[](https://crossmark.crossref.org/dialog/?doi=10.6028/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>



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*

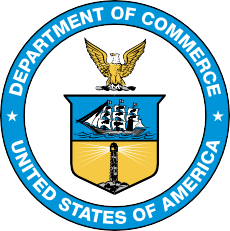
\* Former Foreign Guest Researcher; all work for this publication was done while at NIST.

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*

Certain equipment, instruments, software, or materials, commercial or non-commercial, are identified in this paper in order to specify the experimental procedure adequately. Such identification does not imply

recommendation or endorsement of any product or service by NIST, nor does it imply that the materials or equipment identified are necessarily the best available for the purpose.

There may be references in this publication to other publications currently under development by NIST in accordance with its assigned statutory responsibilities. The information in this publication, including

concepts and methodologies, may be used by federal agencies even before the completion of such

companion publications. Thus, until each publication is completed, current requirements, guidelines, and procedures, where they exist, remain operative. For planning and transition purposes, federal agencies may wish to closely follow the development of these new publications by NIST.

Organizations are encouraged to review all draft publications during public comment periods and provide feedback to NIST. Many NIST cybersecurity publications, other than the ones noted above, are available at <https://csrc.nist.gov/publications>

**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.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the ORCID, or any other federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

**NIST Technical Series Policies**

[Copyright, Use, and Licensing Statements](https://doi.org/10.6028/NIST-TECHPUBS.CROSSMARK-POLICY)

[NIST Technical Series Publication Identifier Syntax](https://www.nist.gov/nist-research-library/nist-technical-series-publications-author-instructions#pubid)

**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](mailto: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.

**All comments are subject to release under the Freedom of Information Act (FOIA).**

### 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

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 lead- ership for the Nation’s measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to ad- vance 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.

## Patent Disclosure Notice

NOTICE: ITL has requested that holders of patent claims whose use may be required for compliance with the guidance or requirements of this publication disclose such patent claims to ITL. However, holders of patents are not obligated to respond to ITL calls for patents and ITL has not undertaken a patent search in order to identify which, if any, patents may apply to this publication.

As of the date of publication and following call(s) for the identification of patent claims whose use may be required for compliance with the guidance or requirements of this publication, no such patent claims have been identified to ITL.

No representation is made or implied by ITL that licenses are not required to avoid patent infringement in the use of this publication.

# Table of Contents

1. [Introduction](#_bookmark0) [1](#_bookmark0)
2. [Preliminaries](#_bookmark2) [3](#_bookmark2)
   1. [Auxiliary Functions](#_bookmark7) [7](#_bookmark7)
3. [Ascon Permutations](#_bookmark10) [8](#_bookmark10)
   1. [Internal State](#_bookmark11) [8](#_bookmark11)
   2. [Constant-Addition Layer 𝑝𝐶 8](#_bookmark12)
   3. [Substitution Layer 𝑝𝑆 9](#_bookmark15)
   4. [Linear Diffusion Layer 𝑝𝐿](#_bookmark19) [10](#_bookmark19)
4. [Authenticated Encryption Scheme: Ascon-AEAD128 11](#_bookmark21)
   1. [Specification of Ascon-AEAD128 11](#_bookmark22)
      1. [Encryption](#_bookmark23) [11](#_bookmark23)
      2. [Decryption](#_bookmark27) [15](#_bookmark27)
   2. [Implementation Options](#_bookmark31) [18](#_bookmark31)
      1. [Truncation](#_bookmark32) [18](#_bookmark32)
      2. [Nonce Masking](#_bookmark33) [18](#_bookmark33)
   3. [AEAD Requirements](#_bookmark35) [19](#_bookmark35)
   4. [Security Properties](#_bookmark37) [20](#_bookmark37)
      1. [Single-Key Setting](#_bookmark38) [20](#_bookmark38)
      2. [Multi-Key Setting](#_bookmark39) [20](#_bookmark39)
      3. [Nonce-Misuse Setting](#_bookmark40) [21](#_bookmark40)
5. [Hash and eXtendable-Output Functions (XOFs)](#_bookmark43) [23](#_bookmark43)
   1. [Specification of Ascon-Hash256 23](#_bookmark44)
   2. [Specification of Ascon-XOF128 25](#_bookmark49)
   3. [Specification of Ascon-CXOF128 28](#_bookmark53)
   4. [Streaming API for XOF](#_bookmark56) [30](#_bookmark56)
   5. [Security Strengths](#_bookmark57) [31](#_bookmark57)
6. [Conformance](#_bookmark59) [31](#_bookmark59)

Appendix A. [Implementation Notes](#_bookmark84) [35](#_bookmark84)

* 1. [Conversion Functions](#_bookmark86) [35](#_bookmark86)
  2. [Implementing with Integers](#_bookmark88) [36](#_bookmark88)
  3. [Precomputation](#_bookmark91) [39](#_bookmark91)

Appendix B. [Determination of the Initial Values](#_bookmark93) [40](#_bookmark93)

**List of Tables**

Table 1. [Acronyms](#_bookmark3) [3](#_bookmark3)

Table 2. [Terms and definitions](#_bookmark4) [3](#_bookmark4)

Table 3. [Notations](#_bookmark5) [5](#_bookmark5)

Table 4. [Basic operations and functions](#_bookmark6) [6](#_bookmark6)

Table 5. [The constants const𝑖 to derive round constants of the Ascon permutations](#_bookmark13) . [9](#_bookmark13) Table 6. [Lookup table representation of SBOX](#_bookmark17) [10](#_bookmark17)

Table 7. [Security strength of Ascon-AEAD128 with 𝜆-bit tag in the 𝝊-key setting,](#_bookmark41)

[where (𝑁, 𝐴) pairs are distinct for encryption per key](#_bookmark41) [21](#_bookmark41)

[𝝊](#_bookmark42)

Table 8. [Integrity security strength of Ascon-AEAD128 with keys in the nonce-](#_bookmark42)

[misuse setting](#_bookmark42) [22](#_bookmark42)

Table 9. [Security strengths of Ascon-Hash256, Ascon-XOF128, and Ascon-CXOF128](#_bookmark58) [algorithms](#_bookmark58) [31](#_bookmark58)

Table 10. [Address for each byte of Ascon state word 𝑆𝑖 in memory on little-endian](#_bookmark87)

[and big-endian machines, where the word 𝑆𝑖 begins at memory address 𝑎 36](#_bookmark87)

[𝑥 𝑥](#_bookmark90)

Table 11. [Examples of padding an unsigned integer to a 64-bit block, where](#_bookmark90)

[encodes a sequence of bytes that each have value 0xFF in little-endian byte order](#_bookmark90) [38](#_bookmark90)

Table 12. [Precomputed initialization phase values for Ascon-Hash256, Ascon-XOF128,](#_bookmark92)

[and Ascon-CXOF128 provided in hexadecimal integer form](#_bookmark92) [40](#_bookmark92)

Table 13. [Parameters for initial value construction](#_bookmark94) [40](#_bookmark94)

Table 14. [Initial values as hexadecimal integers](#_bookmark95) [41](#_bookmark95)

# List of Figures

Figure 1. [Application of constant-addition layer 𝑝𝐶 to Ascon state](#_bookmark14) [9](#_bookmark14)

Figure 2. [Application of substitution layer 𝑝𝑆 to Ascon state](#_bookmark16) [9](#_bookmark16)

Figure 3. [Circuit representation of the 5-bit S-box SBOX](#_bookmark18) [10](#_bookmark18)

Figure 4. [Application of linear diffusion layer 𝑝𝐿 to Ascon state](#_bookmark20) [10](#_bookmark20)

Figure 5. [Ascon-AEAD128 encryption](#_bookmark24) [11](#_bookmark24)

Figure 6. [Ascon-AEAD128 decryption](#_bookmark28) [15](#_bookmark28)

Figure 7. [Structure of Ascon-Hash256 and Ascon-XOF128 23](#_bookmark45)

Figure 8. [Structure of Ascon-CXOF128 30](#_bookmark55)

Figure 9. [Mapping between state words, bytes, and bits](#_bookmark85) [35](#_bookmark85)

Figure 10. [Representation of the Ascon state as 64-bit unsigned integers, byte se-](#_bookmark89) [quences, and bitstrings](#_bookmark89) [37](#_bookmark89)

## 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.

## 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](#_bookmark60)] was first submitted to CAESAR (*Com- petition for Authenticated Encryption: Security, Applicability, and Robustness*) [1](#_bookmark1) 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](#_bookmark61)] for Round 2 and v1.2 [[3](#_bookmark62)] for Round 3 included minor tweaks, such as reorder- ing 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](#_bookmark63), [5](#_bookmark64)], SHA-2 [[6](#_bookmark65)] and SHA-3 [[7](#_bookmark66)] hash function families) may be resource-intensive. In February 2023, NIST announced the decision to standardize the Ascon family [[8](#_bookmark67)] for lightweight cryptography applications. For more information, refer to NIST Internal Report (IR) 8268 [[9](#_bookmark68)], IR 8369 [[10](#_bookmark69)], and IR 8454 [[11](#_bookmark70)].

*Differences from the Ascon submission v1.2.* The technical differences between this standard and the Ascon submission [[8]](#_bookmark67) 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

1CAESAR 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.

