INTERNATIONAL STANDARD

ISO/IEC 29192-5

First edition
2016-08-01

Information technology — Security techniques — Lightweight cryptography —

Part 5: Hash-functions

Technologies de l'information — Techniques de sécurité — Cryptographie pour environnements contraints —

Partie 5: Fonctions de hachage

Reference number
ISO/IEC 29192-5:2016(E)

Contents

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT) see the following URL: Foreword - Supplementary information

The committee responsible for this document is ISO/IEC JTC 1, *Information technology*, SC 27, *IT Security techniques*.

ISO/IEC 29192 consists of the following parts, under the general title *Information technology — Security techniques — Lightweight cryptography*:

— *Part 1: General*

— *Part 2: Block ciphers*

— *Part 3: Stream ciphers*

— *Part 4: Mechanisms using asymmetric techniques*

— *Part 5: Hash-functions*

Further parts may follow.

Introduction

This part of ISO/IEC 29192 specifies lightweight hash-functions, which are tailored for implementation in constrained environments.

ISO/IEC 29192-1 specifies the requirements for lightweight cryptography.

A hash-function maps an arbitrary string of bits to a fixed-length string of bits.

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this part of ISO/IEC 29192 may involve the use of patents. The ISO and IEC take no position concerning the evidence, validity and scope of these patent rights.

The holders of these patent rights have assured the ISO and IEC that they are willing to negotiate licences under reasonable and non-discriminatory terms and conditions with applicants throughout the world.

In this respect, the statements of the holders of these patent rights are registered with the ISO and IEC. Information may be obtained from the following:

Nanyang Technological University - NTUitive Pte Ltd

16 Nanyang Drive, #01-109, Innovation Centre, Singapore 637722

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights other than those identified above. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO (www.iso.org/patents) and IEC (http://patents.iec.ch) maintain on-line databases of patents relevant to their standards. Users are encouraged to consult the databases for the most up to date information concerning patents.

Information technology — Security techniques — Lightweight cryptography —

Part 5: Hash-functions

1 Scope

This part of ISO/IEC 29192 specifies three hash-functions suitable for applications requiring lightweight cryptographic implementations.

— PHOTON: a lightweight hash-function with permutation sizes of 100, 144, 196, 256 and 288 bits computing hash-codes of length 80, 128, 160, 224, and 256 bits, respectively.

— SPONGENT: a lightweight hash-function with permutation sizes of 88, 136, 176, 240 and 272 bits computing hash-codes of length 88, 128, 160, 224, and 256 bits, respectively.

— Lesamnta-LW: a lightweight hash-function with permutation size 384 bits computing a hash-code of length 256 bits.

The requirements for lightweight cryptography are given in ISO/IEC 29192-1.

2 Normative references

The following documents, in whole or in part, are normatively referenced in this document and are indispensable for its application. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 29192-1, *Information technology — Security techniques — Lightweight cryptography — Part 1: General*

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1
absorbing phase
input phase of a sponge function

[SOURCE: [4]]

3.2
bitrate
part of the internal state of a sponge function of length r bits

[SOURCE: [4]]

3.3
capacity
part of the internal state of a sponge function of length c bits

[SOURCE: [4]]

3.4
collision resistance
computationally infeasible to find any two distinct inputs which map to the same output of a hash-function

Note 1 to entry: Computational feasibility depends on the specific security requirements and environment.

3.5
hash-code
string of bits which is the output of a hash-function

Note 1 to entry: The literature on this subject contains a variety of terms that have the same or similar meaning as hash-code. Modification Detection Code, Manipulation Detection Code, digest, hash-result, hash-value and imprint are some examples.

[SOURCE: ISO/IEC 10118-1:—[1)], 2.3]

3.6
hash-function
function which maps strings of bits to fixed-length strings of bits, satisfying the following two properties:

— it is computationally infeasible to find for a given output, an input which maps to this output;

— it is computationally infeasible to find for a given input, a second input which maps to the same output

Note 1 to entry: Computational feasibility depends on the specific security requirements and environment.

[SOURCE: ISO/IEC 10118-1:—[1)], 2.4]

3.7
initializing value
value used in defining the starting point of a hash-function

Note 1 to entry: The literature on this subject contains a variety of terms that have the same or similar meaning as initializing value. Initialization vector and starting value are examples.

[SOURCE: ISO/IEC 10118-1:—[1)], 2.5]

3.8
preimage resistance
computationally infeasible to find for a given output of a hash-function, an input which maps to this output

Note 1 to entry: Computational feasibility depends on the specific security requirements and environment.

3.9
second preimage resistance
computationally infeasible to find for a given input of a hash-function, a second input which maps to the same output

Note 1 to entry: Computational feasibility depends on the specific security requirements and environment.

3.10
sponge function
mode of operation, based on a fixed-length permutation (or transformation) and a padding rule, which builds a function mapping variable-length input to variable-length output

[SOURCE: [4]]

1) To be published. (Revision of ISO/IEC 10118-1:2000)

3.11
squeezing phase
output phase of a sponge function

[SOURCE: [4]]

4 Symbols

$\{0\}^c$	bit-string containing exactly c zeros
0x	prefix indicating a binary string in hexadecimal notation
$\|\|$	concatenation of bit strings
$a \leftarrow b$	set variable a to the value of b
$\oplus$	bitwise exclusive-OR operation
c	length of the capacity in bits
hash	n-bit hash-code
IV	t-bit initialization value
m_i	message block i of r bits
n	length of the hash code in bits
r	length of the bitrate in bits
S_i	t-bit internal state at iteration i
t	length of the internal state in bits
$\lceil x \rceil$	the smallest integer greater than or equal to the real number x

5 Lightweight hash-functions optimized for hardware implementations

5.1 General

Clause 5 specifies PHOTON and SPONGENT hash-functions which are optimized for hardware implementations. ISO/IEC 29192-1 shall be referred to for the requirements for lightweight cryptography.

