NIST Special Publication 800
NIST SP 800-232 ipd

Ascon-Based Lightweight Cryptography Standards for Constrained Devices

Authenticated Encryption, Hash, and Extendable Output Functions

Initial Public Draft

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.ipd

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

NIST Special Publication 800
NIST SP 800-232 ipd

Ascon-Based Lightweight Cryptography Standards for Constrained Devices

Authenticated Encryption, Hash, and Extendable Output Functions

Initial Public Draft

Meltem Sönmez Turan
Kerry A. McKay
Jinkeon Kang
John Kelsey

Computer Security Division
Information Technology Laboratory

Donghoon Chang

Strativia

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

November 2024

U.S. Department of Commerce
Gina M. Raimondo, Secretary

National Institute of Standards and Technology
Laurie E. Locascio, NIST Director and Under Secretary of Commerce for Standards and Technology

Authority

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3551 et seq., Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130.

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 YYYY-MM-DD [Will be added in the final publication.]

How to cite this NIST Technical Series Publication: Meltem Sönmez Turan, Kerry A. McKay, Donghoon Chang, Jinkeon Kang, John Kelsey (2024) Ascon-Based Lightweight Cryptography Standards for Constrained Devices. (National Institute of Standards and Technology, Gaithersburg, MD),NIST Special Publication (SP) NIST SP 800-232 ipd. https://doi.org/10.6028/NIST.SP.800-232.ipd

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

Public Comment Period
November 8, 2024 -– February 7, 2025

Submit Comments
SP800-232-comments@list.nist.gov

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930

Additional Information
Additional information about this publication is available at https://csrc.nist.gov/pubs/sp/800/232/ipd, 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 cryptography solutions for resource-constrained devices. This decision emerged from a rigorous, multi-round lightweight cryptography standardization process. This standard introduces a new Ascon-based family of symmetric-key cryptographic primitives designed to deliver Authenticated Encryption with Associated Data (AEAD), hash, and Extendable Output Function (XOF) capabilities, namely `Ascon-AEAD128`, `Ascon-Hash256`, `Ascon-XOF128`, and `Ascon-CXOF128`. 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 draft standard outlines the technical specifications of `Ascon-AEAD128`, `Ascon-Hash256`, `Ascon-XOF128`, and `Ascon-CXOF128`, and provides their security properties.

Keywords

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

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems. The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

Call for Patent Claims

This public review includes a call for information on essential patent claims (claims whose use would be required for compliance with the guidance or requirements in this Information Technology Laboratory (ITL) draft publication). Such guidance and/or requirements may be directly stated in this ITL Publication or by reference to another publication. This call also includes disclosure, where known, of the existence of pending U.S. or foreign patent applications relating to this ITL draft publication and of any relevant unexpired U.S. or foreign patents.

ITL may require from the patent holder, or a party authorized to make assurances on its behalf, in written or electronic form, either:

1. assurance in the form of a general disclaimer to the effect that such party does not hold and does not currently intend holding any essential patent claim(s); or
2. assurance that a license to such essential patent claim(s) will be made available to applicants desiring to utilize the license for the purpose of complying with the guidance or requirements in this ITL draft publication either:
    (a) under reasonable terms and conditions that are demonstrably free of any unfair discrimination; or
    (b) without compensation and under reasonable terms and conditions that are demonstrably free of any unfair discrimination.

Such assurance shall indicate that the patent holder (or third party authorized to make assurances on its behalf) will include in any documents transferring ownership of patents subject to the assurance, provisions sufficient to ensure that the commitments in the assurance are binding on the transferee, and that the transferee will similarly include appropriate provisions in the event of future transfers with the goal of binding each successor-in-interest.

The assurance shall also indicate that it is intended to be binding on successors-in-interest regardless of whether such provisions are included in the relevant transfer documents.

Such statements should be addressed to: SP800-232-comments@list.nist.gov

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 drafting 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, Lily Chen, Andrew Regenscheid, Noah Ross and Sara Kerman, who provided technical and administrative support.

1. Introduction

This draft 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 specified in this standard are as follows:

1. `Ascon-AEAD128` is a nonce-based authenticated encryption with associated data that provides 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 an 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 the 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 suitable for constrained environments in which conventional cryptographic standards (e.g., AES-GCM [4, 5] and the SHA-2 [6] and the 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], NIST IR 8369 [10], and NIST IR 8454 [11]).

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

[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.