1. **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.
2. **Hash function variants.** The Ascon submission defined ASCON-HASH and ASCON-HASHA. This standard specifies Ascon-Hash256, which is based on ASCON-HASH.
3. **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.
4. **Initial values.** The initial values of the algorithms have been updated to support a new format that accommodates potential functionality extensions.
5. **Endianness.** The endianness has been switched from big endian to little endian to improve performance on little-endian microcontrollers.
6. **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 func- tionalities, 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 permuta- tions in the forward direction, implementing the inverse permutations is not needed.

*Organization.* Section [2](#_bookmark2) provides preliminaries, including the acronyms, terms, definitions, notation, basic operations, and auxiliary functions. Section [3](#_bookmark10) specifies the Ascon permuta- tions for up to 16 rounds. Section [4](#_bookmark21) specifies the Ascon-AEAD128AEAD scheme, provides some implementation options for truncation and nonce masking, lists the requirements for validation, and provides security properties. Section [5](#_bookmark43) specifies the Ascon-Hash256 hash function, the Ascon-XOF128 XOF, and the Ascon-CXOF128 customized XOF and describes their security properties. Section [6](#_bookmark59) provides information about conformance. [Appendix A](#_bookmark84) provides additional notes and conversion functions for implementations. [Appendix B](#_bookmark93) pro- vides additional information regarding the construction of initial values.

## Preliminaries

Table [1](#_bookmark3) 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](#_bookmark4) 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)

forgery

hash function

A function on bit strings in which the output can be extended to any desired length.

A (ciphertext, tag) pair produced by an adversary who is not knowledge- able of the secret key and yet is accepted as valid by the verified decryp- tion procedure.

A mathematical function that maps a string of arbitrary length to a fixed- length string.

message Input to the hash function.

nonce

nonce-misuse nonce-respecting rate

secret key

An input value to the authenticated encryption algorithm that is used only once for encryption performed under a given key.

A setting in which a nonce is used more than once for the encryption algorithm under a given key.

A setting in which a nonce is never repeated for the encryption algorithm under a given key.

The number of input bits processed or output bits generated per invoca- tion of the underlying permutation.

A cryptographic key that is used by a secret-key (i.e., symmetric) crypto- graphic 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 acciden- tal 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](#_bookmark5) lists the notations used in this standard.

**Table 3.** Notations

### Notation Definition

𝐾 128-bit secret key

𝑁 128-bit nonce

𝐴 Associated data

𝐴𝑖 𝑖th block of associated data 𝐴

𝑃 Plaintext

𝑃𝑖 𝑖th block of plaintext 𝑃

𝐶 Ciphertext

𝐶𝑖 𝑖th block of ciphertext 𝐶

𝑍 Customization string

𝑍𝑖 𝑖th block of customization string 𝑍

𝑇 128-bit authentication tag 64-bit constant initial value

𝐼𝑉

fail Error message to indicate that the verification of authenticated cipher- text failed

𝑀 Message

𝑀𝑖 𝑖th block of message 𝑀

𝐻 Hash value 𝐻

𝐻𝑖 𝑖th block of hash value 𝐻

𝑆0,…, 𝑆4 T𝑆he‖ 𝑆five‖ …64-‖bi𝑆t words of the internal state *S*, where *S* =

*S* 320-bit internal state of the underlying permutation

𝑠(𝑖,𝑗) 𝑗 0

1 𝑆𝑖, 0 ≤ 𝑖4≤ 4, 0 ≤ 𝑗 ≤ 63

th bit of

𝑆𝑖[𝑗] 𝑗𝑡ℎ byte of state word 𝑆𝑖 for 0 ≤ 𝑖 ≤ 4, 0 ≤ 𝑗 ≤ 7

𝑆[𝑖∶𝑗] The subset of state *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

𝑐𝑖 The constant value for round 𝑖 of the Ascon permutation

𝑝𝐶, 𝑝𝑆, 𝑝𝐿 Constant-addition, substitution, and linear layers of the round function 𝑝

Table [4](#_bookmark6) 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

0𝑠

When 𝑠 ≥ 0, 0𝑠 is th𝑠e bit string that consists of 𝑠 consecutive 0s.

|𝐸|

When 𝑠 = 0, then 0 is the empty string.

𝐸 ‖ 𝑌

𝑥 ×𝑦

𝑥 +𝑦

𝑥 −𝑦

𝑥/𝑦

𝑥

Length of the bitstring 𝐸 in bits

Concatenation of bitstrings 𝐸 and 𝑌 Multiplication of integers 𝑥 and 𝑦 Addition of integers 𝑥 and 𝑦

Subtraction of integers 𝑥 and 𝑦

Division of integer 𝑥 and non-zero integer 𝑦

mod 𝑦 Remainder in the integer division of 𝑥 by 𝑦

⌈𝑥⌉ For a real number 𝑥, the smallest integer greater than or equal

⌊𝑥⌋

𝑓 ∘ 𝑔

⊙

to 𝑥

For a real number 𝑥, the largest integer less than or equal to 𝑥 C(o𝑥m)position of functions 𝑓 and 𝑔 (e.g., for functions 𝑓(𝑥) and

𝑔 , 𝑓 ∘ 𝑔 is evaluated as 𝑓(𝑔(𝑥)))

Bitwise AND operation

⊕ Bitwise XOR operation

𝐸 ⋙ 𝑖 𝑖 𝐸

Right rotation (circular shift) by bits of the 64-bit word , where the least significant bit is the rightmost bit

𝐸 ≪ 𝑖 Left shift by 𝑖 bits

𝐸[𝑖∶𝑗] The subset of bitstring 𝐸 beginning at index 𝑖 and ending at

index 𝑗, inclusive. When 𝑖 > 𝑗, 𝐸 is the empty string. When

𝐸

𝑖 = 𝑗 [𝑖∶𝑗]

𝑥 == 𝑦

, [𝑖∶𝑗] is a single bit.

Boolean operator to perform an equality comparison, (i.e., true if is equal to ; otherwise, false)

𝑥 𝑦

0x Hexadecimal notation

int64(𝑥) 64-bit representation of integer 𝑥

### Auxiliary Functions

**Parse function.** Th̃e parse(𝐸, 𝜏) function parses the input bitstring̃𝐸 into a sequence of

blocks 𝐸0, 𝐸1,…, 𝐸ℓ, where 𝑙 ← ⌊|𝐸|/𝜏⌋ (i.e., 𝐸 ← 𝐸̃0 ‖ 𝐸1 ‖…‖ 𝐸ℓ). The 𝐸𝑖 blocks for

0 ≤ 𝑖 ≤ 𝑙 − 1

each have a bit length 𝜏, whereas 0 ≤ 𝐸ℓ ≤ 𝜏 − 1 (see Algorithm [1](#_bookmark8)). When

mod 𝜏 = 0, the final block is empty (i.e., |𝐸ℓ| = 0).

|𝐸| ̃

**Algorithm 1** parse(𝐸, 𝜏)

**Output:** bitstrings 𝐸0,…, 𝐸ℓ−1, 𝐸ℓ

**Input:** bitstring 𝐸, a positive intẽger 𝜏

𝑙 ← ⌊|𝐸|/𝜏⌋

[𝑖×𝜏∶(𝑖+1)×𝜏−1]

### end for

𝑖

**for** 𝑖𝐸= ←0 to𝐸𝑙 − 1 **do**

𝐸̃ℓ ← 𝐸[ℓ×𝜏∶|𝐸|−1] ̃

**return** 𝐸0,…, 𝐸ℓ−1, 𝐸ℓ

bitstring 0 , where 𝑗 is equal to (−|𝐸| − 1) mod 𝜏. The length of the output bitstring is a

**Padding ru**𝑗**le.** The function pad(𝐸, 𝜏) appends the bit 1 to the bitstring 𝐸, followed by the multiple of 𝜏 (see Algorithm [2](#_bookmark9)). For examples of padding when representing the data as

**Algorithm 2** pad(𝐸, 𝜏)

**Output:** padded bitstring 𝐸

**Input:** bitstring 𝐸, a positive′ integer 𝜏

64-bit unsigned integers, see Appendix [A.2](#_bookmark88).

𝑗𝐸←′ ←(−𝐸|𝐸∥|1−∥10)𝑗mod 𝜏

**return** 𝐸′

## Ascon Permutations

This section specifies the 𝜏𝑛𝑑-round 𝐴𝑠𝑐𝑜𝑛-𝑝[𝜏𝑛𝑑] permutations, where 𝜏𝑛𝑑 indicates

the number of rounds to be performed and 1 ≤ 𝜏𝑛𝑑 ≤ 16. The permutations follow the

Substitution-Permutation-Network (SPN) structure and consist of iterations of the round function that is defined as the composition of three steps

𝑝

𝑝 = 𝑝𝐿 ∘ 𝑝𝑆 ∘ 𝑝𝐶, (1)

where 𝑝𝐶 is the constant-addition layer (see Sec. [3.2](#_bookmark12)), 𝑝𝑆 is the substitution layer (see Sec. as a series of function invocations on an input 𝑥 as 𝑝𝐿(𝑝𝑆(𝑝𝐶(𝑥))).