5.2 PHOTON

5.2.1 General

In order to cover a wide spectrum of applications, five different variants of PHOTON[5] are specified. Each variant is defined by its internal permutation size $t = c + r$, where c and r denote the *capacity* and the *bitrate*, respectively. For a fixed permutation size t, the choice of c and r provides a security-efficiency trade-off. PHOTON-t denotes the variant using a t-bit internal permutation.

The five variants are the following:

a) **PHOTON-100** computes an 80-bit hash-code and offers 64-bit preimage resistance, 40-bit second preimage resistance, and 40-bit collision resistance.

b) **PHOTON-144** computes a 128-bit hash-code and offers 112-bit preimage resistance, 64-bit second preimage resistance, and 64-bit collision resistance.

c) **PHOTON-196** computes a 160-bit hash-code and offers 124-bit preimage resistance, 80-bit second preimage resistance, and 80-bit collision resistance.

d) **PHOTON-256** computes a 224-bit hash-code and offers 192-bit preimage, 112-bit second preimage resistance, and 112-bit collision resistance.

e) **PHOTON-288** computes a 256-bit hash-code and offers 224-bit preimage, 128-bit second preimage resistance, and 128-bit collision resistance.

PHOTON-100 does not provide the minimum security strength as required in ISO/IEC 29192-1. It shall not be used as a general purpose hash function. PHOTON-144 does not provide the minimum security strength for collision resistance and second preimage resistance as required in ISO/IEC 29192-1. It shall only be used in applications where collision resistance and second preimage resistance are not required.

5.2.2 PHOTON specific notation

P_t internal permutation, where $t \in \{100,144,196,256,288\}$

z_i the r' leftmost bits of the internal state S

c' length of the capacity in bits during the squeezing phase of PHOTON

d number of rows and columns of the internal state matrix

r' length of the bitrate in bits during the squeezing phase of PHOTON

$S[i,j]$ the s-bit internal state cell located at row i and column j, with $0 \le i, j < d$

$RC(v)$ round constant of round v

$IC_d(i)$ internal constants of row i

X_r 3-bit or 4-bit internal state of a shift register to generate the round constants $RC(v)$ or the internal constants $IC_d(i)$

$FB()$ feedback function to update the internal state of a shift register

$\mathrm{SBOX}_{\mathrm{PRE}}$ the 4-bit substitution table (S-box) also used in the block cipher PRESENT[1]

$\mathrm{SBOX}_{\mathrm{AES}}$ the 8-bit substitution table (S-box) also used in the Advanced Encryption Algorithm[2]

5.2.3 Domain extension algorithm

The message M to hash is first padded by appending a “1” bit and as many zeros (possibly none), such that the total length is a multiple of the bitrate, r, and finally l message blocks $m_0,\dots, m_{l-1}$ of r bits each can be obtained. The t-bit internal state, S, is initialized by setting it to the value $S_0=IV=\{0\}^{t-24}||n/4||r||r'$, where each value is coded on 8 bits.

NOTE For implementation purposes, each byte is interpreted in big-endian form, that is, the leftmost bit is the most significant bit.

Then, as for the classical sponge strategy, at iteration i the message block m_i is absorbed on the leftmost part of the internal state S_i and then the permutation P_t is applied, i.e.

$$S_{i+1} \leftarrow P_t(S_i \oplus (m_i \| \{0\}^c)).$$

Once all l message blocks have been absorbed, the hash value is built by concatenating the successive r'-bit output blocks z_i until the appropriate output size n is reached:

$$\text{hash} = z_0 \| \ldots \| z_{l'-1}$$

with the rightmost bits truncated if necessary to produce an n-bit hash. More precisely, z_i is the r' leftmost bits of the internal state S_{l+i} and $S_{l+i+1} \leftarrow P_t(S_{l+i})$ for $0 \le i < l'$, where l' denotes the number of squeezing iterations, that is $l' = \lceil n / r' \rceil - 1$. If the hash output size is not a multiple of r', one just truncates $z_{l'-1}$ to $n \bmod r'$ bits.

5.2.4 Internal permutation

5.2.4.1 General

The internal permutations P_t, where $t \in \{100,144,196,256,288\}$, are applied to an internal state of d^2 elements of s bits each, which can be represented as a $(d \times d)$ matrix. P_t is composed of N_r rounds, each containing four layers as depicted in Figure 1:

a) AddConstants (AC),

b) SubCells (SC),

c) ShiftRows (ShR), and

d) MixColumnsSerial (MCS).

Table 1 shows an overview of the parameters of the different variants of PHOTON.