1. **Permutations.** The Ascon submission defined three Ascon permutations having 6, 8, and 12 rounds. This standard specifies additional Ascon permutations by providing round 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 extendable output functions, 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 are 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 the 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. This approach significantly reduces implementation costs compared to designs that require inverse operations for decryption.

Organization. Section 2 provides preliminaries, including the notation, basic operations, and auxiliary functions. Section 3 specifies the Ascon permutations for up to 16 rounds. Section 4 specifies the authenticated encryption scheme `Ascon-AEAD128`, provides some implementation options for truncation and nonce masking, lists the requirements for validation, and provides security properties. Section 5 specifies the hash function `Ascon-Hash256` , the XOF function `Ascon-XOF128`, and the customized `Ascon-CXOF128` and describes their security properties. 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
GCM	**G**alois/**C**ounter **M**ode
NIST	**N**ational **I**nstitute of **S**tandards and **T**echnology
PRF	**P**seudo-**R**andom **F**unction
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 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.
bit string	A finite, ordered sequence of bits.

Table 2. Terms and definitions

Term	Definition
capacity	The width of the underlying permutation minus the rate.
digest	Hash value.
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 the nonce-uniqueness requirement is unintentionally or accidentally violated.
nonce-respecting	A setting that satisfies the nonce-uniqueness requirement.
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.
shall	Term used to express a requirement that needs to be fulfilled to claim conformance to this standard.
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.
width	The state size of the underlying permutation.

284 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, $0 \leq i \leq 4, 0 \leq j \leq 63$
$S_i[j]$	j^{th} 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 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.
$\lvert X \rvert$	Length of the bitstring X in bits
$X \Vert Y$	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 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 evaluate as $f(g(x))$.
$\odot$	Bitwise AND operation
$\oplus$	Bitwise XOR operation
$X \ggg i$	Right rotation (circular shift) by i bits of 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 equality comparison, i.e., `true`, if x is equal to y, `false` otherwise.

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 $X_0, X_1, \ldots, \widetilde{X_\ell}$, where $\ell \leftarrow \lfloor |X|/r \rfloor$ (i.e., $X \leftarrow X_0 \parallel X_1 \parallel \ldots \parallel \widetilde{X_\ell}$). The X_i blocks for $0 \leq i \leq \ell - 1$ each have a bit length r, whereas the bit length of the final block $\widetilde{X_\ell}$ is between 0 and $r - 1$ (see Algorithm 1).

Algorithm 1 parse(X, r)

Input: bitstring X, rate r
Output: bitstrings $X_0, \ldots, X_{\ell-1}, \widetilde{X_\ell}$

$\ell \leftarrow \lfloor |X|/r \rfloor$
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 $X_0, \ldots, X_{\ell-1}, \widetilde{X_\ell}$

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 $(-|X| - 1)$ mod r. The length of the output bitstring is a multiple of r (see Algorithm 2).

Algorithm 2 pad(X, r)

Input: bitstring X, rate r
Output: padded bitstring X'

$j \leftarrow (-|X| - 1)$ mod 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 $1 \leq rnd \leq 16$. 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).

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 $s_{(i,j)}$ represents 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 Fig. 1.

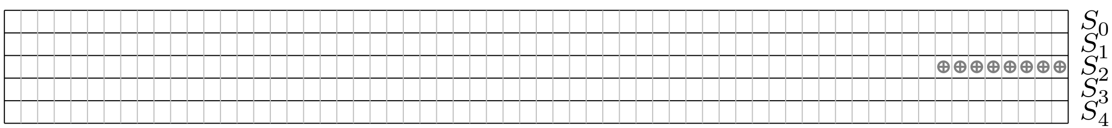

Figure 1. Constant-Addition Layer p_C

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

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

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

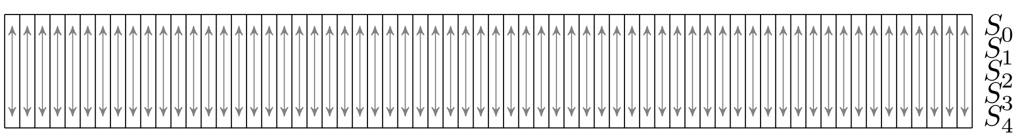

Figure 2. Substitution layer p_S

The 5-bit SBox has a 5-bit input $x = (x_0, x_1, \dots, x_4)$ and computes the 5-bit output using the circuit provided in Figure 3. SBox may also be implemented as a lookup table, as shown in Table 6.

