon permutations by providing round

[1]CAESAR is a competition organized by a group of international cryptologic researchers to identify a portfolio of authenticated encryption schemes that offer advantages over AES-GCM and are suitable for widespread adoption. The final portfolio of the competition was announced in February 2019. For more information, see https://competitions.cr.yp.to/caesar.html.

constants for up to 16 rounds to accommodate potential functionality extensions in the future.

2. **AEAD variants.** The Ascon submission package defined AEAD variants ASCON-128, ASCON-128a, and ASCON-80pq. This standard specifies the `Ascon-AEAD128` algorithm, which is based on ASCON-128a.

3. **Hash function variants.** The Ascon submission defined ASCON-HASH and ASCON-HASHA. This standard specifies `Ascon-Hash256`, which is based on ASCON-HASH.

4. **XOF variants.** The Ascon submission defined two XOFs, ASCON-XOF and ASCON-XOFA. This standard specifies `Ascon-XOF128`, which is based on ASCON-XOF, and a new customized XOF, `Ascon-CXOF128`.

5. **Initial values.** The initial values of the algorithms have been updated to support a new format that accommodates potential functionality extensions.

6. **Endianness.** The endianness has been switched from big endian to little endian to improve performance on little-endian microcontrollers.

7. **Truncation and nonce masking.** The implementation options of `Ascon-AEAD128` with truncation and nonce masking have been added.

Main features of Ascon. The main features of the Ascon family are:

- **Multiple functionalities.** The same permutations are used to construct multiple functionalities, which allows an implementation of AEAD, hash, and XOF functionalities to share logic and, therefore, have a more compact implementation than functions that were developed independently.

- **Online and single pass.** `Ascon-AEAD128` is online, meaning that the i-th ciphertext block is determined by the key, nonce, associated data, and first i plaintext blocks. Ascon family members require only a single pass over the data.

- **Inverse-free.** Since all of the Ascon family members only use the underlying permutations in the forward direction, implementing the inverse permutations is not needed.

Organization. Section 2 provides preliminaries, including the acronyms, terms, definitions, notation, basic operations, and auxiliary functions. Section 3 specifies the Ascon permutations for up to 16 rounds. Section 4 specifies the `Ascon-AEAD128`AEAD scheme, provides some implementation options for truncation and nonce masking, lists the requirements for validation, and provides security properties. Section 5 specifies the `Ascon-Hash256` hash function, the `Ascon-XOF128` XOF, and the `Ascon-CXOF128` customized XOF and describes their security properties. Section 6 provides information about conformance. Appendix A provides additional notes and conversion functions for implementations. Appendix B provides additional information regarding the construction of initial values.

2. Preliminaries

Table 1 lists the acronyms used in this standard.

Table 1. Acronyms

Acronym	Definition
AD	**A**ssociated **D**ata
AE	**A**uthenticated **E**ncryption
AEAD	**A**uthenticated **E**ncryption with **A**ssociated **D**ata
AES	**A**dvanced **E**ncryption **S**tandard
CAESAR	**C**ompetition for **A**uthenticated **E**ncryption: **S**ecurity, **A**pplicability, and **R**obustness
CXOF	**C**ustomized e**X**tendable-**O**utput **F**unction
GCM	**G**alois/**C**ounter **M**ode
NIST	**N**ational **I**nstitute of **S**tandards and **T**echnology
SHA	**S**ecure **H**ash **A**lgorithm
SPN	**S**ubstitution–**P**ermutation **N**etwork
SP	**S**pecial **P**ublication
XOF	e**X**tendable-**O**utput **F**unction
XOR	e**X**clusive **OR**

Table 2 defines the terms used in this standard.

Table 2. Terms and definitions

Term	Definition
approved	An algorithm or technique that is either specified or adopted in a FIPS publication or NIST Special Publication (SP) in the Computer Security SP 800 series (i.e., FIPS-approved or NIST-recommended).
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.
bitstring	A finite, ordered sequence of bits.

Table 2. Terms and definitions

Term	Definition
capacity	The width of the underlying permutation minus the rate.
digest	Output of a hash function or XOF.
eXtendable-Output Function (XOF)	A function on bit strings in which the output can be extended to any desired length.
forgery	A (ciphertext, tag) pair produced by an adversary who is not knowledgeable of the secret key and yet is accepted as valid by the verified decryption procedure.
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.
nonce-misuse	A setting in which a nonce is used more than once for the encryption algorithm under a given key.
nonce-respecting	A setting in which a nonce is never repeated for the encryption algorithm 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 that is used by a secret-key (i.e., symmetric) cryptographic algorithm and not made public.
shall	Term used to express a requirement that needs to be fulfilled to claim conformance to this standard.
should	Term used to indicate a strong recommendation but not a requirement of this standard. Ignoring the recommendation could result in undesirable results.
tag	A cryptographic checksum on data that is designed to reveal both accidental errors and the intentional modification of the data whose computation and verification require knowledge of a secret key.
truncation	A process that shortens an input bitstring, preserving only a sub-string of a specified length.

Table 3 lists the notations used in this standard.

Table 3. Notations

Notation	Definition
K	128-bit secret key
N	128-bit nonce
A	Associated data
A_i	i^{th} block of associated data A
P	Plaintext
P_i	i^{th} block of plaintext P
C	Ciphertext
C_i	i^{th} block of ciphertext C
Z	Customization string
Z_i	i^{th} 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	i^{th} block of message M
H	Hash value H
H_i	i^{th} block of hash value H
$\mathcal{S}$	320-bit internal state of the underlying permutation
$S_0, \ldots, S_4$	The five 64-bit words of the internal state $\mathcal{S}$, where $\mathcal{S} = S_0 \parallel S_1 \parallel \ldots \parallel S_4$
$s_{(i,j)}$	j^{th} bit of S_i, $0 \le i \le 4$, $0 \le j \le 63$
$S_i[j]$	j^{th} byte of state word S_i for $0 \le i \le 4$, $0 \le j \le 7$
$S_{[i:j]}$	The subset of state $\mathcal{S}$ beginning at index i and ending at index j, inclusive. When $i > j$, $S_{[i:j]}$ is the empty string. When $i = j$, $S_{[i:j]}$ is a single bit.
λ	Length of the truncated tag in bits
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 4 lists the basic operations and functions used in this standard.

Table 4. 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 $s \geq 0$, 0^s is the bit string that consists of s consecutive 0s. When $s = 0$, then 0^s is the empty string.
$\|X\|$	Length of the bitstring X in bits
$X \| Y$	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 and non-zero integer y
x mod y	Remainder in the integer division of x by y
$\lceil x \rceil$	For a real number x, the smallest integer greater than or equal to x
$\lfloor x \rfloor$	For a real number x, the largest integer less than or equal to x
$f \circ g$	Composition of functions f and g (e.g., for functions $f(x)$ and $g(x)$, $f \circ g$ is evaluated as $f(g(x))$)
$\odot$	Bitwise AND operation
$\oplus$	Bitwise XOR operation
$X \ggg i$	Right rotation (circular shift) by i bits of the 64-bit word X, where the least significant bit is the rightmost bit
$X \ll i$	Left shift by i bits
$X_{[i:j]}$	The subset of bitstring X beginning at index i and ending at index j, inclusive. When $i > j$, $X_{[i:j]}$ is the empty string. When $i = j$, $X_{[i:j]}$ is a single bit.
$x == y$	Boolean operator to perform an equality comparison, (i.e., `true` if x is equal to y; otherwise, `false`)
0x	Hexadecimal notation
int64(x)	64-bit representation of integer x

2.1. Auxiliary Functions

Parse function. The parse(X, r) function parses the input bitstring X into a sequence of blocks $X_0, X_1, \dots, \widetilde{X_\ell}$, where $\ell \leftarrow \lfloor |X|/r \rfloor$ (i.e., $X \leftarrow X_0 \parallel X_1 \parallel \dots \parallel \widetilde{X_\ell}$). The X_i blocks for $0 \le i \le \ell - 1$ each have a bit length r, whereas $0 \le \widetilde{X_\ell} \le r - 1$ (see Algorithm 1). When $|X| \bmod r = 0$, the final block is empty (i.e., $|\widetilde{X_\ell}| = 0$).

Algorithm 1 parse(X, r)

```
Input: bitstring $X$, a positive integer $r$
Output: bitstrings $X_0, \dots, X_{\ell-1}, \widetilde{X_\ell}$

$\ell \leftarrow \lfloor |X|/r \rfloor$
for $i = 0$ to $\ell - 1$ do
    $X_i \leftarrow X_{[i \times r : (i+1) \times r - 1]}$
end for
$\widetilde{X_\ell} \leftarrow X_{[\ell \times r : |X| - 1]}$
return $X_0, \dots, X_{\ell-1}, \widetilde{X_\ell}$
```

Padding rule. The function pad(X, r) appends the bit 1 to the bitstring X, followed by the bitstring 0^j, where j is equal to $(-|X| - 1) \bmod r$. The length of the output bitstring is a multiple of r (see Algorithm 2). For examples of padding when representing the data as 64-bit unsigned integers, see Appendix A.2.

Algorithm 2 pad(X, r)

```
Input: bitstring $X$, a positive integer $r$
Output: padded bitstring $X'$

$j \leftarrow (-|X| - 1) \bmod r$
$X' \leftarrow X \parallel 1 \parallel 0^j$
return $X'$
```

3. Ascon Permutations

This section specifies the rnd-round $Ascon\text{-}p[rnd]$ permutations, where rnd indicates the number of rounds to be performed and $1 \leq rnd \leq 16$. The permutations follow the Substitution-Permutation-Network (SPN) structure and consist of iterations of the round function p that is defined as the composition of three steps

$$p = p_L \circ p_S \circ p_C, \tag{1}$$

where p_C is the constant-addition layer (see Sec. 3.2), p_S is the substitution layer (see Sec. 3.3), and p_L is the linear diffusion layer (see Sec. 3.4). This composition can also be written as a series of function invocations on an input x as $p_L(p_S(p_C(x)))$.

Note that $Ascon\text{-}p[8]$ and $Ascon\text{-}p[12]$ are the main building blocks of the Ascon family, and the permutation instantiated with other numbers of rounds may later be used to standardize other functionalities.