Table 1 — Overview of parameters of PHOTON

Variant	t	c	r	r'	d	s	N_r	$IC_d(.)$	Irr. polynomial	Z_i coefficients
PHOTON-100	100	80	20	16	5	4	12	[0, 1, 3, 6, 4]	$x^4 + x + 1$	(1, 2, 9, 9, 2)
PHOTON-144	144	128	16	16	6	4	12	[0,1, 3, 7, 6, 4]	$x^4 + x + 1$	(1, 2, 8, 5, 8, 2)
PHOTON-196	196	160	36	36	7	4	12	[0,1, 2, 5, 3, 6, 4]	$x^4 + x + 1$	(1, 4, 6, 1, 1, 6, 4)
PHOTON-256	256	224	32	32	8	4	12	[0,1, 3, 7, 15, 14, 12, 8]	$x^4 + x + 1$	(2, 4, 2, 11, 2, 8, 5, 6)
PHOTON-288	288	256	32	32	6	8	12	[0, 1, 3, 7, 6, 4]	$x^8 + x^4 + x^3 + x + 1$	(2, 3, 1, 2, 1, 4)

NOTE Always a cell size of 4 bits is used, except for the largest version for which 8-bit cells are used, and that the number of rounds is always $N_r = 12$ for all values of t. The output rate r' is always the same as the input rate r, except for PHOTON-100. The internal state cell located at row i and column j is denoted $S[i,j]$ with $0 \le i, j < d$.

Informally, AddConstants simply consists in adding fixed values to the cells of the internal state, while SubCells applies an s-bit S-box to each of them. ShiftRows rotates the position of the cells in each of the rows and MixColumnsSerial linearly mixes all the columns independently.

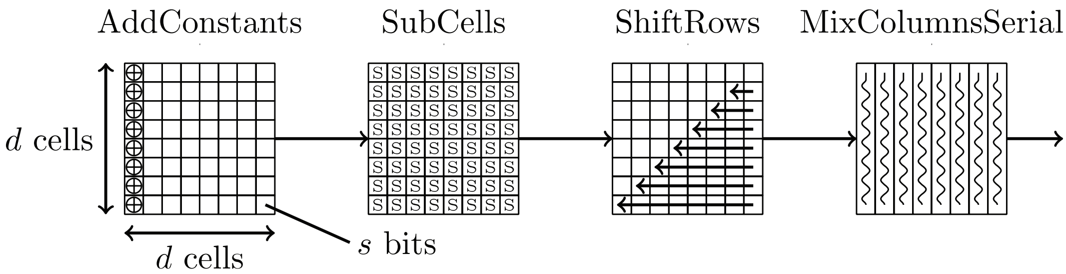

Figure 1 — One round of a PHOTON permutation

5.2.4.2 AddConstants

At round number v (start the counting from 1), first a round constant $RC(v)$ is XORed to each cell $S[i,0]$ of the first column of the internal state. Then, distinct internal constants $IC_d(i)$ are XORed to each cell $S[i,0]$ of the same first column. Overall, for round v it holds that

$$S'\left[i,0\right] \leftarrow S\left[i,0\right] \oplus RC\left(v\right) \oplus IC_d\left(i\right) \text{ for all } 0 \leq i < d.$$

The round constants $RC(v)$ have been generated by a 4-bit linear feedback shift register with maximum cycle length; they are

$$RC\left(v\right) \;=\; \left[1,\ 3,\ 7,\ 14,\ 13,\ 11,\ 6,\ 12,\ 9,\ 2,\ 5,\ 10\right].$$

The internal constants, $IC_d(i)$, depend on the square size d and on the row position i and they have been generated by shift registers with a cycle length of d. For all variants shift registers with $l = 3$ bits are used, except for $d = 8$, where $l = 4$ is used. The internal state of the shift register is denoted with $X_r = (x_{l-1}, ..., x_1, x_0)$, where each $x_i = \{0,1\}$, and the state is initialized with all 0's, that is $X_0 = (0, . . . , 0, 0)$. Then in each update iteration the new content of the shift register is given by $X_{r+1} \leftarrow (x_{l-2}, ..., x_0, FB(X_r))$, where $FB(X_r)$ is the feedback function. The round constants are computed by $FB(X_r) = x_3$ XNOR x_2, while the feedback functions for the internal constants are shown in Table 2. Constants for all square sizes, round numbers, and row positions are displayed in Table 3 through Table 6.

Table 2 — Feedback functions for internal constants generation

d	**5**	**6**	**7**	**8**
$FB(X_r)$	x_2 NOR x_1	NOT x_2	x_2 XNOR x_0	NOT x_3
$IC_d(.)$	[0, 1, 3, 6, 4]	[0, 1, 3, 7, 6, 4]	[0, 1, 2, 5, 3, 6, 4]	[0, 1, 3, 7, 15, 14, 12, 8]

Table 3 — $RC(v) \oplus IC_d(i)$ for d = 5

Round v / Row i	1	2	3	4	5	6	7	8	9	10	11	12
0	1	3	7	14	13	11	6	12	9	2	5	10
1	0	2	6	15	12	10	7	13	8	3	4	11
2	2	0	4	13	14	8	5	15	10	1	6	9
3	7	5	1	8	11	13	0	10	15	4	3	12
4	5	7	3	10	9	15	2	8	13	6	1	14

Table 4 — $RC(v) \oplus IC_d(i)$ for $d = 6$

Round v / Row i	1	2	3	4	5	6	7	8	9	10	11	12
0	1	3	7	14	13	11	6	12	9	2	5	10
1	0	2	6	15	12	10	7	13	8	3	4	11
2	2	0	4	13	14	8	5	15	10	1	6	9
3	6	4	0	9	10	12	1	11	14	5	2	13
4	7	5	1	8	11	13	0	10	15	4	3	12
5	5	7	3	10	9	15	2	8	13	6	1	14