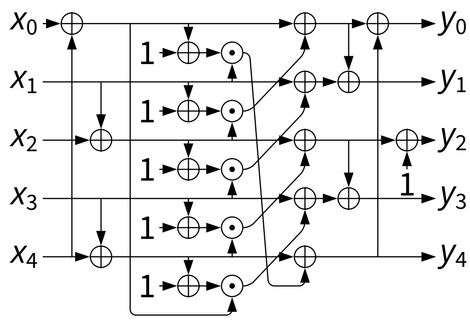

Figure 3. 5-bit S-box SBox

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 Fig. 4.

Figure 4. Linear diffusion layer p_L

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

$$\Sigma_0(S_0) = S_0 \oplus (S_0 \ggg 19) \oplus (S_0 \ggg 28) \quad (6)$$

$$\Sigma_1(S_1) = S_1 \oplus (S_1 \ggg 61) \oplus (S_1 \ggg 39) \quad (7)$$

$$\Sigma_2(S_2) = S_2 \oplus (S_2 \ggg \ \ 1) \oplus (S_2 \ggg \ \ 6) \quad (8)$$

$$\Sigma_3(S_3) = S_3 \oplus (S_3 \ggg 10) \oplus (S_3 \ggg 17) \quad (9)$$

$$\Sigma_4(S_4) = S_4 \oplus (S_4 \ggg \ \ 7) \oplus (S_4 \ggg 41) \quad (10)$$

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 $(0,0,0,0,1)$).

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 Section 4.2.1 for the truncation option):

$$\texttt{Ascon-AEAD128.enc}(K, N, A, P) = (C, T), \tag{11}$$

`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:

$$\texttt{Ascon-AEAD128.dec}(K, N, A, C, T) = \begin{cases} P & \text{if the tag } T \text{ is valid} \\ \texttt{fail} & \text{otherwise} \end{cases} \tag{12}$$

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 Fig. 5).

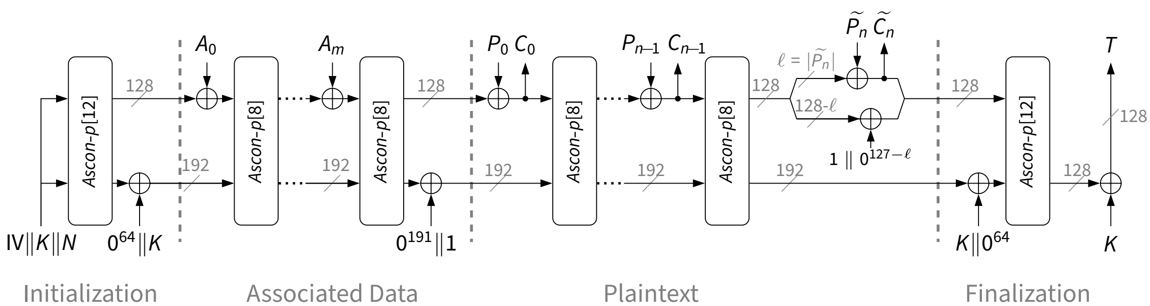

Figure 5. `Ascon-AEAD128` encryption

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

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

$$\mathcal{S} \leftarrow IV \| K \| N \tag{13}$$

Algorithm 3 $\texttt{Ascon-AEAD128.enc}(K, N, A, P)$

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

$IV \leftarrow \texttt{0x00001000808c0001}$ ▷ 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
  $A_0, \ldots, A_{m-1}, \widetilde{A_m} \leftarrow \mathsf{parse}(A, 128)$
  $A_m \leftarrow \mathsf{pad}(\widetilde{A_m}, 128)$
  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
$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{319} \,\|\, 1)$

$P_0, \ldots, P_{n-1}, \widetilde{P_n} \leftarrow \mathsf{parse}(P, 128)$ ▷ 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
$\mathcal{S}_{[0:127]} \leftarrow \mathcal{S}_{[0:127]} \oplus \mathsf{pad}(\widetilde{P_n}, 128)$
$\widetilde{C_n} \leftarrow \mathcal{S}_{[0,\ell-1]}$
$C \leftarrow C_0 \,\|\, \ldots \,\|\, C_{n-1} \,\|\, \widetilde{C_n}$

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

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

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}) \tag{14}$$

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

$$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{192} \| K). \tag{15}$$

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.