𝐴𝑠𝑐𝑜𝑛 𝑝[8] 𝐴𝑠𝑐𝑜𝑛 𝑝[12]

[3.3](#_bookmark15)), and 𝑝𝐿 is the linear diffusion layer (see Sec. [3.4](#_bookmark19)). This composition can also be written

Note that - and - 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.

### Internal State

The permutations operate on the 320-bit state *S*, which is represented as five 64-bit words denoted as 𝑖 for :

𝑆 0 ≤ 𝑖 ≤ 4

*S* = 𝑆0 ∥ 𝑆1 ∥ 𝑆2 ∥ 𝑆3 ∥ 𝑆4. (2)

𝑠(𝑖,𝑗) 𝑗 𝑆𝑖 0 ≤ 𝑗 < 64

Let represent the th bit of , . In this specification of the Ascon permuta- tion, 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](#_bookmark84).

### Constant-Addition Layer 𝑝𝐶

The constant 𝑐𝑖 of round 𝑖 of the Ascon permutation 𝐴𝑠𝑐𝑜𝑛-𝑝[𝜏𝑛𝑑] (instantiated with 𝜏𝑛𝑑

rounds) for 𝜏𝑛𝑑 ≤ 16 and 0 ≤ 𝑖 ≤ 𝜏𝑛𝑑 − 1 is defined as

𝑐𝑖 = const16−𝜏𝑛𝑑+𝑖, (3)

where const0,…, const15 are defined in Table [5](#_bookmark13). The constant-addition layer 𝑝𝐶 adds a

64-bit round constant 𝑐 to 𝑆 in round 𝑖, for 𝑖 ≥ 0,

𝑖 2

𝑆2 = 𝑆2 ⊕ 𝑐𝑖. (4)

**Table 5.** The constants const𝑖 to derive round constants of the Ascon permutations

const const

|  |  |  |  |
| --- | --- | --- | --- |
|  | |  | |
|  |  |  |  |
| 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 2, as shown in Figure [1](#_bookmark14).

𝑆

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

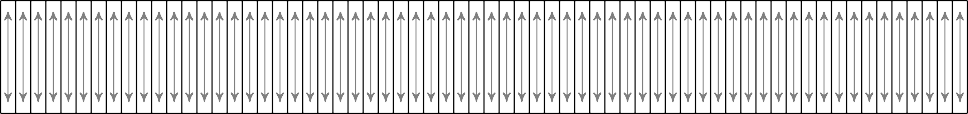
### Substitution Layer 𝑝𝑆

𝑝𝑆

𝑆01

𝑆234

The substitution layer updates the state *S* with 64 parallel applications of the 5-bit substitution box SBOX as

(𝑠(0,𝑗), 𝑠(1,𝑗),…, 𝑠(4,𝑗)) = SBOX(𝑠(0,𝑗), 𝑠(1,𝑗),…, 𝑠(4,𝑗)) (5) for 0 ≤ 𝑗 < 64, as shown in Figure [2](#_bookmark16).

𝑆

𝑆021

𝑆34

**Figure 2.** Application of substitution layer 𝑝𝑆 to Ascon state

The 5-bit SBOX is computed as

(𝑦0, ..., 𝑦4) = SBOX(𝑥0, ..., 𝑥4), (6)

where 𝑦0 = 𝑥4𝑥1 ⊕ 𝑥3 ⊕ 𝑥2𝑥1 ⊕ 𝑥2 ⊕ 𝑥1𝑥0 ⊕ 𝑥1 ⊕ 𝑥0,

𝑦1 = 𝑥4𝑥⊕ 𝑥⊕3𝑥2 ⊕ 𝑥3𝑥⊕1 𝑥⊕ 𝑥⊕3 1⊕, 𝑥2𝑥1 ⊕ 𝑥2 ⊕ 𝑥1 ⊕ 𝑥0,

(7)

𝑦2 = 𝑥4𝑥3 ⊕ 𝑥4 ⊕ 𝑥2𝑥 ⊕1𝑥 ⊕ 𝑥 ⊕ 𝑥 ⊕ 𝑥 ,

𝑦3 = 𝑥4𝑥0 ⊕ 𝑥4 ⊕ 𝑥3 ⊕0𝑥 𝑥3 ⊕ 𝑥2. 1 0

4

4

1

4

3

1

0

1

SBOX may also be implemented as a lookup table, as shown in Table [6](#_bookmark17). The circuit represen- tation of the SBOX is given in Figure [3](#_bookmark18).

**Table 6.** Lookup table representation of SBOX

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | a | b | c | d | e | f |
| 𝑥  SBOX(𝑥)  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 | 1e | 13 | 7 | e | 0 | d | 11 | 18 | 10 | c  𝑥 =1 | 1 | 19 | 16 | a  (0, 0, | f  0, 0, 1) | 17 |

Note that 5-bit inputs are represented in hexadecimal (e.g.,

*x*0 *x*1 *x*2 *x*3 *x*4



1

1

1

1

1

1

corresponds to ).

*y*0 *y*1 *y*2 *y*3 *y*4

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

### Linear Diffusion Layer 𝑝𝐿

𝑝𝐿 𝑆𝑖

The linear diffusion layer provides diffusion within each 64-bit word , as shown in Figure [4](#_bookmark20).

0

𝑆1

𝑆2

4

𝑆

|  |
| --- |
|  |
|  |
|  |
|  |
|  |

𝑆3

**Figure 4.** Application of linear diffusion layer 𝑝𝐿 to Ascon state

This layer applies the linear functions Σ𝑖 to their corresponding state words as 𝑆𝑖 ← Σ𝑖(𝑆𝑖)

for 0 ≤ 𝑖 ≤ 4, where each Σ is defined as:

𝑖

Σ0(𝑆0) = 𝑆0 ⊕ (𝑆0 ⋙ 19) ⊕ (𝑆0 ⋙ 28) (8)

Σ1(𝑆1) = 𝑆1 ⊕ (𝑆1 ⋙ 61) ⊕ (𝑆1 ⋙ 39) (9)

Σ2(𝑆2) = 𝑆2 ⊕ (𝑆2 ⋙ 1) ⊕ (𝑆2 ⋙ 6) (10)

Σ3(𝑆3) = 𝑆3 ⊕ (𝑆3 ⋙ 10) ⊕ (𝑆3 ⋙ 17) (11)

Σ4(𝑆4) = 𝑆4 ⊕ (𝑆4 ⋙ 7) ⊕ (𝑆4 ⋙ 41) (12)

## 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 prop- erties.

### Specification of Ascon-AEAD128

Ascon-AEAD128 consists of the encryption algorithm Ascon-AEAD128.enc (specified in Sec. [4.1.1](#_bookmark23)) and the decryption algorithm Ascon-AEAD128.dec (specified in Sec. [4.1.2](#_bookmark27)).

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](#_bookmark32) for the truncation option):

Ascon-AEAD128.enc(𝐾, 𝑁, 𝐴, 𝑃 ) = (𝐶, 𝑇 ) (13)

Ascon-AEAD128.dec takes key 𝐾, nonce 𝑁, associated data 𝐴, ciphertext 𝐶, and authen-

tication tag 𝑇 as inputs and outputs 𝑃 if the tag is valid:

Ascon-AEAD128.dec(𝐾, 𝑁, 𝐴, 𝐶, 𝑇 ) = {𝑃 if the tag 𝑇is valid (14)

fail otherwise

### 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](#_bookmark24)).

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

IV



*A*0

*Am*

*P C*

0 0

*P*

*n−*1 *n−*1

*C*

*P*˜*n C*˜*n*

*T*

128

⧸

128

⧸

128 ⧸

*ℓ* = *|P*˜*n|*

⧸

128-*ℓ*

128

⧸

⧸

⧸128

192

0127*−ℓ*

⧸

192

⧸

192

⧸

192

⧸

192

1

⧸

128

⧸

*K N*

064 *K*

0191 1

*K* 064

*K*

*Ascon*-*p*[12]

*Ascon*-*p*[8]

*Ascon*-*p*[8]

*Ascon*-*p*[8]

*Ascon*-*p*[8]

*Ascon*-*p*[12]

Initialization

Associated Data

Plaintext

Finalization

**Figure 5.** Ascon-AEAD128 encryption

The pseudocode of Ascon-AEAD128.enc is provided in Algorithm [3](#_bookmark25).