Table 5 — $RC(v) \oplus IC_d(i)$ for $d = 7$

Round v / Row i	1	2	3	4	5	6	7	8	9	10	11	12
0	1	3	7	14	13	11	6	12	9	2	5	10
1	0	2	6	15	12	10	7	13	8	3	4	11
2	3	1	5	12	15	9	4	14	11	0	7	8
3	4	6	2	11	8	14	3	9	12	7	0	15
4	2	0	4	13	14	8	5	15	10	1	6	9
5	7	5	1	8	11	13	0	10	15	4	3	12
6	5	7	3	10	9	15	2	8	13	6	1	14

Table 6 — $RC(v) \oplus IC_d(i)$ for $d = 8$

Round v / Row i	1	2	3	4	5	6	7	8	9	10	11	12
0	1	3	7	14	13	11	6	12	9	2	5	10
1	0	2	6	15	12	10	7	13	8	3	4	11
2	2	0	4	13	14	8	5	15	10	1	6	9
3	6	4	0	9	10	12	1	11	14	5	2	13
4	14	12	8	1	2	4	9	3	6	13	10	5
5	15	13	9	0	3	5	8	2	7	12	11	4
6	13	15	11	2	1	7	10	0	5	14	9	6
7	9	11	15	6	5	3	14	4	1	10	13	2

5.2.4.3 SubCells

This layer simply applies an s-bit S-box to each of the cells of the internal state, i.e.

$$S'\left[i, j\right] \leftarrow \mathrm{SBOX}(S[i, j])\, for\ all\ 0 \leq i,\ j < d.$$

For PHOTON-100, PHOTON-144, PHOTO-196, and PHOTON-256, the PRESENT S-box $\mathrm{SBOX}_{\mathrm{PRE}}$[1] is used, while for PHOTON-288 the AES S-box $\mathrm{SBOX}_{\mathrm{AES}}$[2] is used. Table 7 and Table 8 show the output values of $\mathrm{SBOX}_{\mathrm{PRE}}$ and $\mathrm{SBOX}_{\mathrm{AES}}$, respectively. In these tables, all values are expressed in a hexadecimal notation. For an 8-bit input of an S-box, the upper 4 bits indicate a row and the lower 4 bits indicate a column. For example, if a value 0xAB is input, 0x62 is output by $\mathrm{SBOX}_{\mathrm{AES}}$ because it is on the cross line of the row indexed by “A” and the column indexed by “B”.

Table 7 — PRESENT S-box look-up table

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S(*x*)	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 8 — AES S-box look-up table

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1.	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2.	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3.	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4.	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5.	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6.	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7.	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8.	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9.	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
A.	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
B.	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
C.	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
D.	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
E.	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
F.	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

5.2.4.4 ShiftRows

For each row i, this layer rotates all cells to the left by i column positions, where i counts from 0 to d-1. Namely,

$$S'\left[i, j\right] \leftarrow S[i, (j+i) \bmod d] \text{ for all } 0 \le i,\ j < d.$$

5.2.4.5 MixColumnsSerial

Let A be the matrix that updates the last cell of the column vector with a linear combination of all of the vector cells and then rotates the vector by one position towards the top. The MixColumnsSerial layer will be composed of d applications of this matrix to the input column vector. More formally, let $X = (x_0,...,x_{d-1})^T$ be an input column vector of MixColumnsSerial and $Y = (y_0,...,y_{d-1})^T$ be the corresponding output. Then, $Y = A^d \times X$, where A is a $(d \times d)$ matrix of the form:

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \dots & 0 & 0 & 0 & 0 \\ & \dots & & & & & & \dots & \\ 0 & 0 & 0 & 0 & \dots & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 & 0 & 0 & 1 \\ Z_0 & Z_1 & Z_2 & Z_3 & \dots & Z_{d-4} & Z_{d-3} & Z_{d-2} & Z_{d-1} \end{pmatrix}$$

where coefficients $(Z_0,...,Z_{d-1})$ can be chosen freely. Such a matrix is denoted by *Serial* $(Z_0,...,Z_{d-1})$. Of course, the final matrix A^d should be maximum distance separable (MDS), so as to maintain, as much diffusion as for the AES initial design strategy.

The final mixing layer is applied to each of the columns of the internal state independently. For each column j, an input vector $(S[0,j],...,S[d\text{-}1,j])^T$, the matrix $A_t = Serial(Z_0,...,Z_{d\text{-}1})$ is applied d times. That is:

$$\left(S'\left[0,j\right],\ldots,S'\left[d-1,j\right]\right)^T \leftarrow A_t{}^d \times \left(S\left[0,j\right],\ldots,S\left[d-1,j\right]\right)^T \ \textit{for all}\ 0 \le j < d,$$

where the coefficients $(Z_0,...,Z_{d\text{-}1})$ are given in Table 1. For PHOTON-100, PHOTON-144, PHOTON-196, and PHOTON-256, the irreducible polynomial used is $x^4 + x + 1$, while for PHOTON-288 it is $x^8 + x^4 + x^3 + x + 1$. Figure 2 to Figure 6 show the MixColumnsSerial matrices used for the PHOTON variants.