3.1. Internal State

The permutations operate on the 320-bit state $\mathcal{S}$, which is represented as five 64-bit words denoted as S_i for $0 \leq i \leq 4$:

$$\mathcal{S} = S_0 \parallel S_1 \parallel S_2 \parallel S_3 \parallel S_4. \tag{2}$$

Let $s_{(i,j)}$ represent the jth bit of S_i, $0 \leq j < 64$. In this specification of the Ascon permutation, each state word represents a 64-bit unsigned integer, where the least significant bit is the rightmost bit. Details on other representations of the state can be found in Appendix A.

3.2. Constant-Addition Layer p_C

The constant c_i of round i of the Ascon permutation $Ascon\text{-}p[rnd]$ (instantiated with rnd rounds) for $rnd \leq 16$ and $0 \leq i \leq rnd - 1$ is defined as

$$c_i = \texttt{const}_{16-rnd+i}, \tag{3}$$

where $\texttt{const}_0, \ldots, \texttt{const}_{15}$ are defined in Table 5. The constant-addition layer p_C adds a 64-bit round constant c_i to S_2 in round i, for $i \geq 0$,

$$S_2 = S_2 \oplus c_i. \tag{4}$$

Table 5. The constants $\texttt{const}_i$ to derive round constants of the Ascon permutations

i	$\texttt{const}_i$	i	$\texttt{const}_i$
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

Since the first 56 bits of the constants are zero, in practice, this is equivalent to applying the constant to only the least significant eight bits of S_2, as shown in Figure 1.

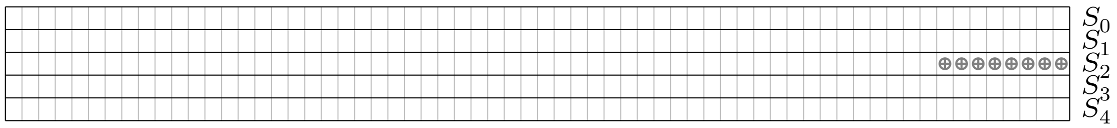

Figure 1. Application of constant-addition layer p_C to Ascon state

3.3. Substitution Layer p_S

The substitution layer p_S updates the state $\mathcal{S}$ with 64 parallel applications of the 5-bit substitution box SBox as

$$(s_{(0,j)}, s_{(1,j)}, \dots, s_{(4,j)}) = \text{SBox}(s_{(0,j)}, s_{(1,j)}, \dots, s_{(4,j)}) \tag{5}$$

for $0 \le j < 64$, as shown in Figure 2.

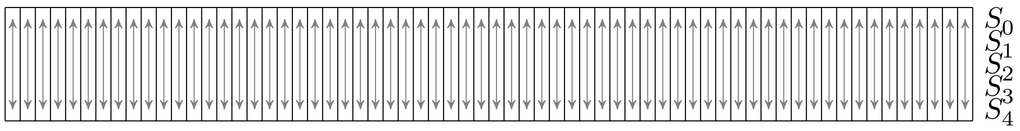

Figure 2. Application of substitution layer p_S to Ascon state

The 5-bit SBox is computed as

$$(y_0, ..., y_4) = \text{SBox}(x_0, ..., x_4), \tag{6}$$

where

$$\begin{aligned}
y_0 &= x_4x_1 \oplus x_3 \oplus x_2x_1 \oplus x_2 \oplus x_1x_0 \oplus x_1 \oplus x_0,\\
y_1 &= x_4 \oplus x_3x_2 \oplus x_3x_1 \oplus x_3 \oplus x_2x_1 \oplus x_2 \oplus x_1 \oplus x_0,\\
y_2 &= x_4x_3 \oplus x_4 \oplus x_2 \oplus x_1 \oplus 1,\\
y_3 &= x_4x_0 \oplus x_4 \oplus x_3x_0 \oplus x_3 \oplus x_2 \oplus x_1 \oplus x_0,\\
y_4 &= x_4x_1 \oplus x_4 \oplus x_3 \oplus x_1x_0 \oplus x_1.
\end{aligned} \tag{7}$$

SBox may also be implemented as a lookup table, as shown in Table 6. The circuit representation of the SBox is given in Figure 3.

Table 6. Lookup table representation of SBox

x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
SBox(x)	4	b	1f	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

Note that 5-bit inputs are represented in hexadecimal (e.g., x =1 corresponds to $(0,0,0,0,1)$).

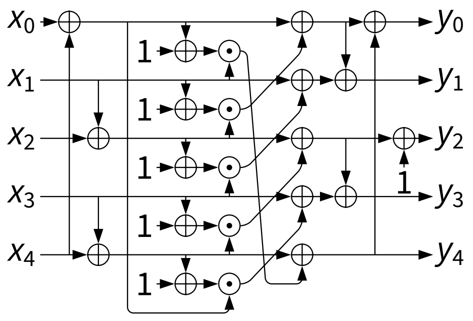

Figure 3. Circuit representation of the 5-bit S-box SBox

3.4. Linear Diffusion Layer p_L

The linear diffusion layer p_L provides diffusion within each 64-bit word S_i, as shown in Figure 4.

S_0
S_1
S_2
S_3
S_4

Figure 4. Application of linear diffusion layer p_L to Ascon state

This layer applies the linear functions Σ_i to their corresponding state words as $S_i \leftarrow \Sigma_i(S_i)$ for $0 \leq i \leq 4$, where each Σ_i is defined as:

$$\Sigma_0(S_0) = S_0 \oplus (S_0 \ggg 19) \oplus (S_0 \ggg 28) \quad (8)$$

$$\Sigma_1(S_1) = S_1 \oplus (S_1 \ggg 61) \oplus (S_1 \ggg 39) \quad (9)$$

$$\Sigma_2(S_2) = S_2 \oplus (S_2 \ggg 1) \oplus (S_2 \ggg 6) \quad (10)$$

$$\Sigma_3(S_3) = S_3 \oplus (S_3 \ggg 10) \oplus (S_3 \ggg 17) \quad (11)$$

$$\Sigma_4(S_4) = S_4 \oplus (S_4 \ggg 7) \oplus (S_4 \ggg 41) \quad (12)$$

4. Authenticated Encryption Scheme: `Ascon-AEAD128`

This section specifies the AEAD scheme `Ascon-AEAD128`, details implementation options (e.g., truncation and nonce-masking), lists AEAD requirements, and provides security properties.

4.1. Specification of `Ascon-AEAD128`

`Ascon-AEAD128` consists of the encryption algorithm `Ascon-AEAD128.enc` (specified in Sec. 4.1.1) and the decryption algorithm `Ascon-AEAD128.dec` (specified in Sec. 4.1.2).

`Ascon-AEAD128.enc` takes a 128-bit secret key K, a 128-bit nonce N, variable-length associated data A, and variable-length plaintext P as inputs and outputs ciphertext C (where $|C| = |P|$) and 128-authentication tag T (see Sec. 4.2.1 for the truncation option):

$$\texttt{Ascon-AEAD128.enc}(K, N, A, P) = (C, T) \tag{13}$$

`Ascon-AEAD128.dec` takes key K, nonce N, associated data A, ciphertext C, and authentication tag T as inputs and outputs P if the tag is valid:

$$\texttt{Ascon-AEAD128.dec}(K, N, A, C, T) = \begin{cases} P & \text{if the tag } T \text{ is valid} \\ \texttt{fail} & \text{otherwise} \end{cases} \tag{14}$$

4.1.1. Encryption

This section outlines the encryption algorithm of `Ascon-AEAD128`, which comprises four phases: initialization, associated data processing, plaintext processing, and finalization (see Figure 5).

Note that the rate and capacity of `Ascon-AEAD128` are 128 and 192 bits, respectively.

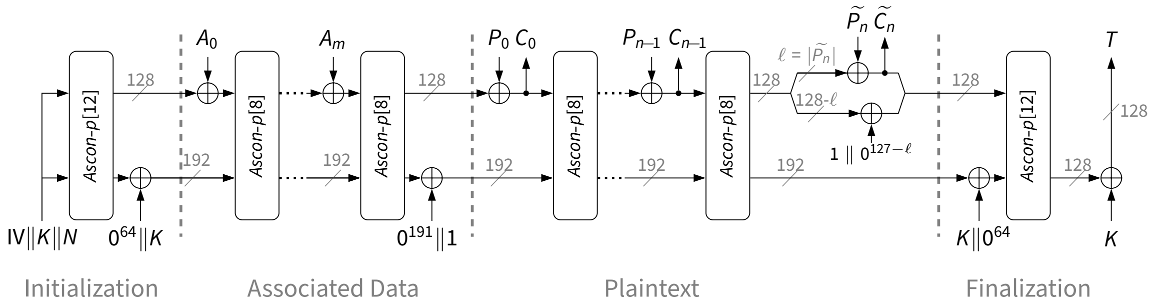

Figure 5. `Ascon-AEAD128` encryption

The pseudocode of `Ascon-AEAD128.enc` is provided in Algorithm 3.