**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 ← 0x00001000808c0001 𝖰 Initialization

*S* ← 𝐼𝑉 ‖ 𝐾 ‖ 𝑁

*S* ← 𝐴𝑠𝑐𝑜𝑛1-9𝑝2[12](*S*)

*S* ← *S* ⊕ (0

**if** |𝐴𝐴| >, …0 ,**th**𝐴**en**

‖ 𝐾)

, 𝐴̃ ←

𝖰 Processing associated data

𝐴0 ←

**for** 𝑖 = 0 to 𝑚 **do**

𝑚

pad(𝐴𝑚, 128)

𝑚−̃1 𝑚

parse(𝐴, 128)

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[8]((*S*[0∶127] ⊕ 𝐴𝑖)‖ *S*[128∶319])

**end for**

**end if**

*S* ← *S* ⊕ (0319 ‖ 1)

𝑃0←, …|𝑃̃, 𝑃|𝑛−1, 𝑃̃𝑛 ← parse(𝑃 , 128) 𝖰 Processing plaintext

𝑙 𝑛

**for** 𝑖 = 0 to←𝑛 − 1 **do**

*S*𝐶[0∶←127] *S*[0∶127] ⊕ 𝑃𝑖

𝑖

*S*[0∶127]

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[8](*S*)

**end for**

*S*̃[0∶127] ← *S*[0∶127]⊕pad(𝑃𝑛, 128)

~

𝐶𝑛 ← *S*[0∶ℓ−1] ̃

𝐶 ← 𝐶0 ‖…‖ 𝐶𝑛−1 ‖ 𝐶𝑛

𝑇*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S* ⊕ (0128 ‖ 𝐾 ‖ 064)) 𝖰 Finalization

*S*[192∶319] ⊕ 𝐾

**return** 𝐶, 𝑇

* + - 1. **Initialization of the state.** Given a 128-bit 𝐾 and a 128-bit 𝑁, the 320-bit internal

state *S* is initialized as the concatenation of 𝐼𝑉, 𝐾, and 𝑁:

*S* ← 𝐼𝑉 ∥𝐾 ∥𝑁, (15)

𝐼𝑉

where the initialization value is 0x00001000808c0001 (see [Appendix B](#_bookmark93) for de- tails on determining the IV and [Appendix A](#_bookmark84) for implementation notes regarding initialization). Next, *S* is updated using the permutation - as

𝐴𝑠𝑐𝑜𝑛 𝑝[12]

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*) (16)

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

*S* ← *S* ⊕ (0192 ∥𝐾). (17)

* + - 1. **Processing associated data.** This step has two parts, including absorbing the asso- ciated 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 𝐴0, 𝐴1, …, 𝐴𝑚−1, 𝐴̃𝑚 ←

w12h8ere 𝑚 = ⌊|𝐴|/128⌋ and |𝐴𝑖| = 128 bits for 0 ≤̃ 𝑖 ≤ 𝑚 − 1 and 0 ≤ |𝐴𝑚| <

parse(𝐴, 128),

(18)

̃̃

, as explained in Algorithm [1](#_bookmark8). The last block 𝐴𝑚 can be empty. Next, 𝐴𝑚 is

padded as

𝐴𝑚 ← pad(𝐴̃𝑚, 128) = 𝐴̃𝑚||1 ∥ 0127−|𝐴̃𝑚| (19) so that |𝐴𝑚| = 128, as explained in Algorithm [2](#_bookmark9).

Each associated data block 𝐴𝑖 (0 ≤ 𝑖 ≤ 𝑚) is absorbed into the first 128 bits of

state as

*S*[0∶127] ← *S*[0∶127] ⊕ 𝐴𝑖, (20)

and the permutation 𝐴𝑠𝑐𝑜𝑛-𝑝[8] is applied to the state as

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[8](*S*). (21)

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

*S* ← *S* ⊕ (0319 ∥ 1) (22)

that provides domain separation.

* + - * + If the AD is empty (i.e., |𝐴| = 0): Only the final step described in ([22](#_bookmark26)) is applied.
      1. **Processing plaintext.** Plaintext 𝑃 (including empty plaintext) is parsed into blocks as

𝑃0, 𝑃1, …, 𝑃𝑛−1, 𝑃̃𝑛 ← parse(𝑃 , 128), (23)

where 𝑛 = ⌊|𝑃 |/128⌋, |𝑃𝑖| = 128 for 0 ≤ 𝑖 ≤ 𝑛 − 1, ãnd |𝑃̃𝑛| = 𝑙, 0 ≤ 𝑙 < 128 using For each 𝑃𝑖, 0 ≤ 𝑖 ≤ 𝑛 − 1, the state *S* is updated as

Algorithm [1](#_bookmark8). When |𝑃 | mod 128 = 0, the last block 𝑃𝑛 is empty.

*S*[0∶127] ← *S*[0∶127] ⊕ 𝑃𝑖, (24)

followed by generating the corresponding ciphertext block 𝐶𝑖 as

𝐶𝑖 ← *S*[0∶127], (25)

and the permutation 𝐴𝑠𝑐𝑜𝑛-𝑝[8] is applied to update the state as

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[8](*S*). (26)

For the last block 𝑃̃𝑛, the state is updated as ̃

*S*[0∶127] ← *S*[0∶127] ⊕ pad(𝑃𝑛, 128), (27)

and the last ciphertext block 𝐶̃𝑛 is obtained as

𝐶̃𝑛 ← *S*[0∶ℓ−1]. (28)

The ciphertext 𝐶 is constructed by concatenating the ciphertext blocks as

𝐶 ← 𝐶0 ∥… ∥𝐶𝑛−1 ∥𝐶̃𝑛. (29)

* + - 1. **Finalization and tag generation.** During finalization, the key is first loaded to the state *S* as

*S* ← *S* ⊕ (0128 ∥𝐾 ∥ 064), (30)

and the state *S* is then updated using the permutation 𝐴𝑠𝑐𝑜𝑛-𝑝[12] as

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*). (31)

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

𝑇 ← 𝑆[192∶319] ⊕ 𝐾. (32)

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

### Decryption

IV



*A*0

*Am*

*P*0 *C*0

*Pn−*1 *Cn−*1

*Pn Cn*

˜ ˜

128

*ℓ* = *|C*˜*n|*

*T′*

⧸

128

⧸

128 ⧸

⧸

128-*ℓ*

128

⧸

⧸

⧸128

192

⧸

192

⧸

192

1

0127*−ℓ*

⧸

192

⧸

192

⧸

128

⧸

*K N*

064 *K*

0191 1

*K* 064

*K*

*Ascon*-*p*[12]

*Ascon*-*p*[8]

*Ascon*-*p*[8]

*Ascon*-*p*[8]

*Ascon*-*p*[8]

*Ascon*-*p*[12]

Initialization

Associated Data

Ciphertext

Finalization

**Figure 6.** Ascon-AEAD128 decryption

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

The pseudocode of Ascon-AEAD128.dec is provided in Algorithm [4](#_bookmark29).

* + - 1. **Initialization of the state.** Given a 128-bit 𝐾 and 128-bit 𝑁, the 320-bit internal state

*S* is initialized as the concatenation of 𝐼𝑉, 𝐾, and 𝑁:

*S* ← 𝐼𝑉 ∥𝐾 ∥𝑁, (33)

where the initial value is 0x00001000808c0001 (see [Appendix B](#_bookmark93) for details on determining the IV and [Appendix A](#_bookmark84) for implementation notes regarding initialization). Next, *S* is updated using the permutation - as

[𝐼𝑉](#_bookmark84)

𝐴𝑠𝑐𝑜𝑛 𝑝[12]

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*) (34)

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

*S* ← *S* ⊕ (0192 ∥𝐾). (35)

This step is exactly the same as Step 1 of the encryption function in Sec. [4.1.1](#_bookmark23).

* + - 1. **Processing associated data.** This step has two parts, including absorbing the asso- ciated 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 𝐴0, 𝐴1, …, 𝐴𝑚−1, 𝐴̃𝑚 ←

parse(𝐴, 128),

(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 ← 0x00001000808c0001 𝖰 Initialization

*S* ← 𝐼𝑉 ‖ 𝐾 ‖ 𝑁

*S* ← 𝐴𝑠𝑐𝑜𝑛1-9𝑝2[12](*S*)

*S* ← *S* ⊕ (0

**if** |𝐴𝐴| >, …0 ,**th**𝐴**en**

‖ 𝐾)

, 𝐴̃ ←

𝖰 Processing associated data

𝐴0 ←

**for** 𝑖 = 0 to 𝑚 **do**

𝑚

pad(𝐴𝑚, 128)

𝑚−̃1 𝑚

parse(𝐴, 128)

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[8]((*S*[0∶127] ⊕ 𝐴𝑖)‖ *S*[128∶319])

**end for**

**end if**

*S* ← *S* ⊕ (0319 ‖ 1)

𝐶0,…, 𝐶𝑛−1, 𝐶̃𝑛 ← parse(𝐶, 128) 𝖰 Processing ciphertext

**for** 𝑖 = 0 to 𝑛 − 1 **do**

*S*[0∶127] ⊕ 𝐶𝑖

𝑃 ←

*S*[0∶127] ← 𝐶𝑖

𝑖

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[8](*S*)

**end for**

𝑙 = |𝐶̃𝑛|

*S*𝑛 ←[0∶ℓ*S*−1] ⊕𝑛 (1||0127−ℓ)

*S*[ℓ∶127] ← 𝐶̃[ℓ∶127]

𝑃̃ ← *S* ⊕ 𝐶̃

[0∶ℓ−1] 𝑛

𝑇*S* ′←←𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S* ⊕ (0128 ‖ 𝐾 ‖ 064)) 𝖰 Finalization

′ *S*[192∶319] ⊕ 𝐾

**if** 𝑇 𝑃=←=𝑃𝑇 **t**‖**h**…**en**‖ 𝑃 **else**

**return** 𝑃

0

**return** fail **end if**

𝑛−1

‖ 𝑃̃𝑛

w12h8ere 𝑚 = ⌊|𝐴|/128⌋ and |𝐴𝑖| = 128 bits for 0 ≤̃ 𝑖 ≤ 𝑚 − 1 and 0 ≤ |𝐴̃𝑚| <

[𝐴](#_bookmark9)̃, as explained in Algorithm [1](#_bookmark8). The last block 𝐴𝑚 can be empty.