   When associated data A is non-empty (i.e., $|A| > 0$), it is parsed into blocks, as

   $$A_0,\ A_1,\ \ldots,\ A_{m-1},\ \widetilde{A_m} \leftarrow \text{parse}(A, 128), \tag{16}$$

   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

   $$A_m \leftarrow \text{pad}(\widetilde{A_m}, 128) = \widetilde{A_m} || 1 \,\|\, 0^{127-|\widetilde{A_m}|} \tag{17}$$

   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

   $$\mathcal{S}_{[0:127]} \leftarrow \mathcal{S}_{[0:127]} \oplus A_i, \tag{18}$$

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

   $$\mathcal{S} \leftarrow Ascon\text{-}p[8](\mathcal{S}). \tag{19}$$

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

   $$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{319} \| 1) \tag{20}$$

   that provides domain separation. For empty associated data, only the final step described in (20) is applied.

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

   $$P_0,\ P_1,\ \ldots,\ P_{n-1}, \widetilde{P_n} \leftarrow \text{parse}(P, 128), \tag{21}$$

   where $n = \lfloor |P|/128 \rfloor$ and $|P_i| = 128$ 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 follows:

$$\mathcal{S}_{[0:127]} \leftarrow \mathcal{S}_{[0:127]} \oplus P_i, \tag{22}$$

followed by generating the corresponding ciphertext block C_i as

$$C_i \leftarrow \mathcal{S}_{[0:127]}, \tag{23}$$

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

$$\mathcal{S} \leftarrow Ascon\text{-}p[8](\mathcal{S}). \tag{24}$$

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

$$\mathcal{S}_{[0:127]} \leftarrow \mathcal{S}_{[0:127]} \oplus \mathsf{pad}(\widetilde{P_n}, 128), \tag{25}$$

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

$$\widetilde{C_n} \leftarrow \mathcal{S}_{[0:\ell-1]}. \tag{26}$$

The ciphertext C is constructed by concatenating the ciphertext blocks as

$$C \leftarrow C_0 \| \ldots \| C_{n-1} \| \widetilde{C_n}. \tag{27}$$

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

$$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{128} \| K \| 0^{64}), \tag{28}$$

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

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}). \tag{29}$$

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

$$T \leftarrow S_{[192:319]} \oplus K. \tag{30}$$

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

4.1.2. 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.

Algorithm 4 $\texttt{Ascon-AEAD128.dec}(K, N, A, C, T)$

Input: 128-bit key K; 128-bit nonce N; Associated data A; Ciphertext C; 128-bit tag T
Output: Plaintext P or $\texttt{fail}$

$IV \leftarrow \texttt{0x00001000808c0001}$ ▷ 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**
    $\mathcal{S}_{[0:127]} \leftarrow \mathcal{S}_{[0:127]} \oplus A_i$
    $\mathcal{S} \leftarrow Ascon\text{-}p[8](\mathcal{S})$
  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 $\texttt{fail}$
end if

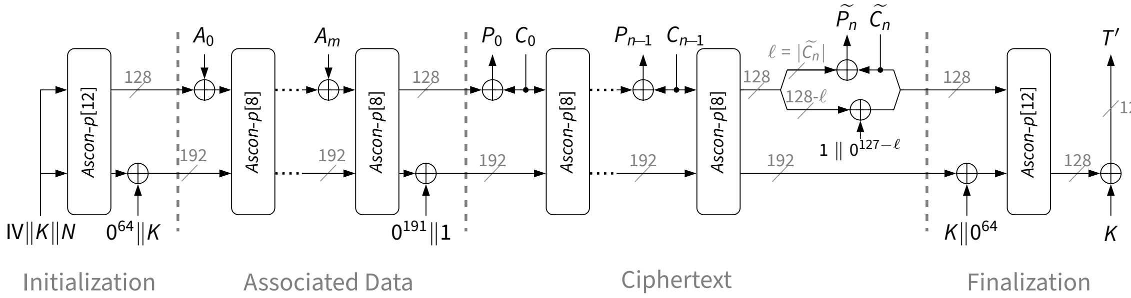

Figure 6. `Ascon-AEAD128` decryption

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

$$\mathcal{S} \leftarrow IV \| K \| N, \tag{31}$$

where the initial value IV is assigned to `0x00001000808c0001`. Next, $\mathcal{S}$ is updated using the permutation $Ascon\text{-}p[12]$ as

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}) \tag{32}$$

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

$$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{192} \| K). \tag{33}$$

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.