Algorithm 3 `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 T

```
$IV \leftarrow$ 0x00001000808c0001                          ▷ Initialization
$\mathcal{S} \leftarrow IV \| K \| N$
$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S})$
$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{192} \| K)$

if $|A| > 0$ then                                             ▷ Processing associated data
    $A_0, \ldots, A_{m-1}, \widetilde{A_m} \leftarrow$ parse$(A, 128)$
    $A_m \leftarrow$pad$(\widetilde{A_m}, 128)$
    for $i = 0$ to $m$ do
        $\mathcal{S} \leftarrow Ascon\text{-}p[8]((\mathcal{S}_{[0:127]} \oplus A_i) \| \mathcal{S}_{[128:319]})$
    end for
end if
$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{319} \| 1)$

$P_0, \ldots, P_{n-1}, \widetilde{P_n} \leftarrow$ parse$(P, 128)$           ▷ Processing plaintext
$\ell \leftarrow |\widetilde{P_n}|$
for $i = 0$ to $n - 1$ do
    $\mathcal{S}_{[0:127]} \leftarrow \mathcal{S}_{[0:127]} \oplus P_i$
    $C_i \leftarrow \mathcal{S}_{[0:127]}$
    $\mathcal{S} \leftarrow Ascon\text{-}p[8](\mathcal{S})$
end for
$\mathcal{S}_{[0:127]} \leftarrow \mathcal{S}_{[0:127]} \oplus$pad$(\widetilde{P_n}, 128)$
$\widetilde{C_n} \leftarrow \mathcal{S}_{[0:\ell-1]}$
$C \leftarrow C_0 \| \ldots \| C_{n-1} \| \widetilde{C_n}$

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

1. **Initialization of the state.** Given a 128-bit K and a 128-bit N, the 320-bit internal state $\mathcal{S}$ is initialized as the concatenation of IV, K, and N:

$$\mathcal{S} \leftarrow IV \| K \| N, \tag{15}$$

where the initialization value IV is 0x00001000808c0001 (see Appendix B for details on determining the IV and Appendix A for implementation notes regarding initialization). Next, $\mathcal{S}$ is updated using the permutation $Ascon\text{-}p[12]$ as

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}) \tag{16}$$

and followed by XORing the secret key K into the last 128 bits of internal state:

$$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{192} \| K). \tag{17}$$

2. **Processing associated data.** This step has two parts, including absorbing the associated data (when it is non-empty) and applying the domain separation bit to the state.

   - If the AD is non-empty (i.e., $|A| > 0$): The associated data A is parsed into blocks as

   $$A_0,\ A_1,\ \ldots,\ A_{m-1},\ \widetilde{A_m} \leftarrow \text{parse}(A, 128), \tag{18}$$

   where $m = \lfloor |A|/128 \rfloor$ and $|A_i| = 128$ bits for $0 \le i \le m-1$ and $0 \le |\widetilde{A_m}| < 128$, as explained in Algorithm 1. The last block $\widetilde{A_m}$ can be empty. Next, $\widetilde{A_m}$ is padded as

   $$A_m \leftarrow \text{pad}(\widetilde{A_m}, 128) = \widetilde{A_m} || 1 \; || \; 0^{127-|\widetilde{A_m}|} \tag{19}$$

   so that $|A_m| = 128$, as explained in Algorithm 2.

   Each associated data block A_i $(0 \le i \le m)$ is absorbed into the first 128 bits of state as

   $$\mathcal{S}_{[0:127]} \leftarrow \mathcal{S}_{[0:127]} \oplus A_i, \tag{20}$$

   and the permutation $Ascon\text{-}p[8]$ is applied to the state as

   $$\mathcal{S} \leftarrow Ascon\text{-}p[8](\mathcal{S}). \tag{21}$$

   The final step of processing associated data is to update the state with a constant

   $$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{319} \| 1) \tag{22}$$

   that provides domain separation.

- If the AD is empty (i.e., $|A| = 0$): Only the final step described in (22) is applied.

3. **Processing plaintext.** Plaintext P (including empty plaintext) is parsed into blocks as

$$P_0,\, P_1,\, \ldots,\, P_{n-1}, \widetilde{P_n} \leftarrow \text{parse}(P, 128), \tag{23}$$

where $n = \lfloor |P|/128 \rfloor$, $|P_i| = 128$ for $0 \le i \le n-1$, and $|\widetilde{P_n}| = \ell$, $0 \le \ell < 128$ using Algorithm 1. When $|P| \bmod 128 = 0$, the last block $\widetilde{P_n}$ is empty.

For each P_i, $0 \le i \le n-1$, the state $\mathcal{S}$ is updated as

$$\mathcal{S}_{[0:127]} \leftarrow \mathcal{S}_{[0:127]} \oplus P_i, \tag{24}$$

followed by generating the corresponding ciphertext block C_i as

$$C_i \leftarrow \mathcal{S}_{[0:127]}, \tag{25}$$

and the permutation $Ascon\text{-}p[8]$ is applied to update the state as

$$\mathcal{S} \leftarrow Ascon\text{-}p[8](\mathcal{S}). \tag{26}$$

For the last block $\widetilde{P_n}$, the state is updated as

$$\mathcal{S}_{[0:127]} \leftarrow \mathcal{S}_{[0:127]} \oplus \text{pad}(\widetilde{P_n}, 128), \tag{27}$$

and the last ciphertext block $\widetilde{C_n}$ is obtained as

$$\widetilde{C_n} \leftarrow \mathcal{S}_{[0:\ell-1]}. \tag{28}$$

The ciphertext C is constructed by concatenating the ciphertext blocks as

$$C \leftarrow C_0 \| \ldots \| C_{n-1} \| \widetilde{C_n}. \tag{29}$$

4. **Finalization and tag generation.** During finalization, the key is first loaded to the state $\mathcal{S}$ as

$$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{128} \| K \| 0^{64}), \tag{30}$$

and the state $\mathcal{S}$ is then updated using the permutation $Ascon\text{-}p[12]$ as

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}). \tag{31}$$

Finally, the tag T is generated by XORing the key with the last 128 bits of the state:

$$T \leftarrow S_{[192:319]} \oplus K. \tag{32}$$

The encryption algorithm returns the ciphertext C and the tag T.

4.1.2. Decryption

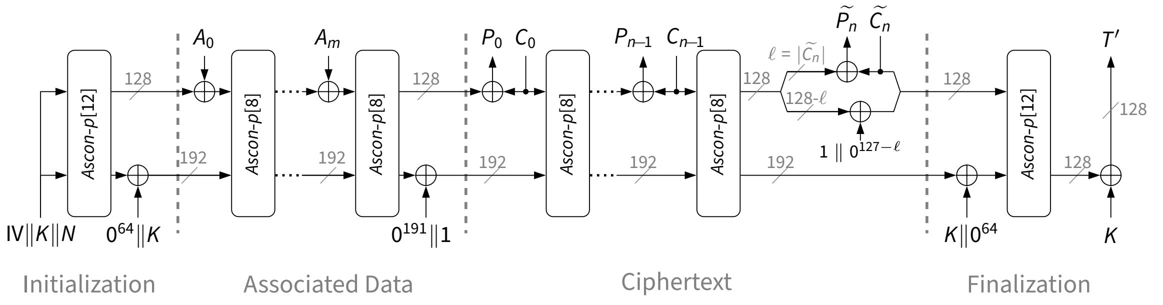

Figure 6. `Ascon-AEAD128` decryption

This section describes each of the phases for decryption with `Ascon-AEAD128.dec`. Decryption in `Ascon-AEAD128` consists of four phases: initialization, associated data processing, ciphertext processing, and finalization. Decryption in `Ascon-AEAD128` is similar to encryption; only the last two phases differ from the encryption mode.

The pseudocode of `Ascon-AEAD128.dec` is provided in Algorithm 4.

1. **Initialization of the state.** Given a 128-bit K and 128-bit N, the 320-bit internal state $\mathcal{S}$ is initialized as the concatenation of IV, K, and N:

$$\mathcal{S} \leftarrow IV \| K \| N, \tag{33}$$

   where the initial value IV is `0x00001000808c0001` (see Appendix B for details on determining the IV and Appendix A for implementation notes regarding initialization). Next, $\mathcal{S}$ is updated using the permutation $Ascon\text{-}p[12]$ as

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}) \tag{34}$$

   and followed by XORing the secret key into the last 128 bits of the state as

$$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{192} \| K). \tag{35}$$

   This step is exactly the same as Step 1 of the encryption function in Sec. 4.1.1.

2. **Processing associated data.** This step has two parts, including absorbing the associated data (when it is non-empty) and applying the domain separation bit to the state.

   - If the AD is non-empty (i.e., $|A| > 0$): The associated data A is parsed into blocks as

$$A_0,\ A_1,\ \ldots,\ A_{m-1},\ \widetilde{A_m} \leftarrow \text{parse}(A, 128), \tag{36}$$

Algorithm 4 `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`

```
$IV \leftarrow$ 0x00001000808c0001                                ▷ Initialization
$\mathcal{S} \leftarrow IV \| K \| N$
$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S})$
$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{192} \| K)$

if $|A| > 0$ then                                                 ▷ Processing associated data
    $A_0, \ldots, A_{m-1}, \widetilde{A_m} \leftarrow$ parse$(A, 128)$
    $A_m \leftarrow$pad$(\widetilde{A_m}, 128)$
    for $i = 0$ to $m$ do
        $\mathcal{S} \leftarrow Ascon\text{-}p[8]((\mathcal{S}_{[0:127]} \oplus A_i) \| \mathcal{S}_{[128:319]})$
    end for
end if
$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{319} \| 1)$

$C_0, \ldots, C_{n-1}, \widetilde{C_n} \leftarrow$ parse$(C, 128)$                  ▷ Processing ciphertext
for $i = 0$ to $n-1$ do
    $P_i \leftarrow \mathcal{S}_{[0:127]} \oplus C_i$
    $\mathcal{S}_{[0:127]} \leftarrow C_i$
    $\mathcal{S} \leftarrow Ascon\text{-}p[8](\mathcal{S})$
end for
$\ell = |\widetilde{C_n}|$
$\widetilde{P_n} \leftarrow \mathcal{S}_{[0:\ell-1]} \oplus \widetilde{C_n}$
$\mathcal{S}_{[\ell:127]} \leftarrow \mathcal{S}_{[\ell:127]} \oplus (1||0^{127-\ell})$
$\mathcal{S}_{[0:\ell-1]} \leftarrow \widetilde{C_n}$