𝑚 is further processed by padding to a full 𝜏 = 128-bit block using Algorithm

[2](#_bookmark9) as

𝐴𝑚 ← pad(𝐴̃𝑚, 128) = 𝐴̃𝑚||1 ∥ 0127−|𝐴̃𝑚|. (37)

The associated data blocks 𝐴𝑖’s (0 ≤ 𝑖 ≤ 𝑚) are absorbed into the state *S* as

follows:

*S*[0∶127] ← (*S*[0∶127] ⊕ 𝐴𝑖), (38)

and the permutation 𝐴𝑠𝑐𝑜𝑛-𝑝[8] is applied to the state as

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[8](*S*). (39)

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

*S* ← *S* ⊕ (0319 ∥ 1) (40)

for domain separation.

|𝐴| = 0

* + - * + If the AD is empty (i.e., ): Only the final step described in Equation ([40](#_bookmark30)) is applied.

This step is exactly the same as Step 2 of the encryption function in Sec. [4.1.1](#_bookmark23).

* + - 1. **Processing the ciphertext.** Ciphertext 𝐶 is parsed into blocks as

𝐶0, 𝐶1, …, 𝐶𝑛−1, 𝐶̃𝑛 ← parse(𝐶, 128), (41) where 𝑛 = ⌊|𝐶|/128⌋, |𝐶𝑖| = 128 for 0 ≤ 𝑖 ≤ 𝑛 − 1, |𝐶̃𝑛| =̃ 𝑙 and 0 ≤ 𝑙 < 128 using

Algorithm [1](#_bookmark8). Ciphertext 𝐶 or the last block of ciphertext 𝐶𝑛 can be empty.

For each 𝐶𝑖, 0 ≤ 𝑖 ≤ 𝑛 − 1, the following steps are applied:

𝑃𝑖 ← *S*[0∶127] ⊕ 𝐶𝑖 (42)

*S*[0∶127] ← 𝐶𝑖 (43)

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[8](*S*). (44)

For the last block of the ciphertext 𝐶̃𝑛 (with length 𝑙), the following steps are applied:

𝑃̃𝑛 ← *S*[0,ℓ−1] ⊕ 𝐶̃𝑛 (45)

*S*[0,ℓ−1] ← 𝐶̃𝑛 (46)

*S*[ℓ,127] ← *S*[ℓ,127] ⊕ (1||0127−ℓ). (47)

Note that when 𝐶̃𝑛 is an empty block, 𝑃̃𝑛 is an empty block as well.

The plaintext 𝑃 is constructed by concatenating the plaintext blocks as

𝑃 ← 𝑃0 ∥… ∥𝑃𝑛−1 ∥𝑃̃𝑛. (48)

* + - 1. **Finalization.** During finalization, the key is loaded into the state *S* as

*S* ← *S* ⊕ (0128 ∥𝐾 ∥ 064), (49)

and the state *S* is then updated using the permutation Ascon-p[12]as

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*). (50)

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

𝑇 ′ ← (𝑆[192∶319]) ⊕ 𝐾. (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.

### Implementation Options

### Truncation

Some applications may truncate the tag 𝑇 to a specific length 𝜆 (≤ |𝑇 |). The truncation

function outputs the leftmost 𝜆 bits of the tag (i.e., 𝑇 ).

[0∶𝜆−1]

The requirements on the tag lengths are provided in Sec. [4.3](#_bookmark35).

### Nonce Masking

In this option, an additional 128-bit key is used to mask the input nonce. Let 𝐾 = (𝐾1 ‖ 𝐾2)

be a 256-bit key, where |𝐾 | = |𝐾 | = 128. Ascon-AEAD128 with nonce masking is pro-

1 2

cessed as follows:

E(𝐾1 ∥𝐾2, 𝑁, 𝐴, 𝑃 ) = Ascon-AEAD128.enc(𝐾1, 𝑁 ⊕ 𝐾2, 𝐴, 𝑃 ), (52)

D(𝐾1 ∥𝐾2, 𝑁, 𝐴, 𝐶, 𝑇 ) = Ascon-AEAD128.dec(𝐾1, 𝑁 ⊕ 𝐾2, 𝐴, 𝐶, 𝑇 ) (53)

Ascon-AEAD128 with nonce masking should only be used when context-commitment secu- rity [2](#_bookmark36) [[12](#_bookmark71)] and related-key security [[13](#_bookmark72)] are not concerns. This is because the encryption of

Ascon-AEAD128 wit′h nonce masking alway″s ou′tputs the same (𝐶, 𝑇 ) p′air for′ two d″ifferent

input tuples (𝐾 ‖ 𝐾 , 𝑁, 𝐴, 𝑃 ) and (𝐾 ‖ 𝐾 , 𝑁 , 𝐴, 𝑃 ), where 𝑁 ⊕ 𝐾 = 𝑁 ⊕ 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](#_bookmark73)] (see Section [4.4.2](#_bookmark39)).

### 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](#_bookmark74)] 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., in Equation ([52](#_bookmark34))) **shall** also be kept secret.

𝑁 ⊕ 𝐾2

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

128

𝜆−32

𝜆 satisfies 64 ≤𝜆−𝜆3≤2

failures up to 2

, the probability of a forgery is low enough that decryption can be tolerated without compromising security. Therefore, the

maximum number of decryption failures under a fixed key **shall not** exceed 2 . For

32 ≤ 𝜆 < 64

shorter tags, with , 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 .

2𝜆−32

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

2

2In 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 54 bytes, and **should** be updated when the number of decryption failures reaches its limit.

2

### Security Properties

This section provides the security properties of Ascon-AEAD128 in various scenarios, in- cluding 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 Ascon-AEAD128 mode in both single-key and multi-key settings was evaluated in [[14](#_bookmark73), [16](#_bookmark75)–[19](#_bookmark77)]. The committing security of the Ascon-AEAD128 mode was also evaluated in [[20](#_bookmark78), [21](#_bookmark79)].

### Single-Key Setting

128

Ascon-AEAD128 with no tag truncation provides a -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 (i.e., blocks).

2−𝜆 𝜆

254 250

*Impact of truncation.* For a tag of length , a forgery attempt succeeds with a probability of

. 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, Ascon-AEAD128 with tag length provides -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 , provided that the secret key remains uncompromised. This also holds for the nonce masking option.

𝜆 𝜆

2−𝜆

### Multi-Key Setting

When 𝝊 keys are independently selected for an application, Ascon-AEAD128 with no tag

truncation provides a (128 − log2 𝝊)-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 1is28u−sed with 𝝊 keys, an attacker may discover one of the 𝝊 keys with

a time complexity of [[14](#_bookmark73), [17](#_bookmark76)–[19](#_bookmark77)].

2

log2 𝑢, thereby compromising both confidentiality and integrity

To improve security in a multi-key setting, the nonce-masking implementation option with no truncation (see Sec. [4.2.2](#_bookmark33)) can be used. This option provides 128-bit security (rather than log ) for confidentiality and integrity.

128 − 𝝊

2

*Impact of truncation.* In the multi-key setting, Ascon-AEAD128 with tag length 𝜆 pro-

vides min{128 − log2 𝝊, 𝜆}-bit security for both confidentiality and integrity in the nonce-

respecting setting. Additionally, when using the nonce-masking option with tag length 𝜆,

𝜆

it provides -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 , provided that the secret key is uncompromised.

2−𝜆

### 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 𝝊-key setting, where 𝝊 = 1 integrity guarantees of Ascon-AEAD128 and the nonce-masking option with a 𝜆-bit tag are

as follows:

{128 − 2 𝝊, 𝜆}

corresponds to a single key and 𝝊 > 1 to multiple independent keys, the confidentiality and

* Ascon-AEAD128 provides min log bits of security for both confidential- ity 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 p8er key, and that any

given nonce is reused for encryption with the same key no more than 2 times. Additionally,

even after a successful forgery, the probability of another successful forgery attempt remains at most , provided that none of the multiple keys is compromised. The resulting security levels under these conditions are summarized in Table [7](#_bookmark41).

2−𝜆

𝐴 𝜆 𝝊 𝑁

**T****able 7.** Security strength of Ascon-AEAD128 with -bit tag in the -key setting, where ( ,

) pairs are distinct for encryption per key

|  |  |  |
| --- | --- | --- |
| **Security** | **Security**  **strength in bits** | **Total number**  **of repetitions of a nonce** |
| Confidentiality of plaintext  𝑁, 𝐴, 𝐶, 𝑇 | min{{112288− log2𝝊,,𝜆}} | ≤ 28  8 |
| Integrity of ( ) | min log2 |  |

**When each (**𝑁**,** 𝐴**) pair are reused up to** 28 **times for encryption per key:** In the 𝝊-key of Ascon-AEAD128 and the nonce-masking option with a 𝜆-bit tag are as follows:

setting, where 𝝊 denotes the number of independent keys, the integrity security guarantees

* Ascon-AEAD128 provides min{128 − log2 𝝊, 𝜆} 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 28

times

for encryption with the same key. The corresponding integrity security levels are summa- rized in Table [8](#_bookmark42). 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 , provided that none of the multiple keys has been compromised.