When the associated data A is non-empty (i.e., $|A| > 0$), it is parsed into blocks, as

$$A_0,\ A_1,\ \ldots,\ A_{m-1},\ \widetilde{A_m} \leftarrow \text{parse}(A, 128), \tag{34}$$

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

$$A_m \leftarrow \text{pad}(\widetilde{A_m}, 128) = \widetilde{A_m} || 1 \parallel 0^{127-|\widetilde{A_m}|}. \tag{35}$$

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

$$\mathcal{S}_{[0:127]} \leftarrow (\mathcal{S}_{[0:127]} \oplus A_i), \tag{36}$$

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

$$\mathcal{S} \leftarrow Ascon\text{-}p[8](\mathcal{S}). \tag{37}$$

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

$$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{319} \| 1) \tag{38}$$

for domain separation. For empty associated data, only the final step described in (38) 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

$$C_0,\ C_1,\ \dots, C_{n-1},\ \widetilde{C_n} \leftarrow \text{parse}(C, 128), \tag{39}$$

where $n = \lfloor |C|/128 \rfloor$, $|C_i| = 128$ for $0 \le i \le n-1$, $|\widetilde{C_n}| = \ell$, $0 \le \ell < 128$ 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:

$$P_i \leftarrow \mathcal{S}_{[0:127]} \oplus C_i \tag{40}$$

$$\mathcal{S}_{[0:127]} \leftarrow C_i \tag{41}$$

$$\mathcal{S} \leftarrow Ascon\text{-}p[8](\mathcal{S}) \tag{42}$$

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

$$\widetilde{P_n} \leftarrow \mathcal{S}_{[0,\ell-1]} \oplus \widetilde{C_n} \tag{43}$$

$$\mathcal{S}_{[0,\ell-1]} \leftarrow \widetilde{C_n} \tag{44}$$

$$\mathcal{S}_{[\ell,127]} \leftarrow \mathcal{S}_{[\ell,127]} \oplus (1||0^{127-\ell}) \tag{45}$$

The plaintext P is constructed by concatenating the plaintext blocks as

$$P \leftarrow P_0 \| \dots \| P_{n-1} \| \widetilde{P_n}. \tag{46}$$

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

$$\mathcal{S} \leftarrow \mathcal{S} \oplus (0^{128} \| K \| 0^{64}), \tag{47}$$

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

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}). \tag{48}$$

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

$$T' \leftarrow (S_{[192:319]}) \oplus K. \tag{49}$$

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 $T_{[0:\lambda-1]}$ of the tag.

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

4.2.2. Nonce Masking

This section provides an option to implement Ascon-AEAD128 using a 256-bit key, mainly to maintain the 128-bit security strength of Ascon-AEAD128 in a multi-key setting [12]. In this option, an additional 128-bit key is used to mask the input nonce.

Let K be the 128-bit key of Ascon-AEAD128 and K' be an independently generated additional 128-bit key. Ascon-AEAD128 with nonce masking is processed as follows:

$$\mathtt{E}(K \big\| K', N, A, P) = \texttt{Ascon-AEAD128.enc}(K, N \oplus K', A, P), \tag{50}$$

$$\mathtt{D}(K \big\| K', N, A, C, T) = \texttt{Ascon-AEAD128.dec}(K, N \oplus K', A, C, T) \tag{51}$$

Ascon-AEAD128 with nonce masking should only be used when context-commitment security [13] and related-key security are not concerns 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 $N \oplus K' = N' \oplus K''$.

4.3. AEAD Requirements

This section specifies requirements for Ascon-AEAD128.

R1. Key generation. The secret key K and the nonce-masking key K' (if available) **shall** be generated following the recommendations for cryptographic key generation specified in SP 800-133 [14] 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. Use of unique nonce. Nonce **shall** be distinct for each encryption operation for a given key to ensure that identical plaintexts encrypted multiple times produce different ciphertext.

R3. Minimum length of truncated tag. When an application uses truncated tags, the bit length of the truncated tags **shall** be at least 64 bits, and the tag length **shall** be the same across the life-span of the key.

R4. Limit on the maximum number of decryption failures. When the tag bit length is λ, $64 \le \lambda \le 128$, the maximum number of decryption failures for a fixed key **shall** be at most $2^{\lambda-64}$.

R5. 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.

R6. Key update. The key **shall** be updated to a new one when the total number of input data blocks or the number of decryption failures reach their respective limits or if the nonce uniqueness requirement is violated.

4.4. Security Properties

This section provides the security properties of 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 attacker focuses on a specific key that is shared by one or more users. In contrast, in the multi-key setting with u keys, the attacker aims to compromise any of the u keys used by the users.

The security of the Ascon-AEAD128 mode, in both single-key and multi-key settings, was evaluated in [12, 15–17].