$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S} \oplus (0^{128} \| K \| 0^{64}))$        ▷ Finalization
$T' \leftarrow \mathcal{S}_{[192:319]} \oplus K$
if $T' == T$ then
    $P \leftarrow P_0 \| \ldots \| P_{n-1} \| \widetilde{P_n}$
    return $P$
else
    return fail
end if
```

where $m = \lfloor |A|/128 \rfloor$ and $|A_i| = 128$ bits for $0 \le i \le m-1$ and $0 \le |\widetilde{A_m}| < 128$, as explained in Algorithm 1. The last block $\widetilde{A_m}$ can be empty.

$\widetilde{A_m}$ is further processed by padding to a full $r = 128$-bit block using Algorithm 2 as

$$A_m \leftarrow \text{pad}(\widetilde{A_m}, 128) = \widetilde{A_m} || 1 \parallel 0^{127-|\widetilde{A_m}|}. \tag{37}$$

The associated data blocks A_i's ($0 \le i \le m$) are absorbed into the state $\mathcal{S}$ as follows:

$$\mathcal{S}_{[0:127]} \leftarrow (\mathcal{S}_{[0:127]} \oplus A_i), \tag{38}$$

and the permutation $Ascon\text{-}p[8]$ is applied to the state as

$$\mathcal{S} \leftarrow Ascon\text{-}p[8](\mathcal{S}). \tag{39}$$

The final step of processing associated data is to update the state to

$$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{319} \| 1) \tag{40}$$

for domain separation.

- If the AD is empty (i.e., $|A| = 0$): Only the final step described in Equation (40) is applied.

This step is exactly the same as Step 2 of the encryption function in Sec. 4.1.1.

3. **Processing the ciphertext.** Ciphertext C is parsed into blocks as

$$C_0,\ C_1,\ \ldots, C_{n-1},\ \widetilde{C_n} \leftarrow \text{parse}(C, 128), \tag{41}$$

where $n = \lfloor |C|/128 \rfloor$, $|C_i| = 128$ for $0 \le i \le n-1$, $|\widetilde{C_n}| = \ell$ and $0 \le \ell < 128$ using Algorithm 1. Ciphertext C or the last block of ciphertext $\widetilde{C_n}$ can be empty.

For each C_i, $0 \le i \le n-1$, the following steps are applied:

$$P_i \leftarrow \mathcal{S}_{[0:127]} \oplus C_i \tag{42}$$

$$\mathcal{S}_{[0:127]} \leftarrow C_i \tag{43}$$

$$\mathcal{S} \leftarrow Ascon\text{-}p[8](\mathcal{S}). \tag{44}$$

For the last block of the ciphertext $\widetilde{C_n}$ (with length ℓ), the following steps are applied:

$$\widetilde{P_n} \leftarrow \mathcal{S}_{[0,\ell-1]} \oplus \widetilde{C_n} \tag{45}$$

$$\mathcal{S}_{[0,\ell-1]} \leftarrow \widetilde{C_n} \tag{46}$$

$$\mathcal{S}_{[\ell,127]} \leftarrow \mathcal{S}_{[\ell,127]} \oplus (1 || 0^{127-\ell}). \tag{47}$$

Note that when $\widetilde{C_n}$ is an empty block, $\widetilde{P_n}$ is an empty block as well.

The plaintext P is constructed by concatenating the plaintext blocks as

$$P \leftarrow P_0 \| ... \| P_{n-1} \| \widetilde{P_n}. \tag{48}$$

4. **Finalization.** During finalization, the key is loaded into the state $\mathcal{S}$ as

$$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{128} \| K \| 0^{64}), \tag{49}$$

and the state $\mathcal{S}$ is then updated using the permutation Ascon-p[12]as

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}). \tag{50}$$

Finally, the tag is generated by XORing the key with the last 128 bits of the state:

$$T' \leftarrow (S_{[192:319]}) \oplus K. \tag{51}$$

As the last step, the computed T' is compared with the input T. If the two match, the plaintext P is returned. Otherwise, an error message `fail` is returned.

4.2. Implementation Options

4.2.1. Truncation

Some applications may truncate the tag T to a specific length λ $(\leq |T|)$. The truncation function outputs the leftmost λ bits of the tag (i.e., $T_{[0:\lambda-1]}$).

The requirements on the tag lengths are provided in Sec. 4.3.

4.2.2. Nonce Masking

In this option, an additional 128-bit key is used to mask the input nonce. Let $K = (K_1 \| K_2)$ be a 256-bit key, where $|K_1| = |K_2| = 128$. `Ascon-AEAD128` with nonce masking is processed as follows:

$$\mathtt{E}(K_1 \| K_2, N, A, P) = \mathtt{Ascon\text{-}AEAD128.enc}(K_1, N \oplus K_2, A, P), \tag{52}$$

$$\mathtt{D}(K_1 \| K_2, N, A, C, T) = \mathtt{Ascon\text{-}AEAD128.dec}(K_1, N \oplus K_2, A, C, T) \tag{53}$$

`Ascon-AEAD128` with nonce masking should only be used when context-commitment security [2] [12] and related-key security [13] are not concerns. This is because the encryption of `Ascon-AEAD128` with nonce masking always outputs the same (C,T) pair for two different input tuples $(K \| K', N, A, P)$ and $(K \| K'', N', A, P)$, where $N \oplus K' = N' \oplus K''$.

When the output tag is not truncated, this option maintains its 128-bit security strength in both single-key and multi-key settings [14] (see Section 4.4.2).

4.3. AEAD Requirements

This section specifies requirements for `Ascon-AEAD128`.

R1. Key generation. The secret key and the nonce-masking key (if available) **shall** be generated following the recommendations for cryptographic key generation specified in SP 800-133 [15] and using an approved random bit generator that supports at least a 128-bit security strength. The keys **shall not** be used for other purposes.

R2. Secrecy of key. The `Ascon-AEAD128` key **shall** be kept secret. When the nonce masking option is implemented, the masked nonce (i.e., $N \oplus K_2$ in Equation (52)) **shall** also be kept secret.

R3. Use of unique nonce. Nonces **shall** be distinct for each encryption algorithm for a given key to ensure that identical plaintexts encrypted multiple times produce different ciphertexts.

R4. Minimum length of truncated tag. When an application uses truncated tags, the bit length of the truncated tags **shall** be at least 32 bits and **shall** only select a tag length less than 64 bits after a careful risk analysis is performed. The tag length **shall** be the same across the lifespan of the key.

R5. Limit on the maximum number of decryption failures. When the tag bit length λ satisfies $64 \leq \lambda \leq 128$, the probability of a forgery is low enough that decryption failures up to $2^{\lambda-32}$ can be tolerated without compromising security. Therefore, the maximum number of decryption failures under a fixed key **shall not** exceed $2^{\lambda-32}$. For shorter tags, with $32 \leq \lambda < 64$, the forgery probability is higher, in these cases, the number of allowable decryption failures **should** be limited to 1. However, if a careful risk analysis shows that the system's overall security goal remains satisfied, this limit may be relaxed – up to the same bound of $2^{\lambda-32}$.

R6. Data limit. The total amount of data processed during encryption and decryption, including the nonce, **shall not** exceed 2^{54} bytes for a given key.

[2]In AEAD schemes, context commitment is a security property that ensures a ciphertext cannot be decrypted successfully under two different, adversarially-chosen contexts – where context includes a secret key, nonce, and associated data.

R7. Key update. The key **shall** be updated to a new key once the total amount of input data reaches the limit of 2^{54} bytes, and **should** be updated when the number of decryption failures reaches its limit.

4.4. Security Properties

This section provides the security properties of $\texttt{Ascon-AEAD128}$ in various scenarios, including single-key and multi-key settings, nonce-respecting and nonce-misuse settings, and with or without the truncation option.

In the single-key setting, the security of the scheme is analyzed under the assumption that the scheme uses a single key; in contrast, in the multi-key setting, multiple independent keys are used, and the adversary may interact with many instances of the scheme, each with a different key. The security of the $\texttt{Ascon-AEAD128}$ mode in both single-key and multi-key settings was evaluated in [14, 16–19]. The committing security of the $\texttt{Ascon-AEAD128}$ mode was also evaluated in [20, 21].

4.4.1. Single-Key Setting

$\texttt{Ascon-AEAD128}$ with no tag truncation provides a 128-bit security strength in the single-key and nonce-respecting setting for the confidentiality of the plaintext (except for its length) and the integrity of the tuple (nonce, associated data, ciphertext, tag), where the total number of input bytes is limited to 2^{54} (i.e., 2^{50} blocks).

Impact of truncation. For a tag of length λ, a forgery attempt succeeds with a probability of $2^{-\lambda}$. Once a forgery is successful, the confidentiality of the plaintext may be compromised, as the decryption algorithm could reveal some information about the plaintext. Therefore, in the single-key setting, $\texttt{Ascon-AEAD128}$ with tag length λ provides λ-bit security for both confidentiality and integrity in the nonce-respecting setting.Note that even if a forgery attempt is successful, the probability of another successful forgery is $2^{-\lambda}$, provided that the secret key remains uncompromised. This also holds for the nonce masking option.

4.4.2. Multi-Key Setting

When u keys are independently selected for an application, $\texttt{Ascon-AEAD128}$ with no tag truncation provides a ($128 - \log_2 u$)-bit security strength in the nonce-respecting setting for the confidentiality of the plaintext and the integrity of the tuple (nonce, associated data, ciphertext, tag). Note that, in the nonce-respecting setting, an attacker may select the same nonce for use with different keys, but is not permitted to reuse a nonce with the same key.

When the same nonce is used with u keys, an attacker may discover one of the u keys with a time complexity of $2^{128-\log_2 u}$, thereby compromising both confidentiality and integrity [14, 17–19].

To improve security in a multi-key setting, the nonce-masking implementation option with no truncation (see Sec. 4.2.2) can be used. This option provides 128-bit security (rather than $128-\log_2 u$) for confidentiality and integrity.

Impact of truncation. In the multi-key setting, `Ascon-AEAD128` with tag length λ provides min$\{128-\log_2 u, \lambda\}$-bit security for both confidentiality and integrity in the nonce-respecting setting. Additionally, when using the nonce-masking option with tag length λ, it provides λ-bit security for both confidentiality and integrity in the same setting. Note that, similar to the single-key case, even if a forgery attempt is successful, the probability of another successful forgery is $2^{-\lambda}$, provided that the secret key is uncompromised.

4.4.3. Nonce-Misuse Setting

The confidentiality of plaintext both in `Ascon-AEAD128` and `Ascon-AEAD128` with nonce masking can be compromised if a nonce or, more specifically, (nonce, associated data) pair, is reused with the same secret key. However, these algorithms are designed to provide some resilience against unintentional nonce reuse.

When (N, A) pairs are distinct for encryption per key: In the u-key setting, where $u = 1$ corresponds to a single key and $u > 1$ to multiple independent keys, the confidentiality and integrity guarantees of `Ascon-AEAD128` and the nonce-masking option with a λ-bit tag are as follows:

- `Ascon-AEAD128` provides min$\{128-\log_2 u, \lambda\}$ bits of security for both confidentiality and integrity.
- Nonce-masking option provides λ-bit security for both confidentiality and integrity.

These guarantees assume that each (N, A) pair is used at most once per key, and that any given nonce is reused for encryption with the same key no more than 2^8 times. Additionally, even after a successful forgery, the probability of another successful forgery attempt remains at most $2^{-\lambda}$, provided that none of the multiple keys is compromised. The resulting security levels under these conditions are summarized in Table 7.

Table 7. Security strength of `Ascon-AEAD128` with λ-bit tag in the u-key setting, where (N, A) pairs are distinct for encryption per key

Security	Security strength in bits	Total number of repetitions of a nonce
Confidentiality of plaintext	min$\{128-\log_2 u, \lambda\}$	$\leq 2^8$
Integrity of (N, A, C, T)	min$\{128-\log_2 u, \lambda\}$	$\leq 2^8$

When each (N, A) pair are reused up to 2^8 times for encryption per key: In the u-key setting, where u denotes the number of independent keys, the integrity security guarantees of `Ascon-AEAD128` and the nonce-masking option with a λ-bit tag are as follows:

- `Ascon-AEAD128` provides min$\{128 - \log_2 u, \lambda\}$ bits of integrity security.
- Nonce-masking option provides λ-bit integrity security.

These guarantees hold under the condition that each (N, A) pair is reused at most 2^8 times for encryption with the same key. The corresponding integrity security levels are summarized in Table 8. Furthermore, for both `Ascon-AEAD128` and the nonce-masking option, even after a successful forgery, the probability that a subsequent forgery attempt succeeds remains at most $2^{-\lambda}$, provided that none of the multiple keys has been compromised.

Table 8. Integrity security strength of `Ascon-AEAD128` with u keys in the nonce-misuse setting

Security	**Security strength in bits**	**Total number of repetitions of any (N, A) pair**
Integrity of (N, A, C, T)	min$\{128 - \log_2 u, \lambda\}$	$\leq 2^8$

5. Hash and eXtendable-Output Functions (XOFs)

Hash and XOF algorithms are built on the $Ascon\text{-}p[12]$ permutation in a sponge-based mode. This section specifies three functions:

1. Hash function `Ascon-Hash256`, which produces a 256-bit digest
2. `Ascon-XOF128` function that produces arbitrary length outputs
3. Customized XOF `Ascon-CXOF128`, which also produces arbitrary length outputs

The designs of these functions differ from the design of `Ascon-AEAD128` in three important ways. First, they employ traditional sponge-based modes that only extract output from the state after all input data has been absorbed. Second, the rate of these functions is reduced to 64 bits — half the rate used in `Ascon-AEAD128`. Finally, the hash and XOF algorithms rely only on the $Ascon\text{-}p[12]$ permutation, whereas `Ascon-AEAD128` employs both $Ascon\text{-}p[12]$ and $Ascon\text{-}p[8]$.

In `Ascon-XOF128`, when different output lengths are specified for the same input message, the shorter output is a prefix of the longer one. If this prefix property is undesirable in a given application, domain separation offers a more robust solution. For instance, `Ascon-CXOF128` enables domain separation by allowing output lengths to be encoded into the user-defined customization string.

5.1. Specification of `Ascon-Hash256`

The mode of operation used by `Ascon-Hash256` and `Ascon-XOF128` is shown in Figure 7. This mode comprises three main steps: initialization, absorbing the message, and squeezing the output. The length of the output L is 256 for `Ascon-Hash256` and $L > 0$ for `Ascon-XOF128`.

Note that the rate and capacity of `Ascon-Hash256` are 64 and 256 bits, respectively.

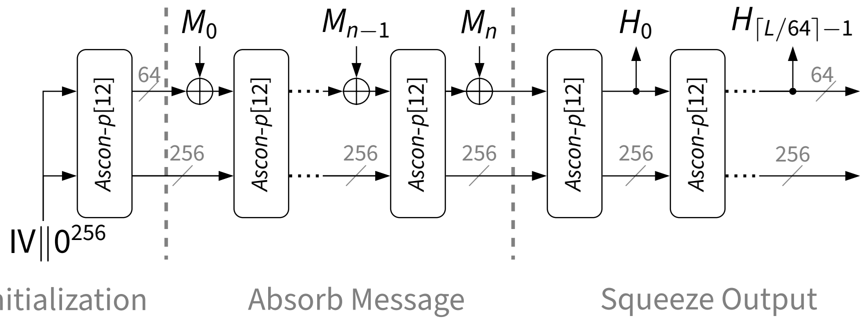

Figure 7. Structure of `Ascon-Hash256` and `Ascon-XOF128`

`Ascon-Hash256` takes a variable length message M as input and produces a 256-bit digest. The full specification of `Ascon-Hash256` can be found in Algorithm 5 and operates as follows:

1. **Initialization.** The 320-bit internal state of `Ascon-Hash256` is initialized with the concatenation of the 64-bit $IV =$ `0x0000080100cc0002` and 256 zeroes, followed by the $Ascon\text{-}p[12]$ permutation as

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](IV \| 0^{256}). \tag{54}$$

2. **Absorbing the message.** The absorbing phase behaves similarly to the associated data processing of `Ascon-AEAD128`. The message is parsed into blocks, as

$$M_0, \ldots, M_{n-1}, \widetilde{M_n} \leftarrow \mathsf{parse}(M, 64), \tag{55}$$

where $|M_i| = 64$ bits for $0 \le i \le n-1$ and $0 \le |\widetilde{M_n}| \le 63$. The last block $\widetilde{M_n}$ can be empty. Next, $\widetilde{M_n}$ is padded to create a full block M_n:

$$M_n \leftarrow \mathsf{pad}(\widetilde{M_n}, 64). \tag{56}$$

Each message block M_i is XORed with the state as

$$\mathcal{S}_{[0:63]} \leftarrow \mathcal{S}_{[0:63]} \oplus M_i. \tag{57}$$

For all message blocks except the final block M_n,the XOR operation is immediately followed by applying $Ascon\text{-}p[12]$ to the state.

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}) \tag{58}$$

3. **Squeezing the hash.** The squeezing phase begins after M_n is absorbed with an application of $Ascon\text{-}p[12]$ to the state:

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}). \tag{59}$$

The value of $\mathcal{S}_{[0:63]}$ is then taken as the 64-bit hash block H_i, and the state is again updated by $Ascon\text{-}p[12]$:

$$H_i \leftarrow \mathcal{S}_{[0:63]} \tag{60}$$

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}). \tag{61}$$

Steps (60) and (61) are repeated alternately until hash blocks H_0, H_1, and H_2 have been extracted. The final hash block is then extracted but is not followed by the Ascon-p[12] permutation:

$$H_3 \leftarrow \mathcal{S}_{[0:63]}. \tag{62}$$

The resulting 256-bit digest is the concatenation of hash blocks as

$$H \leftarrow H_0 \Big\| H_1 \Big\| H_2 \Big\| H_3. \tag{63}$$

Algorithm 5 `Ascon-Hash256`(M)

Input: Bitstring $M \in \{0,1\}^*$
Output: Digest $H \in \{0,1\}^{256}$