𝝊

2−𝜆

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

|  |  |  |
| --- | --- | --- |
| **Security** | **Security strength**  **in bits** | **Total number of repetitions**  **of any (**𝑁**,**8𝐴**) pair** |
| Integrity of (𝑁, 𝐴, 𝐶, 𝑇) | min{128 − log2 𝝊, 𝜆} | ≤ 2 |

## Hash and eXtendable-Output Functions (XOFs)

𝐴𝑠𝑐𝑜𝑛 𝑝[12]

Hash and XOF algorithms are built on the - 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 - permutation, whereas Ascon-AEAD128 employs

both 𝐴𝑠𝑐𝑜𝑛-𝑝[12] and 𝐴𝑠𝑐𝑜𝑛-𝑝[8].

𝐴𝑠𝑐𝑜𝑛 𝑝[12]

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.

### Specification of Ascon-Hash256

The mode of operation used by Ascon-Hash256 and Ascon-XOF128 is shown in Figure [7](#_bookmark45). This mode comprises three main steps: initialization, absorbing the message, and squeezing the output. The length of the output is for Ascon-Hash256 and for Ascon- XOF128.

𝐿 256 𝐿 > 0

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

IV



*M*0

*Mn*−1

*Mn*

*H*0

6⧸4

*H*⌈*L/*64⌉−1

6⧸4

256

⧸

256

⧸

256

⧸

256

⧸

256

⧸

0256

*Ascon*-*p*[12]

*Ascon*-*p*[12]

*Ascon*-*p*[12]

*Ascon*-*p*[12]

*Ascon*-*p*[12]

Initialization

Absorb Message

Squeeze Output

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

Ascon-Hash256 takes a variable length message as input and produces a 256-bit digest. The full specification of Ascon-Hash256 can be found in Algorithm [5](#_bookmark48) and operates as follows:

𝑀

1. **Initialization.** The 320-bit internal state of Ascon-Hash256 is initialized with the concatenation of the 64-bit 0x0000080100cc0002 and 256 zeroes, followed

by the 𝐴𝑠𝑐𝑜𝑛-𝑝[12] permutation as

𝐼𝑉 =

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](𝐼𝑉 ∥ 0256). (54)

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

𝑀0,…, 𝑀𝑛−1, 𝑀̃𝑛 ← parse(𝑀 , 64), (55)

where |𝑀𝑖| = 64̃bits for 0 ≤ 𝑖 ≤ 𝑛 − 1 and 0 ≤ |𝑀̃𝑛| ≤ 63. The last block 𝑀̃𝑛 can

be empty. Next, 𝑀𝑛 is padded to create a full block 𝑀𝑛:

𝑀𝑛 ← pad(𝑀̃𝑛, 64). (56)

Each message block 𝑀𝑖 is XORed with the state as

*S*[0∶63] ← *S*[0∶63] ⊕ 𝑀𝑖. (57)

For all message blocks except the final block 𝑀𝑛,the XOR operation is immediately

followed by applying 𝐴𝑠𝑐𝑜𝑛-𝑝[12] to the state.

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*) (58)

1. **Squeezing the hash.** The squeezing phase begins after 𝑀𝑛 is absorbed with an

application of 𝐴𝑠𝑐𝑜𝑛-𝑝[12] to the state:

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*). (59)

The value of *S*[0∶63] is then taken as the 64-bit hash block 𝐻𝑖, and the state is again

updated by 𝐴𝑠𝑐𝑜𝑛-𝑝[12]: 𝐻𝑖 ←

*S*[0∶63]

(60)

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*). (61)

Steps ([60](#_bookmark46)) and ([61](#_bookmark47)) are repeated alternately until hash blocks and have been extracted. The final hash block is then extracted but is not followed by the Ascon-p[12] permutation:

𝐻0, 𝐻1, 𝐻2

𝐻3 ← *S*[0∶63]. (62)

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

𝐻 ← 𝐻0 ∥𝐻1 ∥𝐻2 ∥𝐻3. (63)

**Algorithm 5** Ascon-Hash256(𝑀) **Output:** Digest 𝐻 ∈ {0, 1}256

**Input:** Bitstring 𝑀 ∈ {0, 1}∗

𝐼𝑉 ←0x0000080100cc0205062 𝖰 Initialization

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](𝐼𝑉 ‖ 0 )

𝑀0,←… , 𝑀𝑛−̃1, 𝑀̃𝑛 ← parse(𝑀 , 64) 𝖰 Absorbing

𝑛

pad(𝑀𝑛, 64)

**for** 𝑖 = 0 to 𝑛 − 1 **do**

*S*[0∶63] ← *S*[0∶63] ⊕ 𝑀𝑖

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*)

**end for**

*S*[0∶63] ← *S*[0∶63] ⊕ 𝑀𝑛

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*) 𝖰 Squeezing

**for** 𝑖 = 0 to 2 **do**

𝐻 ←

𝑖

*S*[0∶63]

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*)

**end for**

𝐻3 ← *S*[0∶63]

𝐻 ← 𝐻0 ‖ 𝐻1 ‖ 𝐻2 ‖ 𝐻3

**return** 𝐻

### Specification of Ascon-XOF128

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

𝐿 > 0

1. Ascon-XOF128 accepts an additional input , which specifies the desired output length in bits.
2. The number of blocks that are squeezed is ⌈𝐿/64⌉.
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](#_bookmark52) and is described as follows:

1. **Initialization.** The 320-bit internal state of Ascon-XOF128 is initialized with the concatenation of the 64-bit 0x0000080000cc0003 and 256 zeroes, followed

by the 𝐴𝑠𝑐𝑜𝑛-𝑝[12] permutation:

𝐼𝑉 =

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](𝐼𝑉 ∥ 0256). (64)

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

𝑀0,…, 𝑀𝑛−1, 𝑀̃𝑛 ← parse(𝑀 , 64). (65)

where |𝑀𝑖| = 64 bits for 0 ≤ 𝑖 ≤ 𝑛 − 1 and 0 ≤ |𝑀̃𝑛| ≤ 63. Partial block 𝑀̃𝑛 is then

padded to a full block 𝑀𝑛 as

𝑀𝑛 ← pad(𝑀̃𝑛, 64). (66)

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

*S*[0∶63] ← *S*[0∶63] ⊕ 𝑀𝑖. (67)

For all message blocks except the final block, the XOR operation is immediately followed by an application of - to the state:

𝐴𝑠𝑐𝑜𝑛 𝑝[12]

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*). (68)

1. **Squeezing the outputs.** To obtain the requested output bits, blocks must be extracted from the state. The squeezing phase begins with an application of

𝐿 ℎ = ⌈𝐿/64⌉

𝐴𝑠𝑐𝑜𝑛 𝑝[12]

- to the state:

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*). (69)

The value of *S*[0∶63] is then taken as output block 𝐻𝑖, and the state is again updated

by 𝐴𝑠𝑐𝑜𝑛-𝑝[12]: 𝐻𝑖 ←

*S*[0∶63]

(70)

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*). (71)

Steps ([70](#_bookmark50)) and ([71](#_bookmark51)) are repeated alternately until output blocks have been squeezed. The final block is then squeezed without an additional permutation call:

𝐻0,…, 𝐻ℎ−1

𝐻ℎ ← *S*[0∶63]. (72)

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

𝐻:

𝐻′ ← 𝐻0 ∥… ∥𝐻ℎ (73)

𝐻 ← 𝐻[′0∶𝐿−1]. (74)

**Algorithm 6** Ascon-XOF128(𝑀, 𝐿)

**Output:** Digest 𝐻 ∈ {0, 1}𝐿

**Input:** Bitstring 𝑀 ∈ {0, 1}∗, output length 𝐿 > 0

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](𝐼𝑉 ‖ 0 )

𝐼𝑉 ← 0x0000080000cc0205603 𝖰 Initialization

𝑀0,←… , 𝑀𝑛−̃1, 𝑀̃𝑛 ← parse(𝑀 , 64) 𝖰 Absorbing

𝑛

pad(𝑀𝑛, 64)

**for** 𝑖 = 0 to 𝑛 − 1 **do**

*S*[0∶63] ← *S*[0∶63] ⊕ 𝑀𝑖

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*)

**end for**

*S*[0∶63] ← *S*[0∶63] ⊕ 𝑀𝑛

ℎ*S* ← ⌈𝐴𝐿𝑠/𝑐6𝑜4𝑛⌉-𝑝−[112](*S*) 𝖰 Squeezing

**for** 𝑖𝐻= ←0 to ℎ − 1 **do**

𝑖

*S*[0∶63]

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*)

**end for**

𝐻ℎ ← *S*[0∶63]

′ ← 𝐻0 ‖…‖ 𝐻ℎ [0∶𝐿−1]

𝐻 ← 𝐻′

𝐻

### return

### 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

0 as 𝑍0 =

s𝑍tring 𝑍 as input. After initialization, the length of 𝑍, in bits, is assigned to the 64-bit block

int64(|𝑍|).