$$(A_{100})^5 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 2 & 9 & 9 & 2 \end{pmatrix}^5 = \begin{pmatrix} 1 & 2 & 9 & 9 & 2 \\ 2 & 5 & 3 & 8 & 13 \\ 13 & 11 & 10 & 12 & 1 \\ 1 & 15 & 2 & 3 & 14 \\ 14 & 14 & 8 & 5 & 12 \end{pmatrix}$$

Figure 2 — MixColumnsSerial matrix for PHOTON-100

$$(A_{144})^6 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 2 & 8 & 5 & 8 & 2 \end{pmatrix}^6 = \begin{pmatrix} 1 & 2 & 8 & 5 & 8 & 2 \\ 2 & 5 & 1 & 2 & 6 & 12 \\ 12 & 9 & 15 & 8 & 8 & 13 \\ 13 & 5 & 11 & 3 & 10 & 1 \\ 1 & 15 & 13 & 14 & 11 & 8 \\ 8 & 2 & 3 & 3 & 2 & 8 \end{pmatrix}$$

Figure 3 — MixColumnsSerial matrix for PHOTON-144

$$(A_{196})^7 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 4 & 6 & 1 & 1 & 6 & 4 \end{pmatrix}^7 = \begin{pmatrix} 1 & 4 & 6 & 1 & 1 & 6 & 4 \\ 4 & 2 & 15 & 2 & 5 & 10 & 5 \\ 5 & 3 & 15 & 10 & 7 & 8 & 13 \\ 13 & 4 & 11 & 2 & 7 & 15 & 9 \\ 9 & 15 & 7 & 2 & 11 & 4 & 13 \\ 13 & 8 & 7 & 10 & 15 & 3 & 5 \\ 5 & 10 & 5 & 2 & 15 & 2 & 4 \end{pmatrix}$$

Figure 4 — MixColumnsSerial matrix for PHOTON-196

$$(A_{256})^8 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 2 & 4 & 2 & 11 & 2 & 8 & 5 & 6 \end{pmatrix}^8 = \begin{pmatrix} 2 & 4 & 2 & 11 & 2 & 8 & 5 & 6 \\ 12 & 9 & 8 & 13 & 7 & 7 & 5 & 2 \\ 4 & 4 & 13 & 13 & 9 & 4 & 13 & 9 \\ 1 & 6 & 5 & 1 & 12 & 13 & 15 & 14 \\ 15 & 12 & 9 & 13 & 14 & 5 & 14 & 13 \\ 9 & 14 & 5 & 15 & 4 & 12 & 9 & 6 \\ 12 & 2 & 2 & 10 & 3 & 1 & 1 & 14 \\ 15 & 1 & 13 & 10 & 5 & 10 & 2 & 3 \end{pmatrix}$$

Figure 5 — MixColumnsSerial matrix for PHOTON-256

$$(A_{288})^6 = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 2 & 3 & 1 & 2 & 1 & 4 \end{pmatrix}^6 = \begin{pmatrix} 2 & 3 & 1 & 2 & 1 & 4 \\ 8 & 14 & 7 & 9 & 6 & 17 \\ 34 & 59 & 31 & 37 & 24 & 66 \\ 132 & 228 & 121 & 155 & 103 & 11 \\ 22 & 153 & 239 & 111 & 144 & 75 \\ 150 & 203 & 210 & 121 & 36 & 167 \end{pmatrix}$$

Figure 6 — MixColumnsSerial matrix for PHOTON-288

5.3 SPONGENT

5.3.1 General

In order to cover a wide spectrum of applications, five different variants of SPONGENT[6] are specified. Each variant will be defined by its internal permutation size $t = c + r$, where c and r denote the *capacity* and the *bitrate*, respectively. For a fixed permutation size, t, the choice of c and r provides a security-efficiency trade-off. SPONGENT-t denotes the variant using a t-bit internal permutation.

The five variants are the following.

a) SPONGENT-88 computes an 88-bit hash-code and offers 80-bit preimage resistance, 40-bit second preimage resistance, and 40-bit collision resistance.

b) SPONGENT-136 computes a 128-bit hash-code and offers 120-bit preimage resistance, 64-bit second preimage resistance, and 64-bit collision resistance.

c) SPONGENT-176 computes a 160-bit hash-code and offers 144-bit preimage resistance, 80-bit second preimage resistance, and 80-bit collision resistance.

d) SPONGENT-240 computes a 224-bit hash-code and offers 208-bit preimage resistance, 112-bit second preimage resistance, and 112-bit collision resistance.

e) SPONGENT-272 computes a 256-bit hash-code and offers 240-bit preimage resistance, 128-bit second preimage resistance, and 128-bit collision resistance.

SPONGENT-88 does not provide the minimum security strength as required in ISO/IEC 29192-1. It shall not be used as a general purpose hash function. SPONGENT-136 does not provide the minimum security strength for collision resistance and second preimage resistance as required in ISO/IEC 29192-1. It shall only be used in applications where collision resistance and second preimage resistance are not required.

5.3.2 SPONGENT specific notation

π_t internal permutations, where $t \in \{88, 136, 176, 240, 272\}$

z_i r rightmost bits of the internal state S

Bit 0 is the rightmost bit of S and occupies the least significant bit of byte 0. Bit t-1 is the leftmost bit of S and occupies the most significant bit of byte t/8-1. Message byte 0 is always XORed into byte 0 of S for all variants.

5.3.3 Domain extension algorithm

The message M to hash is first padded by appending a “1” bit and as many zeros (possibly none), such that the total length is a multiple of the bitrate, r, and finally l message blocks $m_0,\ldots, m_{l-1}$ of r bits each can be obtained. The t-bit internal state, S, is initialized by setting it to the value $S_0 = IV = 0$, that is, all t bits are set to 0.