4.4.1. Single-Key Setting

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. When the tag is λ bits, $64 \le \lambda \le 128$, the maximum number of decryption failures for a fixed key is limited to $2^{\lambda-64}$. Therefore, the probability that there is a valid forgery is at most 2^{-64}. Once a forgery attempt is successful, the confidentiality of the plaintext can be immediately compromised, as the decryption function may reveal some information about the plaintext. Therefore, in the single-key setting, Ascon-AEAD128 with tag length λ provides (min$\{128, \lambda\}$)-bit security strengths for confidentiality and integrity in the nonce-respecting setting.

4.4.2. Multi-Key Setting

When u keys are independently selected for an application, 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 of (nonce, associated data, ciphertext, tag), where the total number of input bytes for all u keys is limited to 2^{54} (i.e., 2^{50} blocks).

When the same nonce is used with u keys, an attacker may be able to discover one of the u keys with a time complexity of $2^{128-\log_2 u}$, thereby compromising both confidentiality and integrity.

To improve security in a multi-key setting, the nonce masking implementation option (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. When the tag is truncated to λ bits, $64 \leq \lambda \leq 128$, the maximum number of decryption failures for all u keys is limited to $2^{\lambda-64}$. Therefore, the probability of obtaining a valid forgery is expected to be at most 2^{-64}. In the multi-key setting, `Ascon-AEAD128` with tag-length λ provides ($\min\{128 - \log_2 u, \lambda\}$)-bit security strengths of confidentiality and integrity in the nonce-respecting setting.

4.4.3. Nonce-Misuse Setting

The plaintext confidentiality of `Ascon-AEAD128` is lost when a nonce is repeated with the same secret key. However, `Ascon-AEAD128` is designed to provide some level of security in case of certain implementation errors that violate the nonce-respecting requirement.

- In the u-key setting, `Ascon-AEAD128` with a λ-bit tag provides ($\min\{128 - \log_2(u), \lambda\}$)-bit security strengths of confidentiality and integrity when a (nonce, associated data) pair is never repeated for two encryptions with each of u keys and the number of nonce repetitions per key for encryption is limited to 2^8. In this scenario, the security strengths of `Ascon-AEAD128` are summarized in Table 7.
- `Ascon-AEAD128` with λ-bit tag also provides a ($\min\{128 - \log_2(u), \lambda\}$)-bit integrity security strength of the tuple (nonce, associated data, ciphertext, tag) if the number of repetitions of any (nonce, associated data) pair per each of u keys for encryption is limited to 2^8. In this scenario, the integrity security strength of `Ascon-AEAD128` with λ-bit tag is summarized in Table 8.

Table 7. Security strength of `Ascon-AEAD128` with λ-bit tag in the u-key setting, where (N, A) pair is unique for encryption.

Security	Security strengths 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)	$\min\{128 - \log_2(u), \lambda\}$	$\leq 2^8$

5. Hash and Extendable Output Functions

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

- The hash function `Ascon-Hash256`, which produces a 256-bit digest,
- The `Ascon-XOF128` function that produces arbitrary length outputs, and
- The customized XOF `Ascon-CXOF128`.

5.1. Specification of `Ascon-Hash256`

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

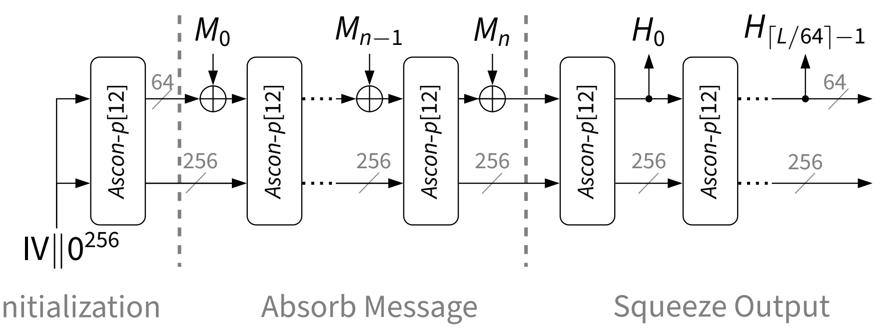

Figure 7. Structure of `Ascon-Hash256` and `Ascon-XOF128`

`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. That is the initialization step is

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](IV \| 0^{256}). \tag{52}$$

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)	$\min\{128 - \log_2(u), \lambda\}$	$\leq 2^8$

2. **Absorbing the message.** The absorbing phase behaves similarly to the associated data processing of `Ascon-AEAD128`. The message is partitioned into 64-bit blocks as

$$M_0, \ldots, M_{n-1}, \widetilde{M_n} \leftarrow \mathsf{parse}(M, 64). \tag{53}$$

Partial block $\widetilde{M_n}$ is then padded to a full block M_n:

$$M_n \leftarrow \mathsf{pad}(\widetilde{M_n}, 64). \tag{54}$$

Each message block M_i is XORed with the state as

$$\mathcal{S}_{[0:63]} \leftarrow \mathcal{S}_{[0:63]} \oplus M_i. \tag{55}$$

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.