```
$IV \leftarrow$0x0000080100cc0002                                  ▷ Initialization
$\mathcal{S} \leftarrow Ascon\text{-}p[12](IV \| 0^{256})$

$M_0, \ldots, M_{n-1}, \widetilde{M_n} \leftarrow$ parse$(M, 64)$                 ▷ Absorbing
$M_n \leftarrow$ pad$(\widetilde{M_n}, 64)$
for $i = 0$ to $n-1$ do
    $\mathcal{S}_{[0:63]} \leftarrow \mathcal{S}_{[0:63]} \oplus M_i$
    $\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S})$
end for
$\mathcal{S}_{[0:63]} \leftarrow \mathcal{S}_{[0:63]} \oplus M_n$

$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S})$                             ▷ Squeezing
for $i = 0$ to $2$ do
    $H_i \leftarrow \mathcal{S}_{[0:63]}$
    $\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S})$
end for
$H_3 \leftarrow \mathcal{S}_{[0:63]}$

$H \leftarrow H_0 \| H_1 \| H_2 \| H_3$
return $H$
```

5.2. Specification of `Ascon-XOF128`

`Ascon-XOF128` is similar to `Ascon-Hash256` but differs in three aspects:

1. `Ascon-XOF128` accepts an additional input $L > 0$, which specifies the desired output length in bits.
2. The number of blocks that are squeezed is $\lceil L/64 \rceil$.

3. The initial values used in `Ascon-XOF128` and `Ascon-Hash256` are different.

The 128 in the name `Ascon-XOF128` refers to the target security strength, not the output size. Note that the rate and capacity of `Ascon-XOF128` are 64 and 256 bits, respectively.

`Ascon-XOF128` is specified by Algorithm 6 and is described as follows:

1. **Initialization.** The 320-bit internal state of `Ascon-XOF128` is initialized with the concatenation of the 64-bit $IV =$ 0x0000080000cc0003 and 256 zeroes, followed by the $Ascon\text{-}p[12]$ permutation:

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](IV \| 0^{256}). \tag{64}$$

2. **Absorbing the message.** The absorbing phase behaves the same as that of `Ascon-Hash256`. The message is parsed into blocks as

$$M_0, \ldots, M_{n-1}, \widetilde{M_n} \leftarrow \text{parse}(M, 64). \tag{65}$$

where $|M_i| = 64$ bits for $0 \leq i \leq n-1$ and $0 \leq |\widetilde{M_n}| \leq 63$. Partial block $\widetilde{M_n}$ is then padded to a full block M_n as

$$M_n \leftarrow \text{pad}(\widetilde{M_n}, 64). \tag{66}$$

Each message block M_i is absorbed by XORing the block into the state as

$$\mathcal{S}_{[0:63]} \leftarrow \mathcal{S}_{[0:63]} \oplus M_i. \tag{67}$$

For all message blocks except the final block, the XOR operation is immediately followed by an application of $Ascon\text{-}p[12]$ to the state:

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}). \tag{68}$$

3. **Squeezing the outputs.** To obtain the requested L output bits, $h = \lceil L/64 \rceil$ blocks must be extracted from the state. The squeezing phase begins with an application of $Ascon\text{-}p[12]$ to the state:

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}). \tag{69}$$

The value of $\mathcal{S}_{[0:63]}$ is then taken as output block H_i, and the state is again updated by $Ascon\text{-}p[12]$:

$$H_i \leftarrow \mathcal{S}_{[0:63]} \tag{70}$$

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}). \tag{71}$$

Steps (70) and (71) are repeated alternately until output blocks $H_0, \ldots, H_{h-1}$ have been squeezed. The final block is then squeezed without an additional permutation call:

$$H_h \leftarrow \mathcal{S}_{[0:63]}. \tag{72}$$

Finally, the output blocks are concatenated, and the first L bits are returned as output H:

$$H' \leftarrow H_0 \| \ldots \| H_h \tag{73}$$

$$H \leftarrow H'_{[0:L-1]}. \tag{74}$$

Algorithm 6 `Ascon-XOF128`(M, L)

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