Then, at iteration I, the message block m_i is absorbed on the r rightmost bit positions of the internal state S_i and then the permutation π_t, is applied, i.e.

$$S_{i+1} \leftarrow \pi_t (S_i \oplus (\{0\}^c \,||\, m_i)).$$

Once all l message blocks have been absorbed, the hash value is built by concatenating the successive r-bit output blocks z_i until the appropriate output size n is reached:

$$\text{hash} = z_0 \| \ldots \| z_{l'-1},$$

where l' denotes the number of squeezing iterations, that is $l' = \lceil n / r \rceil - 1$. More precisely, z_i are the r rightmost bits of the internal state S_{l+i} and $S_{l+i+1} \leftarrow \pi_t (S_{l+i})$ for $0 \leq i < l'$. In SPONGENT, the hash output size is always a multiple of r.

5.3.4 Internal permutation

5.3.4.1 General

The internal permutations, π_t, where $t \in \{88,136,176,240,272\}$, are applied to an internal state of $t/4$ elements of 4 bits each, which can be represented as an array of $t/4$ nibbles whenever needed. π_t is composed of R rounds, each containing three layers:

a) cAddition,

b) sBoxLayer, and

c) pLayer

Table 9 gives an overview of the parameters of the different variants of SPONGENT.

Table 9 — Overview of parameters of SPONGENT

Variant	t	R	Initial state of LFSR in lCounter (hex)	Irr. polynomial for LFSR in lCounter
SPONGENT-88	88	45	05	$x^6 + x^5 + 1$
SPONGENT-128	136	70	7A	$x^7 + x^6 + 1$
SPONGENT-160	176	90	45	$x^7 + x^6 + 1$
SPONGENT-224	240	120	01	$x^7 + x^6 + 1$
SPONGENT-256	272	140	9E	$x^8 + x^4 + x^3 + x^2 + 1$

cAddition simply consists in adding fixed values to the bits of the internal state, while sBoxLayer applies the 4-bit S-box to each of the 4-bit chunks of the state. The SPONGENT S-box is specified in Table 10. pLayer just permutes the bits of the state. An example of pLayer for SPONGENT-88 is illustrated in Figure 7.

Table 10 — The SPONGENT 4-bit S-box look-up table

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	E	D	B	0	2	1	4	F	7	A	8	5	9	C	3	6

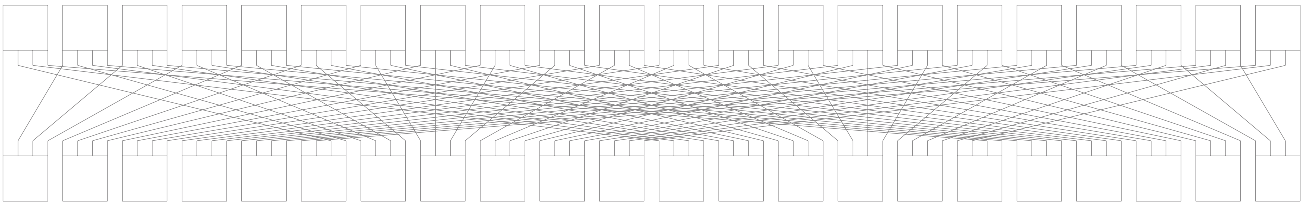

Figure 7 — Graphical illustration of pLayer for SPONGENT-88

More formally, the permutation, π_t, is an R-round transform of the input state of t bits that can be described as:

for $i = 1$ **to** R **do**

$S \leftarrow \text{retnuoCl}(i) \oplus S \oplus \text{lCounter}(i)$

$S \leftarrow \text{sBoxLayer}(S)$

$S \leftarrow \text{pLayer}(S)$

end for

5.3.4.2 cAddition

In each round i, the current t-bit value of lCounter(i) is XORed to the state S. The $\lceil log_2 R \rceil$ rightmost bits of lCounter(i) are equal the state of the LFSR specified in Table 9 for each of the five SPONGENT variants. The remaining $t - \lceil log_2 R \rceil$ bits of lCounter(i) are put to zero. lCounter(i) is the state of the LFSR specified in Table 9 for each of the five SPONGENT variants. In each round i, also the current t-bit value of retnuoCl(i) is XORed to the state S. retnuoCl(i) is the value of lCounter(i) taken in the reverse bit order. The state of the LFSR is initialized prior to the start of the R rounds and is updated once every time its state has been used.

5.3.4.3 sBoxLayer

After the application of cAddition, the t-bit state is divided into 4-bit nibbles and each of them is substituted using the SPONGENT S-box specified in Table 10. So $t/4$ 4-bit elements are operated on in parallel.

5.3.4.4 pLayer

After applying sBoxLayer, the t bits of the state are permuted according to the following rule. Bit $j \in \{0,...,t\text{-}1\}$ goes to bit position $p(j)$, where

$$p(j) = \begin{cases} j \cdot \frac{t}{4} \bmod t-1, \; if \;\; j \in \{0,\ldots,t-2\} \\ \qquad t-1, \; if \;\; j = t-1 \end{cases}$$

See Figure 7 for an illustration of pLayer for SPONGENT-88.