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}) \tag{56}$$

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

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}) \tag{57}$$

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

$$H_i \leftarrow \mathcal{S}_{[0:63]} \tag{58}$$

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

Steps (58) and (59) 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 permutation.

$$H_3 \leftarrow \mathcal{S}_{[0:63]} \tag{60}$$

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

$$H \leftarrow H_0 \| H_1 \| H_2 \| H_3. \tag{61}$$

Algorithm 5 `Ascon-Hash256`(M)

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

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

$M_0, \ldots, M_{n-1}, \widetilde{M_n} \leftarrow \mathsf{parse}(M, 64)$ ▷ 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
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 has three main differences:

1. Ascon-XOF128 accepts an additional input, $L > 0$, that specifies the desired output length in bits.
2. The number of blocks that are squeezed is equal to $\lceil L/64 \rceil$.
3. The initial value differs in one bit.

The 128 in the name Ascon-XOF128 refers to the target security strength, not the output size.

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:

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](IV \| 0^{256}). \quad (62)$$

2. **Absorbing the message.** The absorbing phase behaves similar to the associated data processing of AEAD. The message is partitioned into 64-bit blocks as:

$$M_0, \ldots, M_{n-1}, \widetilde{M_n} \leftarrow \text{parse}(M, 64). \quad (63)$$

Partial block $\widetilde{M_n}$ is then padded to a full block M_n as

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

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

$$\mathcal{S}_{[0:63]} \leftarrow \mathcal{S}_{[0:63]} \oplus M_i. \quad (65)$$

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.

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}) \quad (66)$$

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.

$$\mathcal{S} \leftarrow Ascon\text{-}p[12](\mathcal{S}) \quad (67)$$

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

$$H_i \leftarrow \mathcal{S}_{[0:63]} \tag{68}$$

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

Steps (68) and (69) 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.

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

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

$$H' \leftarrow H_0 \| \ldots \| H_h \tag{71}$$

$$H \leftarrow H'_{[0:L-1]} \tag{72}$$

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

$M_0, \ldots, M_{n-1}, \widetilde{M_n} \leftarrow$ 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
$h \leftarrow \lceil L/64 \rceil - 1$
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

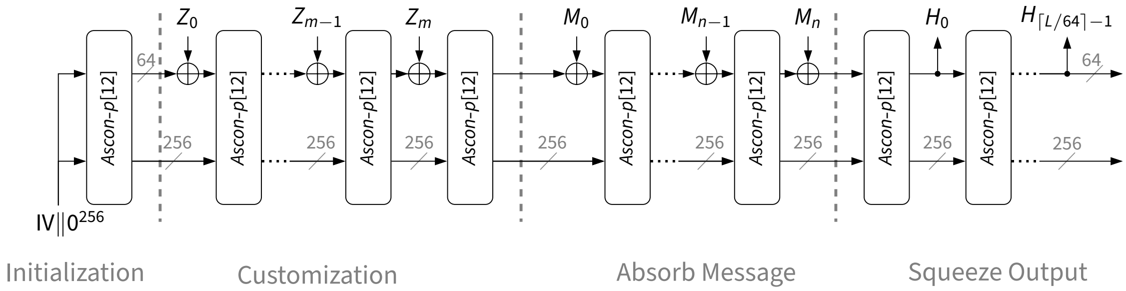

Figure 8. Structure of `Ascon-CXOF128`

5.3. Specification of `Ascon-CXOF128`

This section specifies the customized version of `Ascon-XOF128` called `Ascon-CXOF128`. Customization extends the functionality of `Ascon-XOF128` by allowing users to incorporate a customization string into the computation. For the same input message, two instances of a customized XOF using different customization strings will produce distinct outputs. `Ascon-CXOF128` is a customized XOF that differs from `Ascon-XOF128` in the following ways:

- For domain separation, `Ascon-CXOF128` uses a different IV than `Ascon-XOF128`. The IV for `Ascon-CXOF128` is `0x0000080000cc0004`.
- In addition to the message, `Ascon-CXOF128` takes the customization string Z as input. The length of the customization string **shall** be at most 2048 bits (i.e., 256 bytes).
- The customization string Z is prepended to the message blocks as

$$Z_0 \| Z_1 \| \ldots \| Z_m \| M_0 \| \ldots \| M_{n-1} \| M_n, \tag{73}$$

  where Z_0 is a 64-bit integer that represents the bit-length of the customization string, and $Z_1, \ldots, Z_m$ are 64-bit blocks generated by parsing and padding Z.