```
$IV \leftarrow$ 0x0000080000cc0003                          ▷ Initialization
$\mathcal{S} \leftarrow Ascon\text{-}p[12](IV \| 0^{256})$

$M_0, \ldots, M_{n-1}, \widetilde{M_n} \leftarrow \mathsf{parse}(M, 64)$           ▷ Absorbing
$M_n \leftarrow \mathsf{pad}(\widetilde{M_n}, 64)$
for $i = 0$ to $n-1$ do
    $\mathcal{S}_{[0:63]} \leftarrow \mathcal{S}_{[0:63]} \oplus M_i$
    $\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S})$
end for
$\mathcal{S}_{[0:63]} \leftarrow \mathcal{S}_{[0:63]} \oplus M_n$

$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S})$                        ▷ Squeezing
$h \leftarrow \lceil L/64 \rceil - 1$
for $i = 0$ to $h-1$ do
    $H_i \leftarrow \mathcal{S}_{[0:63]}$
    $\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S})$
end for
$H_h \leftarrow \mathcal{S}_{[0:63]}$

$H' \leftarrow H_0 \| \ldots \| H_h$
$H \leftarrow H'_{[0:L-1]}$
return $H$
```

5.3. Specification of `Ascon-CXOF128`

This section defines `Ascon-CXOF128`, which is a customized variant of `Ascon-XOF128` that extends its functionality by incorporating a user-defined customization string. This feature enables domain separation, ensuring that two instances of `Ascon-CXOF128` with the same input message but different customization strings produce distinct outputs.

In addition to the message M and output length L, `Ascon-CXOF128` takes the customization string Z as input. After initialization, the length of Z, in bits, is assigned to the 64-bit block Z_0 as

$$Z_0 = \text{int64}(|Z|). \tag{75}$$

Then, Z is parsed into blocks as

$$Z_1, \dots, Z_{m-1}, \widetilde{Z_m} \leftarrow \text{parse}(Z, 64), \tag{76}$$

where $|Z_i| = 64$ bits for $0 \le i \le m-1$ and $0 \le |\widetilde{Z_m}| \le 63$. The partial block $\widetilde{Z_m}$ is then padded to a full block Z_m as

$$Z_m \leftarrow \text{pad}(\widetilde{Z_m}, 64). \tag{77}$$

The customization string Z is prepended to the message blocks as

$$Z_0 \| Z_1 \| \dots \| Z_m \| M_0 \| \dots \| M_{n-1} \| M_n, \tag{78}$$

where the message blocks are generated similarly as in `Ascon-XOF128`.

Although similar to `Ascon-XOF128`, `Ascon-CXOF128` uses a different IV. Hence, the concatenation of the customization string and the message produces different outputs for `Ascon-XOF128` and `Ascon-CXOF128`. The IV for `Ascon-CXOF128` is `0x0000080000cc0004`.

The general structure for `Ascon-CXOF128` is shown in Figure 8 and the full specification is provided in Algorithm 7.

The length of the customization string **shall** be at most 2048 bits (i.e., 256 bytes).

Algorithm 7 `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 $H \in \{0,1\}^L$

```
$IV \leftarrow$ 0x0000080000cc0004                                ▷ Initialization
$\mathcal{S} \leftarrow Ascon\text{-}p[12](IV \| 0^{256})$

$Z_0 \leftarrow \text{int64}(|Z|)$                                ▷ Customization
$Z_1 \ldots, Z_{m-1}, \widetilde{Z_m} \leftarrow \text{parse}(Z, 64)$
$Z_m \leftarrow \text{pad}(\widetilde{Z_m}, 64)$
for $i = 0$ to $m$ do
    $\mathcal{S}_{[0:63]} \leftarrow \mathcal{S}_{[0:63]} \oplus Z_i$
    $\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S})$
end for

$M_0, \ldots, M_{n-1}, \widetilde{M_n} \leftarrow \text{parse}(M, 64)$          ▷ Absorbing
$M_n \leftarrow \text{pad}(\widetilde{M_n}, 64)$
for $i = 0$ to $n-1$ do
    $\mathcal{S}_{[0:63]} \leftarrow \mathcal{S}_{[0:63]} \oplus M_i$
    $\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S})$
end for
$\mathcal{S}_{[0:63]} \leftarrow \mathcal{S}_{[0:63]} \oplus M_n$

$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S})$                        ▷ Squeezing
$h \leftarrow \lceil L/64 \rceil - 1$
for $i = 0$ to $h-1$ do
    $H_i \leftarrow \mathcal{S}_{[0:63]}$
    $\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S})$
end for
$H_h \leftarrow \mathcal{S}_{[0:63]}$