(75)

Then, 𝑍 is parsed into blocks as ̃

𝑍1,…, 𝑍𝑚−1, 𝑍𝑚 ← parse(𝑍, 64), (76)

where |𝑍𝑖| = 64 bits for 0 ≤ 𝑖 ≤ 𝑚 − 1 and 0 ≤ |𝑍̃𝑚| ≤ 63. The partial block 𝑍̃𝑚 is then

padded to a full block 𝑍𝑚 as

𝑍

𝑚

← pad(𝑍̃𝑚, 64). (77)

The customization string 𝑍 is prepended to the message blocks as

𝑍0 ∥𝑍1 ∥… ∥𝑍𝑚 ∥𝑀0 ∥… ∥𝑀𝑛−1 ∥𝑀𝑛, (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 concate- nation 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](#_bookmark55) and the full specification is provided in Algorithm [7](#_bookmark54).

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

Ascon-CXOF128

**Input:** Bitstring 𝑀 ∈ {0, 1}∗, output length 𝐿 > 0, customization string 𝑍 ∈ {0, 1}∗, where

**Algorithm 7** (𝑀, 𝐿, 𝑍)

|𝑍| ≤ 2048

**Output:** Digest 𝐻 ∈ {0, 1}𝐿

𝐼𝑉 ← 0x0000080000cc0205604 𝖰 Initialization

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](𝐼𝑉 ‖ 0 )

𝑍0 …←, i𝑍nt64(,|𝑍̃|) ←

𝖰 Customization

1 p𝑚ad−(1𝑍̃𝑚, 64)

𝑍

𝑚

←

𝑚

parse(𝑍, 64)

**for** 𝑖 = 0 to 𝑚 **do**

*S*[0∶63] ← *S*[0∶63] ⊕ 𝑍𝑖

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*)

**end for**

𝑀0,←… , 𝑀𝑛−̃1, 𝑀̃𝑛 ← parse(𝑀 , 64) 𝖰 Absorbing

**for** 𝑖 = 0 to 𝑛 − 1 **do**

*S*[0∶63] ← *S*[0∶63] ⊕ 𝑀𝑖

𝑛

pad(𝑀𝑛, 64)

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*)

**end for**

*S*[0∶63] ← *S*[0∶63] ⊕ 𝑀𝑛

ℎ*S* ← ⌈𝐴𝐿𝑠/𝑐6𝑜4𝑛⌉-𝑝−[112](*S*) 𝖰 Squeezing

**for** 𝑖𝐻= ←0 to ℎ − 1 **do**

𝑖

*S*[0∶63]

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](*S*)

**end for**

𝐻ℎ ← *S*[0∶63]

′ ← 𝐻0 ‖…‖ 𝐻ℎ [0∶𝐿−1]

𝐻 ← 𝐻′

𝐻

### return

*Zm*−1



*Z*0

6⧸4

256

⧸

0256

6⧸4

*Ascon*-*p*[12]

*Zm*

*Mn*−1

*H*0 *H*⌈*L/*64⌉−1

256

*Ascon*-*p*[12]

*Ascon*-*p*[12]

⧸

IV

256

⧸

256

⧸

*Ascon*-*p*[12]

256

⧸

256

⧸

Initialization



*M*0

256

⧸

*Mn*

256

⧸

*Ascon*-*p*[12]

*Ascon*-*p*[12]

*Ascon*-*p*[12]

*Ascon*-*p*[12]

Customization

Absorb Message

Squeeze Output

**Figure 8.** Structure of Ascon-CXOF128

### 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 at the time that the final block is squeezed.

𝐿

The following three functions can be used to create incremental implementations for

Ascon-XOF128.

* 𝑐𝑡𝑥 ← Ascon-XOF128.Init()

Initializes and returns Ascon-XOF128 context 𝑐𝑡𝑥.

**Restriction:** Must be called exactly once before any call to Absorb or Squeeze.

* 𝑐𝑡𝑥 ← Ascon-XOF128.Absorb(𝑐𝑡𝑥, *str*)

Absorbs an arbitrary-length input *str* into the state and updates the context . **Restriction:** May be called multiple times after Init, but cannot be called after any call to Squeeze.

𝑐𝑡𝑥

* (𝑐𝑡𝑥, 𝑜𝝊𝑡) ← 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 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

𝑀

those proposed for SHAKE-128 and SHAKE-256 in [[22](#_bookmark80)]. Similar interfaces can be defined for incremental implementations of Ascon-AEAD128 and Ascon-Hash256.

### Security Strengths

The security strengths of Ascon-Hash256, Ascon-XOF128, and Ascon-CXOF128 are sum- marized in Table [9](#_bookmark58).

**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  𝐿 | 128  𝐿/2 | 128  𝐿 | 128  𝐿 |
| Ascon-XOF128 | 𝐿 | min( ,128)  𝐿/2 | min( ,128)  𝐿 | min( ,128)  𝐿 |
| Ascon-CXOF128 |  | min( ,128) | min( ,128) | min( ,128) |

If the message is known to belong to a set *M*, the preimage resistance is also limited by the size of *M*. For more information about security strengths against preimage attacks in different scenarios, refer to [[19](#_bookmark77), [23](#_bookmark81)].

## Conformance

The implementations of Ascon-AEAD128, Ascon-Hash256, Ascon-XOF128, and Ascon- CXOF128 may be tested for conformance to this standard under the Cryptographic Valida- tion Program [[24](#_bookmark82)]. Example test vectors are available from the Cryptographic Algorithm Validation Program (CAVP) [[25](#_bookmark83)].

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. Simi- larly, the use of Ascon-XOF128and 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/roun](https://competitions.cr.yp.to/round1/asconv1.pdf) [d1/asconv1.pdf](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/roun](https://competitions.cr.yp.to/round2/asconv11.pdf) [d2/asconv11.pdf](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/roun](https://competitions.cr.yp.to/round3/asconv12.pdf) [d3/asconv12.pdf](https://competitions.cr.yp.to/round3/asconv12.pdf).
4. National Institute of Standards and Technology (Published 2001; Updated 2023) Ad- vanced Encryption Standard (AES), FIPS 197. [https://doi.org/10.6028/NIST.FIPS.197-u](https://doi.org/10.6028/NIST.FIPS.197-upd1) [pd1](https://doi.org/10.6028/NIST.FIPS.197-upd1).
5. Dworkin MJ (2007) Recommendation for Block Cipher Modes of Operation: Ga- lois/Counter Mode (GCM) and GMAC (National Institute of Standards and Technology),

Report. [DOI:10.6028/NIST.SP.800-38D](https://doi.org/10.6028/NIST.SP.800-38D)

1. National Institute of Standards and Technology (2015) Secure Hash Standard (SHS) (U.S. Department of Commerce), Report. [DOI:10.6028/NIST.FIPS.180-4](https://doi.org/10.6028/NIST.FIPS.180-4)
2. 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](https://doi.org/10.6028/NIST.FIPS.202)

1. 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:](https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf)

[//csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-r](https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf) [ound/updated-spec-doc/ascon-spec-final.pdf](https://csrc.nist.gov/CSRC/media/Projects/lightweight-cryptography/documents/finalist-round/updated-spec-doc/ascon-spec-final.pdf).

1. Sönmez Turan M, McKay KA, Çalık Ç, Chang D, Bassham I Lawrence E (2019) Status Re- port on the First Round of the NIST Lightweight Cryptography Standardization Process (National Institute of Standards and Technology), Report. [DOI:10.6028/NIST.IR.8268](https://doi.org/10.6028/NIST.IR.8268)
2. 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](https://doi.org/10.6028/NIST.IR.8369)

1. 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](https://doi.org/10.6028/NIST.IR.8454)

1. Bellare M, Hoang VT (2022) Efficient Schemes for Committing Authenticated Encryp- tion. *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-](https://doi.org/10.1007/978-3-031-07085-3_29) [07085-3\_29](https://doi.org/10.1007/978-3-031-07085-3_29)

1. 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](https://doi.org/10.1007/3-540-39200-9_31). Available at [https:](https://doi.org/10.1007/3-540-39200-9_31)

[//doi.org/10.1007/3-540-39200-9\_31](https://doi.org/10.1007/3-540-39200-9_31)

1. 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](https://doi.org/10.1007/978-3-031-54773-7_18)
2. Barker E, Roginsky A, Davis R (2020) Recommendation for Cryptographic Key Genera- tion, (National Institute of Standards and Technology, Gaithersburg, MD), NIST Special Publication (SP) 800-133 Rev. 2. [DOI:10.6028/NIST.SP.800-133r2](https://doi.org/10.6028/NIST.SP.800-133r2).
3. 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](https://doi.org/10.1007/978-981-99-8727-6_12)
4. 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.
5. 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-](https://doi.org/10.1007/978-981-97-5025-2_4) [981-97-5025-2\_4](https://doi.org/10.1007/978-981-97-5025-2_4)
6. 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](https://doi.org/10.46586/TOSC.V2025.I1.138-210). Available at [https:](https://doi.org/10.46586/tosc.v2025.i1.138-210)

[//doi.org/10.46586/tosc.v2025.i1.138-210](https://doi.org/10.46586/tosc.v2025.i1.138-210)