The general structure for `Ascon-CXOF128` is shown in Fig. 8 and the full specification is given by Algorithm 7.

5.4. Security Strengths

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

Algorithm 7 `Ascon-CXOF128`(M, L, Z)

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

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

$Z_0 \leftarrow$ int64($|Z|$) ▷ Customization
$Z_1 \ldots, Z_{m-1}, \widetilde{Z_m} \leftarrow$ parse($Z, 64$)
$Z_m \leftarrow$ 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

$M_0, \ldots, M_{n-1}, \widetilde{M_n} \leftarrow$ parse($M, 64$) ▷ Absorbing message
$M_n \leftarrow$ 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
$h \leftarrow \lceil L/64 \rceil - 1$
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

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

Function	Output size in bits	Security strengths 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)

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 (2001) Advanced Encryption Standard (AES) (U.S. Department of Commerce), Report. DOI: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] 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

[13] 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

[14] 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.

[15] 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

[16] Lefevre C, Mennink B (2023) Generic Security of the Ascon Mode: On the Power of Key Blinding, Cryptology ePrint Archive, Paper 2023/796. Available at https://ia.cr/2023/796.

[17] 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

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 $s_{(0,0)}$ (or $\mathcal{S}_{[0:0]}$) and the most significant bit of S_4 is $s_{(4,63)}$ (or $\mathcal{S}_{[319:319]}$). 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 $0 \leq j \leq 7$.

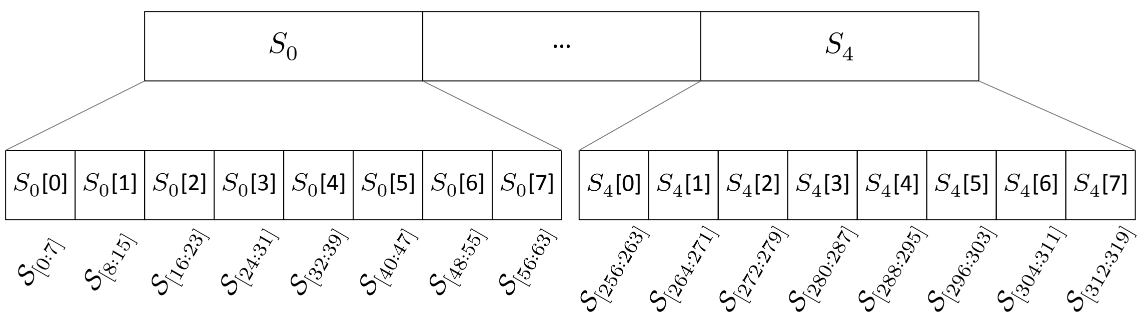

Figure 9. Mapping between state words, bytes, and bits

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 to be reversed, and additionally, 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).

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
$S_i[0]$	$a+0$	$a+7$
$S_i[1]$	$a+1$	$a+6$
$S_i[2]$	$a+2$	$a+5$
$S_i[3]$	$a+3$	$a+4$
$S_i[4]$	$a+4$	$a+3$
$S_i[5]$	$a+5$	$a+2$
$S_i[6]$	$a+6$	$a+1$
$S_i[7]$	$a+7$	$a+0$

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.

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 representing a padded version of x is computed as:

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

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 imple-

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 10001011 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 each having 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

mented 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 having 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:

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

and the key addition at the end of finalization can be implemented as:

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

Appendix B. Determination of the Initial Values

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

$$IV = v \| 0^8 \| a \| b \| t \| r/8 \| 0^{16}, \tag{74}$$

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 AEAD, and the number of rounds during processing the message for hash, XOF and CXOF (represented in 4 bits).
- t is 128 for `Ascon-AEAD128`, 256 for `Ascon-Hash256` and is 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 12, and initial values for each Ascon variant are specified in Table 13.

Table 12. 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 13. Initial values as hexadecimal integers

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