$H' \leftarrow H_0 \| \ldots \| H_h$
$H \leftarrow H'_{[0:L-1]}$
return $H$
```

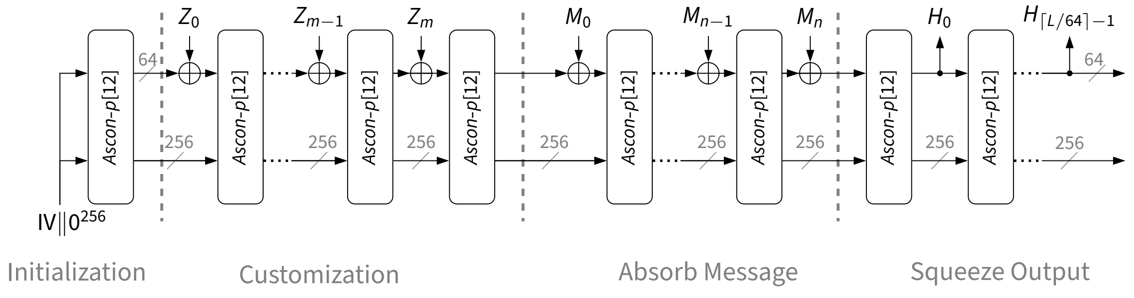

Figure 8. Structure of `Ascon-CXOF128`

5.4. Streaming API for XOF

`Ascon-XOF128` and `Ascon-CXOF128` support incremental processing of input data, without the prior knowledge of complete input or output lengths. This makes them suitable for applications where data is received or processed in blocks. In particular, when using `Ascon-XOF128` and `Ascon-CXOF128`, it is not necessary to know the output length L at the time that the final block is squeezed.

The following three functions can be used to create incremental implementations for `Ascon-XOF128`.

- $ctx \leftarrow$ `Ascon-XOF128`.Init()
  Initializes and returns `Ascon-XOF128` context ctx.
  Restriction: Must be called exactly once before any call to `Absorb` or `Squeeze`.
- $ctx \leftarrow$ `Ascon-XOF128`.Absorb(ctx, *str*)
  Absorbs an arbitrary-length input *str* into the state and updates the context ctx.
  Restriction: May be called multiple times after `Init`, but cannot be called after any call to `Squeeze`.
- (ctx, out) $\leftarrow$ `Ascon-XOF128`.Squeeze(ctx, L)
  Extracts L output bits produced during the squeezing phase of `Ascon-XOF128` and updates context ctx.
  Restriction: May be called multiple times after the absorb phase is complete, but must not be called before `Init`. The first call of `Squeeze` handles the padding of the final message block. Once `Squeeze` is called, no further calls to `Absorb` are permitted.

These functions perform buffering of partial blocks, allowing both input and output to be processed in arbitrary-length segments. Therefore, these functions can then be used to begin execution without knowing the complete message M at the start of the absorption phase or the value L at the time that the final block is squeezed. This API is similar to those proposed for SHAKE-128 and SHAKE-256 in [22]. Similar interfaces can be defined for incremental implementations of `Ascon-AEAD128` and `Ascon-Hash256`.

5.5. Security Strengths

The security strengths of `Ascon-Hash256`, `Ascon-XOF128`, and `Ascon-CXOF128` are summarized in Table 9.

Table 9. Security strengths of `Ascon-Hash256`, `Ascon-XOF128`, and `Ascon-CXOF128` algorithms

Function	**Output size in bits**	**Security strength in bits**		
		Collision	**Preimage**	**2nd Preimage**
`Ascon-Hash256`	256	128	128	128
`Ascon-XOF128`	L	min($L/2$,128)	min(L,128)	min(L,128)
`Ascon-CXOF128`	L	min($L/2$,128)	min(L,128)	min(L,128)

If the message is known to belong to a set $\mathcal{M}$, the preimage resistance is also limited by the size of $\mathcal{M}$. For more information about security strengths against preimage attacks in different scenarios, refer to [19, 23].

6. Conformance

The implementations of `Ascon-AEAD128`, `Ascon-Hash256`, `Ascon-XOF128`, and `Ascon-CXOF128` may be tested for conformance to this standard under the Cryptographic Validation Program [24]. Example test vectors are available from the Cryptographic Algorithm Validation Program (CAVP) [25].

`Ascon-Hash256` is an approved cryptographic hash function; however, its use within the Keyed-Hash Message Authentication Code (HMAC) is not approved in this standard. Similarly, the use of `Ascon-XOF128`and `Ascon-CXOF128` within HMAC is not approved.

`Ascon-XOF128` and `Ascon-CXOF128` are approved XOFs, and their approved uses will be specified in other NIST publications. While some of these uses may overlap with those of approved hash functions, XOFs are not approved as hash functions.

The Ascon permutations, including variants with different numbers of rounds, may be approved for additional applications if corresponding modes of operation are developed and approved within a FIPS or a NIST Special Publication.

References

[1] Dobraunig C, Eichlseder M, Mendel F, Schläffer M (2014) Ascon v1, Submission to Round 1 of the CAESAR competition. Available at https://competitions.cr.yp.to/round1/asconv1.pdf.

[2] Dobraunig C, Eichlseder M, Mendel F, Schläffer M (2015) Ascon v1.1, Submission to Round 2 of the CAESAR competition. Available at https://competitions.cr.yp.to/round2/asconv11.pdf.

[3] Dobraunig C, Eichlseder M, Mendel F, Schläffer M (2016) Ascon v1.2, Submission to Round 3 of the CAESAR competition. Available at https://competitions.cr.yp.to/round3/asconv12.pdf.

[4] National Institute of Standards and Technology (Published 2001; Updated 2023) Advanced Encryption Standard (AES), FIPS 197. https://doi.org/10.6028/NIST.FIPS.197-upd1.

[5] Dworkin MJ (2007) Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC (National Institute of Standards and Technology), Report. DOI:10.6028/NIST.SP.800-38D

[6] National Institute of Standards and Technology (2015) Secure Hash Standard (SHS) (U.S. Department of Commerce), Report. DOI:10.6028/NIST.FIPS.180-4

[7] National Institute of Standards and Technology (2015) SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions (U.S. Department of Commerce), Report. DOI:10.6028/NIST.FIPS.202

[8] Dobraunig C, Eichlseder M, Mendel F, Schläffer M (2021) Ascon v1.2, Submission to Final Round of the NIST Lightweight Cryptography project. Available at https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf.

[9] Sönmez Turan M, McKay KA, Çalık Ç, Chang D, Bassham I Lawrence E (2019) Status Report on the First Round of the NIST Lightweight Cryptography Standardization Process (National Institute of Standards and Technology), Report. DOI:10.6028/NIST.IR.8268

[10] Sönmez Turan M, McKay KA, Chang D, Çalık Ç, Bassham I Lawrence E, Kang J, Kelsey J (2021) Status Report on the Second Round of the NIST Lightweight Cryptography Standardization Process (National Institute of Standards and Technology), Report. DOI:10.6028/NIST.IR.8369

[11] Sönmez Turan M, McKay KA, Chang D, Bassham L, Kang J, Waller N, Kelsey J, Hong D (2023) Status Report on the Final Round of the NIST Lightweight Cryptography Standardization Process (National Institute of Standards and Technology), Report. DOI:10.6028/NIST.IR.8454

[12] Bellare M, Hoang VT (2022) Efficient Schemes for Committing Authenticated Encryption. *Advances in Cryptology - EUROCRYPT 2022 - 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30 - June 3, 2022, Proceedings, Part II*, eds Dunkelman O, Dziembowski S (Springer),

Lecture Notes in Computer Science, Vol. 13276, pp 845–875. DOI:10.1007/978-3-031-07085-3_29

[13] Bellare M, Kohno T (2003) A Theoretical Treatment of Related-Key Attacks: RKA-PRPs, RKA-PRFs, and Applications. *Advances in Cryptology - EUROCRYPT 2003, International Conference on the Theory and Applications of Cryptographic Techniques, Warsaw, Poland, May 4-8, 2003, Proceedings*, ed Biham E (Springer), *Lecture Notes in Computer Science*, Vol. 2656, pp 491–506. DOI:10.1007/3-540-39200-9_31. Available at https://doi.org/10.1007/3-540-39200-9_31

[14] Dobraunig C, Mennink B (2024) Generalized Initialization of the Duplex Construction. *Applied Cryptography and Network Security - 22nd International Conference, ACNS 2024, Abu Dhabi, United Arab Emirates, March 5-8, 2024, Proceedings, Part II*, eds Pöpper C, Batina L (Springer), *Lecture Notes in Computer Science*, Vol. 14584, pp 460–484. DOI:10.1007/978-3-031-54773-7_18

[15] Barker E, Roginsky A, Davis R (2020) Recommendation for Cryptographic Key Generation, (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-133 Rev. 2. DOI:10.6028/NIST.SP.800-133r2.

[16] Chakraborty B, Dhar C, Nandi M (2023) Exact Security Analysis of ASCON. *Advances in Cryptology - ASIACRYPT 2023 - 29th International Conference on the Theory and Application of Cryptology and Information Security, Guangzhou, China, December 4-8, 2023, Proceedings, Part III*, eds Guo J, Steinfeld R (Springer), *Lecture Notes in Computer Science*, Vol. 14440, pp 346–369. DOI:10.1007/978-981-99-8727-6_12

[17] Lefevre C, Mennink B (2025) Generic Security of the Ascon Mode: On the Power of Key Blinding. *Selected Areas in Cryptography – SAC 2024*, eds Eichlseder M, Gambs S (Springer Nature Switzerland, Cham), pp 3–32.

[18] Chakraborty B, Dhar C, Nandi M (2024) Tight Multi-user Security of Ascon and Its Large Key Extension. *Information Security and Privacy - 29th Australasian Conference, ACISP 2024, Sydney, NSW, Australia, July 15-17, 2024, Proceedings, Part I*, eds Zhu T, Li Y (Springer), *Lecture Notes in Computer Science*, Vol. 14895, pp 57–76. DOI:10.1007/978-981-97-5025-2_4

[19] Lefevre C, Mennink B (2025) SoK: Security of the Ascon Modes. *IACR Trans Symmetric Cryptol* 2025(1):138–210. DOI:10.46586/TOSC.V2025.I1.138-210. Available at https://doi.org/10.46586/tosc.v2025.i1.138-210

[20] Naito Y, Sasaki Y, Sugawara T (2023) Committing Security of Ascon: Cryptanalysis on Primitive and Proof on Mode. *IACR Trans Symmetric Cryptol* 2023(4):420–451. DOI:10.46586/TOSC.V2023.I4.420-451. Available at https://doi.org/10.46586/tosc.v2023.i4.420-451

[21] Krämer J, Struck P, Weishäupl M (2024) Committing AE from Sponges Security Analysis of the NIST LWC Finalists. *IACR Trans Symmetric Cryptol* 2024(4):191–248. DOI:10.46586/TOSC.V2024.I4.191-248. Available at https://doi.org/10.46586/tosc.v2024.i4.191-248

[22] National Institute of Standards and Technology (2024) Module-Lattice-Based Key-Encapsulation Mechanism Standard (U.S. Department of Commerce, Washing-

ton, D.C.), Federal Information Processing Standards Publications (FIPS) 203. DOI:10.6028/NIST.FIPS.203

[23] Lefevre C, Mennink B (2022) Tight Preimage Resistance of the Sponge Construction. *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part IV*, eds Dodis Y, Shrimpton T (Springer), *Lecture Notes in Computer Science*, Vol. 13510, pp 185–204. DOI:10.1007/978-3-031-15985-5_7. Available at https://doi.org/10.1007/978-3-031-15985-5_7

[24] National Institute of Standards and Technology (2024) Cryptographic Module Validation Program (CMVP). Available at https://csrc.nist.gov/projects/cryptographic-module-validation-program.

[25] National Institute of Standards and Technology (2023) GitHub repository usnistgov/ACVP-Server: Automated Cryptographic Validation Test System — Gen/Vals). Available at https://github.com/usnistgov/ACVP-Server/tree/master/gen-val/json-files.

Appendix A. Implementation Notes

This specification follows the little-endian ordering convention. That is, on little-endian machines, byte strings or words of any size can be loaded from memory directly into the Ascon state without the need to perform any conversion. Neither bytes nor bits need to be reversed. The hexadecimal forms of the padding for Ascon functions are described in Sec. A.2.

However, the convention for printing the Ascon state using 64-bit integer words in hexadecimal notation (most significant byte and bit first) is different from printing the Ascon state using byte sequences or bitstrings (least significant byte and bit first). The conversion functions between printing byte sequences and printing integers are specified in Sec. A.1.