1. 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](https://doi.org/10.46586/TOSC.V2023.I4.420-451). Available at [https://doi.org/10.46586/tosc.v2](https://doi.org/10.46586/tosc.v2023.i4.420-451) [023.i4.420-451](https://doi.org/10.46586/tosc.v2023.i4.420-451)
2. 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](https://doi.org/10.46586/TOSC.V2024.I4.191-248). Available at [https://doi.org/10.46586/tosc.v2](https://doi.org/10.46586/tosc.v2024.i4.191-248) [024.i4.191-248](https://doi.org/10.46586/tosc.v2024.i4.191-248)
3. 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](https://doi.org/10.6028/NIST.FIPS.203)

1. Lefevre C, Mennink B (2022) Tight Preimage Resistance of the Sponge Construc- tion. *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptol- ogy Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Pro- ceedings, 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](https://doi.org/10.1007/978-3-031-15985-5_7). Available at <https://doi.org/10.1007/978-3-031-15985-5_7>
2. National Institute of Standards and Technology (2024) Cryptographic Module Valida- tion Program (CMVP). Available at [https://csrc.nist.gov/projects/cryptographic-mod](https://csrc.nist.gov/projects/cryptographic-module-validation-program) [ule-validation-program](https://csrc.nist.gov/projects/cryptographic-module-validation-program).
3. National Institute of Standards and Technology (2023) GitHub repository usnist- gov/ACVP-Server: Automated Cryptographic Validation Test System — Gen/Vals). Avail- able 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 ma- chines, 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 re- versed. The hexadecimal forms of the padding for Ascon functions are described in Sec. [A.2](#_bookmark88).

However, the convention for printing the Ascon state using 64-bit integer words in hex- adecimal 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](#_bookmark86).

The least significant bit of 𝑆0 is 𝑠(0,0) (i.e., *S*[0∶0]), and the most significant bit of 𝑆4 is 𝑠(4,63)

(i.e., *S* ). Similarly, the least significant byte of 𝑆 is the first byte of state (*S* ),

[319∶319] 0 [0∶7]

and the most significant byte of 𝑆4 is the last byte of the state (*S*[312∶319]). This relationship

between state words, bytes, and state bits is shown in Fig. [9](#_bookmark85), where 𝑆𝑖[𝑗] denotes the 𝑗𝑡ℎ

byte of state word 𝑆 for 0 ≤ 𝑖 ≤ 4 and 0 ≤ 𝑗 ≤ 7.

𝑖

|  |  |  |
| --- | --- | --- |
| 𝑆0 | … | 𝑆4 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 𝑆0[0] | 𝑆0[1] | 𝑆0[2] | 𝑆0[3] | 𝑆0[4] | 𝑆0[5] | 𝑆0[6] | 𝑆0[7] |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 𝑆4[0] | 𝑆4[1] | 𝑆4[2] | 𝑆4[3] | 𝑆4[4] | 𝑆4[5] | 𝑆4[6] | 𝑆4[7] |

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

### 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 0 in little-endian notation (i.e., without byte reversal on little-endian

machines). The next eight bytes are loaded to 𝑆1. Bytes continue to be loaded in the same

𝑆

way until the final eight bytes of the stored state are loaded into 𝑆4.

An example of the mapping between memory addresses to state word bytes is presented in Table [10](#_bookmark87) 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](#_bookmark89). 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 𝑆𝑖 in memory on little-endian and

big-endian machines, where the word 𝑆 begins at memory address 𝑎

𝑖

### Word Little-endian Big-endian byte address address

𝑆𝑖[2] 𝑎 + 2 𝑎 + 5

𝑆𝑖[01] 𝑎 + 01 𝑎 + 76

𝑆𝑖[3] 𝑎 + 3 𝑎 + 4

𝑆𝑖[4] 𝑎 + 4 𝑎 + 3

𝑆𝑖[5] 𝑎 + 5 𝑎 + 2

𝑆𝑖[6] 𝑎 + 6 𝑎 + 1

𝑆𝑖[7] 𝑎 + 7 𝑎 + 0

𝑖

**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.

### 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](#_bookmark9) appends a one followed by one or

𝑥 𝑛 < 8

𝑦more zeroes to data. For an integer that can be represented with bytes, an integer

that represents a padded version of 𝑥 is computed as

𝑦 ← 𝑥 ⊕ (0x0000000000000001 ≪ 8𝑛).

|  |  |  |
| --- | --- | --- |
| **State**  **bits** | **State**  **word** | **Word value (64-bit unsigned integers)** |
| *S*[0∶63] | 𝑆0 | 0x0706050403020100 |
| *S*[64∶127] | 𝑆1 | 0x0F0E0D0C0B0A0908 |
| *S*[128∶191] | 𝑆2 | 0x1716151413121110 |
| *S*[192∶255] | 𝑆3 | 0x1F1E1D1C1B1A1918 |
| *S*[256∶319] | 4 | 0x2726252423222120  ↕ |

|  |  |  |
| --- | --- | --- |
| **State**  **bits** | **State**  **word** | **Word value (byte sequence)** |
| *S*[0∶63] | 𝑆0 | 0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 |
| *S*[64∶127] | 𝑆1 | 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F |
| *S*[128∶191] | 𝑆2 | 0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 |
| *S*[192∶255] | 𝑆3 | 0x18 0x19 0x1A 0x1B 0x1C 0x1D 0x1E 0x1F |
| *S*[256∶319] | 4 | 0x20 0x21 0x22 0x23 0x24 0x25 0x26 0x27  ↕ |

|  |  |  |
| --- | --- | --- |
| **State**  **bits** | **State**  **word** | **Word value (bitstring)** |
| *S*[0∶63] | 𝑆0  𝑆 | 0000 0000 1000 0000 0100 0000 1100 0000  0010 0000 1010 0000 0110 0000 1110 0000 |
| *S*[64∶127] | 1  𝑆 | 0001 0000 1001 0000 0101 0000 1101 0000  0011 0000 1011 0000 0111 0000 1111 0000 |
| *S*[128∶191] | 2  𝑆 | 0000 1000 1000 1000 0100 1000 1100 1000  0010 1000 1010 1000 0110 1000 1110 1000 |
| *S*[192∶255] | 3  𝑆 | 0001 1000 1001 1000 0101 1000 1101 1000  0011 1000 1011 1000 0111 1000 1111 1000 |
| *S*[256∶319] | 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 to a 64-bit block, where encodes a sequence of bytes that each have value 0xFF in little-endian byte order

𝑥 𝑥

|  |  |  |  |
| --- | --- | --- | --- |
| **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 imple- mented as

𝑆4 ← 𝑆4 ⊕ 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

𝑆0 ← 𝑆0 ⊕ 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

𝑆0 ← 𝑆0 ⊕ 0x0706050403020100

1

1

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

𝑆1 ← 𝑆1 ⊕ 0x0706050403020100

2

2

0x0F0E0D0C0B0A0908.

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

𝑆3 ← 𝑆3 ⊕ 0x0706050403020100

4

4

0x0F0E0D0C0B0A0908.

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

𝑆2 ← 𝑆2 ⊕ 0x0706050403020100

3

3

0x0F0E0D0C0B0A0908.

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

4 4 0x0F0E0D0C0B0A0908.

𝑆3 ← 𝑆3 ⊕ 0x0706050403020100

### Precomputation

The initialization phases of Ascon-Hash256, Ascon-XOF128, and Ascon-CXOF128 are in- dependent 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](#_bookmark92) 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](#_bookmark48)

𝐼𝑉 ← 0x0000080100cc0002

*S* ← 𝐴𝑠𝑐𝑜𝑛-𝑝[12](𝐼𝑉 ∥ 0256)

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

𝑆0 ← 0x9b1e5494e934d681

𝑆1 ← 0x4bc3a01e333751d2

2 0xae65396c6b34b81a

𝑆 ←

𝑆3 ← 0x3c7fd4a4d56a4db3

4

0x1a5c464906c5976d.

The same is true for Ascon-XOF128 and Ascon-CXOF128 using the corresponding values given in Table [12](#_bookmark92).

It may also be beneficial to precompute the intermediate state between the customiza- tion 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** |
| 𝑆0 | 0x9b1e5494e934d681 | 0xda82ce768d9447eb | 0x675527c2a0e8de03 |
| 𝑆1 | 0x4bc3a01e333751d2 | 0xcc7ce6c75f1ef969 | 0x43d12d7dc0377bbc |
| 𝑆2 | 0xae65396c6b34b81a | 0xe7508fd780085631 | 0xe9901dec426e81b5 |
| 𝑆3 | 0x3c7fd4a4d56a4db3 | 0x0ee0ea53416b58cc | 0x2ab14907720780b6 |
| 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∥ 08 ∥𝑎∥𝑏 ∥𝑡∥𝜏/8 ∥ 016, (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)

𝜏/8

* is the number of input bytes processed per invocation of the underlying permu- tation (represented in 8 bits)

The values of these parameters for each variant are given in Table [13](#_bookmark94), and the initial values for each Ascon variant are specified in Table [14](#_bookmark95).

**Table 13.** Parameters for initial value construction

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Ascon variants** | 𝑣  (8 bits) | 𝑎  (4 bits) | 𝑏  (4 bits) | 𝑡  (16 bits) | 𝜏/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 |