The least significant bit of S_0 is $s_{(0,0)}$ (i.e., $\mathcal{S}_{[0:0]}$), and the most significant bit of S_4 is $s_{(4,63)}$ (i.e., $\mathcal{S}_{[319:319]}$). Similarly, the least significant byte of S_0 is the first byte of state ($\mathcal{S}_{[0:7]}$), and the most significant byte of S_4 is the last byte of the state ($\mathcal{S}_{[312:319]}$). This relationship between state words, bytes, and state bits is shown in Fig. 9, where $S_i[j]$ denotes the j^{th} byte of state word S_i for $0 \leq i \leq 4$ and $0 \leq j \leq 7$.

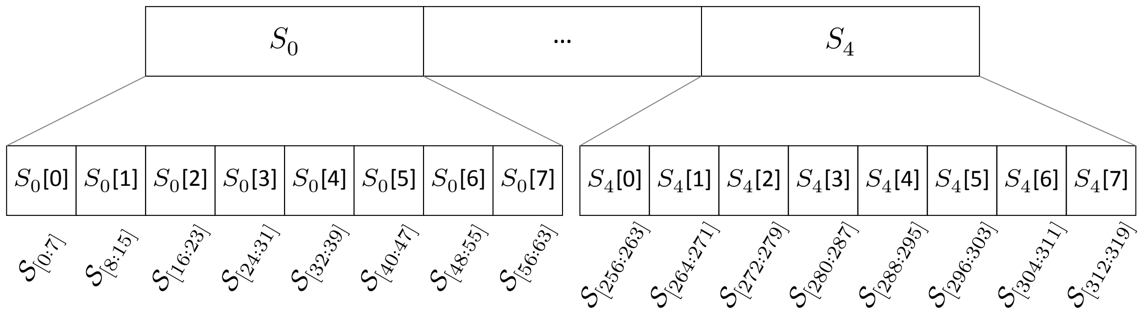

Figure 9. Mapping between state words, bytes, and bits

A.1. Conversion Functions

When printing values as integers using hexadecimal notation, the most significant byte and most significant bit are shown first.

Integers and byte sequences. Printing the integer representation of a byte sequence requires the byte order to be reversed. That is, the first element in the sequence of bytes is the least significant byte of the integer, while the last element in the sequence of bytes is the most significant byte of the integer.

Integers and bitstrings. Printing a bitstring as an integer requires the byte order and the bits within a byte to be reversed. That is, the first element of a bitstring is the least significant bit of the integer (or byte), while the last element of the bitstring is the least significant bit of the integer (or byte).

Loading 64-bit integer words from a byte sequence. When loading the state from a sequence of bytes stored in memory, the first eight bytes are mapped to the first 64-bit unsigned integer word S_0 in little-endian notation (i.e., without byte reversal on little-endian machines). The next eight bytes are loaded to S_1. Bytes continue to be loaded in the same way until the final eight bytes of the stored state are loaded into S_4.

An example of the mapping between memory addresses to state word bytes is presented in Table 10 for both little-endian and big-endian machines. An example of mappings between 64-bit unsigned integers, byte sequences, and bitstrings is shown in Fig. 10. Note that 64-bit integers and bitstrings only appear to be reversed in the visual representation.

Table 10. Address for each byte of Ascon state word S_i in memory on little-endian and big-endian machines, where the word S_i begins at memory address a

Word byte	Little-endian address	Big-endian address
$S_i[0]$	$a+0$	$a+7$
$S_i[1]$	$a+1$	$a+6$
$S_i[2]$	$a+2$	$a+5$
$S_i[3]$	$a+3$	$a+4$
$S_i[4]$	$a+4$	$a+3$
$S_i[5]$	$a+5$	$a+2$
$S_i[6]$	$a+6$	$a+1$
$S_i[7]$	$a+7$	$a+0$

Writing 64-bit integer words to a byte sequence. The process for writing the 64-bit unsigned integer Ascon state words to a byte sequence in memory is simply the reverse of loading a state word from a byte sequence. The byte order does not need to be reversed on little-endian machines.

A.2. Implementing with Integers

This section provides additional information for software implementations that employ 64-bit unsigned integers.

Padding. The padding rule described in Algorithm 2 appends a one followed by one or more zeroes to data. For an integer x that can be represented with $n < 8$ bytes, an integer y that represents a padded version of x is computed as

$$y \leftarrow x \oplus (\texttt{0x0000000000000001} \ll 8n).$$

State bits	State word	Word value (64-bit unsigned integers)
$\mathcal{S}_{[0:63]}$	S_0	`0x0706050403020100`
$\mathcal{S}_{[64:127]}$	S_1	`0x0F0E0D0C0B0A0908`
$\mathcal{S}_{[128:191]}$	S_2	`0x1716151413121110`
$\mathcal{S}_{[192:255]}$	S_3	`0x1F1E1D1C1B1A1918`
$\mathcal{S}_{[256:319]}$	S_4	`0x2726252423222120`

$$\updownarrow$$

State bits	State word	Word value (byte sequence)
$\mathcal{S}_{[0:63]}$	S_0	`0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07`
$\mathcal{S}_{[64:127]}$	S_1	`0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F`
$\mathcal{S}_{[128:191]}$	S_2	`0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17`
$\mathcal{S}_{[192:255]}$	S_3	`0x18 0x19 0x1A 0x1B 0x1C 0x1D 0x1E 0x1F`
$\mathcal{S}_{[256:319]}$	S_4	`0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27`

$$\updownarrow$$

State bits	State word	Word value (bitstring)
$\mathcal{S}_{[0:63]}$	S_0	`0000 0000 1000 0000 0100 0000 1100 0000`
		`0010 0000 1010 0000 0110 0000 1110 0000`
$\mathcal{S}_{[64:127]}$	S_1	`0001 0000 1001 0000 0101 0000 1101 0000`
		`0011 0000 1011 0000 0111 0000 1111 0000`
$\mathcal{S}_{[128:191]}$	S_2	`0000 1000 1000 1000 0100 1000 1100 1000`
		`0010 1000 1010 1000 0110 1000 1110 1000`
$\mathcal{S}_{[192:255]}$	S_3	`0001 1000 1001 1000 0101 1000 1101 1000`
		`0011 1000 1011 1000 0111 1000 1111 1000`
$\mathcal{S}_{[256:319]}$	S_4	`0000 0100 1000 0100 0100 0100 1100 0100`
		`0010 0100 1010 0100 0110 0100 1110 0100`

Figure 10. Representation of the Ascon state as 64-bit unsigned integers, byte sequences, and bitstrings, where 64-bit unsigned integers are used to define the permutation, data stored in memory is represented as byte sequences, and bitstrings are used to specify the modes of operation. Note that 64-bit integers and bitstrings only appear to be reversed in the visual representation.

Table 11. Examples of padding an unsigned integer x to a 64-bit block, where x encodes a sequence of bytes that each have value 0xFF in little-endian byte order

Length of x (in bytes)	# Padding Bytes	Unsigned integer x	Padded 64-bit block
0	8	0x0000000000000000	0x0000000000000001
1	7	0x00000000000000FF	0x00000000000001FF
2	6	0x000000000000FFFF	0x000000000001FFFF
3	5	0x0000000000FFFFFF	0x0000000001FFFFFF
4	4	0x00000000FFFFFFFF	0x00000001FFFFFFFF
5	3	0x000000FFFFFFFFFF	0x000001FFFFFFFFFF
6	2	0x0000FFFFFFFFFFFF	0x0001FFFFFFFFFFFF
7	1	0x00FFFFFFFFFFFFFF	0x01FFFFFFFFFFFFFF

Domain separation bit. The hexadecimal integer form of the domain separation bit is 0x8000000000000000. Therefore, the addition of this bit into the state may be implemented as

$$S_4 \leftarrow S_4 \oplus \texttt{0x8000000000000000}.$$

64-bit block absorption. In Ascon-Hash256, Ascon-XOF128, or Ascon-CXOF128, the absorption of a 64-bit message block expressed as the byte sequence 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07 can be implemented as

$$S_0 \leftarrow S_0 \oplus \texttt{0x0706050403020100}.$$

128-bit block absorption. Absorbing a 128-bit associated data or plaintext block represented by byte sequence 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F can similarly be implemented as

$$S_0 \leftarrow S_0 \oplus \texttt{0x0706050403020100}$$
$$S_1 \leftarrow S_1 \oplus \texttt{0x0F0E0D0C0B0A0908}.$$

Key addition. Ascon-AEAD128 has keyed initialization and finalization, where the key is added to the state in various locations. For a key represented as a sequence of bytes with value 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07, 0x08, 0x09, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, the key addition at the beginning of the initialization phase may be written as

$$S_1 \leftarrow S_1 \oplus \texttt{0x0706050403020100}$$
$$S_2 \leftarrow S_2 \oplus \texttt{0x0F0E0D0C0B0A0908}.$$

The key addition at the end of the initialization phase may be written as

$$S_3 \leftarrow S_3 \oplus \texttt{0x0706050403020100}$$
$$S_4 \leftarrow S_4 \oplus \texttt{0x0F0E0D0C0B0A0908}.$$

The key addition at the beginning of the finalization phase can be expressed as:

$$S_2 \leftarrow S_2 \oplus \texttt{0x0706050403020100}$$
$$S_3 \leftarrow S_3 \oplus \texttt{0x0F0E0D0C0B0A0908}.$$

The key addition at the end of finalization can be implemented as

$$S_3 \leftarrow S_3 \oplus \texttt{0x0706050403020100}$$
$$S_4 \leftarrow S_4 \oplus \texttt{0x0F0E0D0C0B0A0908}.$$

A.3. Precomputation

The initialization phases of `Ascon-Hash256`, `Ascon-XOF128`, and `Ascon-CXOF128` are independent of the input data (e.g., message, output length, customization string), allowing the resulting internal state to be precomputed to reduce runtime computations. See Table 12 for the resulting state at the end of the initialization phase for each function.

For example, an implementation of `Ascon-Hash256` using the precomputed values would replace the first two steps of Alg. 5

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

with steps that assign each of the precomputed words into the corresponding state words, namely:

$$S_0 \leftarrow \texttt{0x9b1e5494e934d681}$$
$$S_1 \leftarrow \texttt{0x4bc3a01e333751d2}$$
$$S_2 \leftarrow \texttt{0xae65396c6b34b81a}$$
$$S_3 \leftarrow \texttt{0x3c7fd4a4d56a4db3}$$
$$S_4 \leftarrow \texttt{0x1a5c464906c5976d}.$$

The same is true for `Ascon-XOF128` and `Ascon-CXOF128` using the corresponding values given in Table 12.

It may also be beneficial to precompute the intermediate state between the customization and absorbing phases when `Ascon-CXOF128` repeatedly uses the same customization string.

Table 12. Precomputed initialization phase values for `Ascon-Hash256`, `Ascon-XOF128`, and `Ascon-CXOF128` provided in hexadecimal integer form

State word	`Ascon-Hash256`	`Ascon-XOF128`	`Ascon-CXOF128`
S_0	0x9b1e5494e934d681	0xda82ce768d9447eb	0x675527c2a0e8de03
S_1	0x4bc3a01e333751d2	0xcc7ce6c75f1ef969	0x43d12d7dc0377bbc
S_2	0xae65396c6b34b81a	0xe7508fd780085631	0xe9901dec426e81b5
S_3	0x3c7fd4a4d56a4db3	0x0ee0ea53416b58cc	0x2ab14907720780b6
S_4	0x1a5c464906c5976d	0xe0547524db6f0bde	0x8f3f1d02d432bc46

Appendix B. Determination of the Initial Values

Each variant of the Ascon family has a 64-bit initial value constructed as

$$IV = v \| 0^8 \| a \| b \| t \| r/8 \| 0^{16}, \tag{79}$$

where

- v is a unique identifier for the algorithm (represented in 8 bits)
- a is the number of rounds during initialization and finalization (represented in 4 bits)
- b is the number of rounds during the processing of AD, plaintext, and ciphertext for `Ascon-AEAD128` and the message for `Ascon-Hash256`, `Ascon-XOF128`, and `Ascon-CXOF128` (represented in 4 bits)
- t is 128 for `Ascon-AEAD128`, 256 for `Ascon-Hash256`, and 0 for `Ascon-XOF128` and `Ascon-CXOF128` (represented in 16 bits)
- $r/8$ is the number of input bytes processed per invocation of the underlying permutation (represented in 8 bits)

The values of these parameters for each variant are given in Table 13, and the initial values for each Ascon variant are specified in Table 14.

Table 13. Parameters for initial value construction

Ascon variants	v (8 bits)	a (4 bits)	b (4 bits)	t (16 bits)	$r/8$ (8 bits)
`Ascon-AEAD128`	1	12	8	128	16
`Ascon-Hash256`	2	12	12	256	8
`Ascon-XOF128`	3	12	12	0	8
`Ascon-CXOF128`	4	12	12	0	8

Table 14. Initial values as hexadecimal integers

Ascon variants	Initial value
`Ascon-AEAD128`	`0x00001000808c0001`
`Ascon-Hash256`	`0x0000080100cc0002`
`Ascon-XOF128`	`0x0000080000cc0003`
`Ascon-CXOF128`	`0x0000080000cc0004`