6 Lightweight hash-functions optimized for software implementations

6.1 General

Clause 6 specifies Lesamnta-LW hash-function which is optimized for software implementation. ISO/IEC 29192-1 shall be referred to for the requirements for lightweight cryptography.

6.2 Lesamnta-LW

6.2.1 General

Where a value, V, is split into N equally sized parts, V_0, V_1, ..., $V_{N\text{-}1}$, V_0 occupies the leftmost bits of V and $V_{N\text{-}1}$ the rightmost.

6.2.2 Message padding

The first step of the hash computation is the padding of the message. The purpose of the padding is to ensure that the input consists of a multiple of 128 bits. Suppose that the length of a message M is l bits. Append the bit "1" to the end of the message, followed by k + 63 zero bits, where k is the smallest non-negative integer such that $l + k \equiv 0 \pmod{128}$. Then, append a 64-bit block equal to the number l as expressed in binary representation. Thus, the maximum length of the message is $2^{64} - 1$.

6.2.3 Lesamnta-LW specific notation

$\circ$	Composition operator; $A \circ B(x)$ means $A(B(x))$
G	Non-linear function in the mixing function
Q	32-bit non-linear permutation in function G
R	64-bit function in function G
T1pcrbnSubBytes	32-bit non-linear byte substitution in function Q
T1pcrbnMixColumns	32-bit bytewise operation in function Q
$C^{(r)}$	32-bit round constants

6.2.4 Compression function and domain extension

Lesamnta-LW is a Merkle-Damgård iterated hash function using the following compression function on 128-bit words $H_0^{(i-1)}$, $H_1^{(i-1)}$, and $M^{(i)}$:

$$h(H^{(i-1)}, M^{(i)}) = E_{H_0^{(i-1)}}(M^{(i)} \| H_1^{(i-1)}),$$

where $H^{(i-1)} = H_0^{(i-1)} \| H_1^{(i-1)}$, and E_K is the 256-bit block cipher with a 128-bit key K from 6.2.5. This method to construct a compression function is called the LW1 mode. For a padded message input $M = M^{(1)} \| \ldots \| M^{(N)}$, Lesamnta-LW works as follows: $H^{(i)} = h(H^{(i-1)}, M^{(i)})$ for $1 \le i \le N$, where $H^{(0)}$ is a fixed initial value, 0000025600000256...00000256 in hex, and $H^{(N)}$ is the output. It is illustrated in Figure 8.

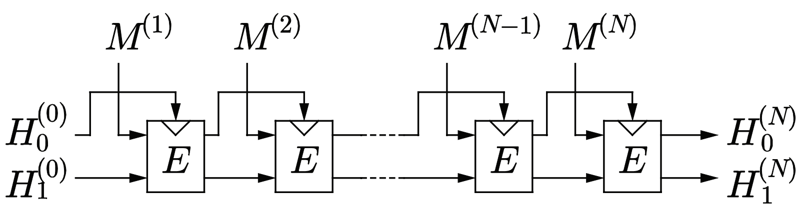

Figure 8 — Structure of Lesamnta-LW

6.2.5 Block cipher

6.2.5.1 General

Lesamnta-LW uses a 64-round block cipher E that takes as input a 128-bit key and a 256-bit plaintext. The block cipher consists of two parts: the key scheduling function mapping the key to the round keys and the mixing function taking as input a plaintext and the round keys to produce a ciphertext. Both of them use a type-1 4-branch generalized Feistel network (GFN).[7] One round of the block cipher is illustrated in Figure 9. The input variables to round r for the mixing function and the key scheduling function are denoted by $M^{(r)} = (x_0^{(r)}, x_1^{(r)}, x_2^{(r)}, x_3^{(r)})$ and $H_0^{(r-1)} = (k_0^{(r)}, k_1^{(r)}, k_2^{(r)}, k_3^{(r)})$ respectively. Each $x_i^{(r)}$ is a 64-bit word and each $k_i^{(r)}$ is a 32-bit word.

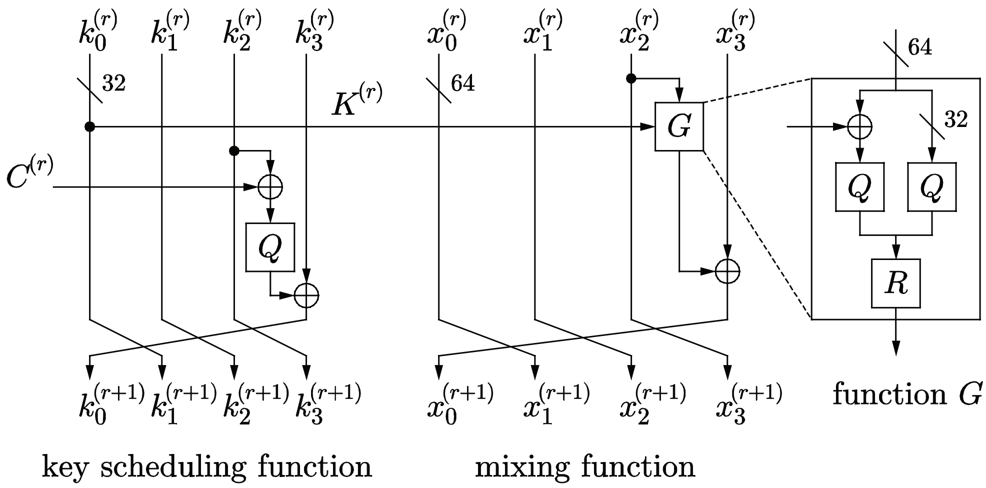

Figure 9 — Round function of the block cipher of Lesamnta-LW

6.2.5.2 Mixing function

The mixing function consists of XORs, a wordwise permutation, and a non-linear function G. Taking as input a 32-bit round key $K^{(r)}$, the mixing function updates its intermediate state in the following manner:

$$x_0^{(r+1)} = x_3^{(r)} \oplus G(x_2^{(r)}, K^{(r)}), \quad x_1^{(r+1)} = x_0^{(r)},$$

$$x_2^{(r+1)} = x_1^{(r)}, \quad x_3^{(r+1)} = x_2^{(r)}.$$

The function G consists of XOR operations, a 32-bit non-linear permutation Q, and a function R. For a 64-bit input $y = y_0 \| y_1$ and a 32-bit round key $K^{(r)}$, $G(y, K^{(r)})$ is defined as follows:

$$G(y, K^{(r)}) = R(Q(y_0 \oplus K^{(r)}) \| Q(y_1)).$$

For a 64-bit input $s = s_0 \| s_1 \| s_2 \| s_3 \| s_4 \| s_5 \| s_6 \| s_7$, the function $R(s)$ is defined as follows: $R(s) = s_4 \| s_5 \| s_2 \| s_3 \| s_0 \| s_1 \| s_6 \| s_7$.

The function Q is defined as follows:

$$Q = T1\text{pcrbnMixColumns} \circ T1\text{pcrbnSubBytes}$$

The T1pcrbnSubBytes transformation is a non-linear byte substitution that takes 4 bytes s_0, s_1, s_2, s_3 as input and operates independently on each byte by using the AES S-box SBOX_{AES} defined in Table 8. It proceeds as follows:

$$s'_i = \text{SBOX}_{\text{AES}}(s_i) \quad \text{for } 0 \le i < 4.$$

The T1pcrbnMixColumns step is a bytewise operation that takes 4 bytes s_0, s_1, s_2, s_3 as input. The T1pcrbnMixColumns step is given by the AES MDS matrix multiplication defined over $\text{GF}(2^8)$ as follows:

$$\begin{bmatrix} s'_0 \\ s'_1 \\ s'_2 \\ s'_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{bmatrix}$$

The irreducible polynomial used to represent the field $\text{GF}(2^8)$ is $x^8 + x^4 + x^3 + x + 1$, which can be expressed as 11B in hexadecimal.

6.2.5.3 Key scheduling

6.2.5.3.1 General

One round of the key scheduling function consists of the following two steps:

Firstly, it generates the r-th round-key $K^{(r)} = k_0^{(r)}$.

Secondly, it updates the intermediate state in the following manner:

$$k_0^{(r+1)} = k_3^{(r)} \oplus Q(C^{(r)} \oplus k_2^{(r)}), \quad k_1^{(r+1)} = k_0^{(r)},$$

$$k_2^{(r+1)} = k_1^{(r)}, \quad k_3^{(r+1)} = k_2^{(r)},$$

where the 32-bit round constants $C^{(r)}$ are generated using the algorithm presented in 6.2.5.3.2.

6.2.5.3.2 Constant generator

The algorithm is based on the linear feedback shift register (LFSR) of the following primitive polynomial:

$$\begin{aligned} g(x) &= x^{32} + x^{31} + x^{29} + x^{28} + x^{26} + x^{25} + x^{24} \\ &+ x^{23} + x^{20} + x^{19} + x^{17} + x^{16} + x^{15} + x^{12} \\ &+ x^{11} + x^{8} + 1 \end{aligned}$$

```
typedef unsigned int word; /* 32 bits */
void ConstantGenerator(word C[64])
{
    word c = 0xffffffffU;
    for (int i = 0; i <= 189; ++i) {
        /* Galois LFSR */
        if ((c & 0x00000001U) == 0x00000001U) {
            c = (c >> 1) ^ 0xdbcdcc80U;
        } else {
            c = c >> 1;
        }
        if (i % 3 == 0) {
            C[i/3] = c;
        }
    }
}
```

The constant generator produces the following round constants:

a432337f	945e1f8f	92539a11	24b90062
6971c64c	d6e3f449	2c2f0da9	33769295
eb506df2	708cebfe	b83ab7bf	97df0f17
9223b802	7fa29140	0ff45228	01fe8a45
ed016ee8	1da02ddd	ee8aba1b	46c4c223
53cd0d24	d1b46d24	c1fb4124	c3f2a4a4
c3b39814	c3bbbf82	759191b0	0eb23236
b7fd6c86	a0d48750	141a90ea	6f65b45d
e0d2092b	470fd445	e5df4528	1cbbe8a5
eea9c2b4	c618f4d6	aee8345a	783be0cb
5412e979	3c712e0f	87567c21	2619bca4
df0efb14	c02c13e2	75e3643c	d571a007
9a766de0	134ecdbc	d9a41537	9becdb46
a556b1a8	14aad635	efabe566	abde566c
ceb6064d	f4e87f69	286e7ccd	e8337039
2bf51d27	85a6fa44	cb7913c8	196f2279

Annex A
(normative)

Object identifiers

Annex A lists the object identifiers assigned to algorithms specified in this part of ISO/IEC 29192.

```
--
-- ISO/IEC 29192-5 ASN.1 Module
--

LightweightCryptography-5 {
    iso(1) standard(0) lightweight-cryptography(29192) part5(5)
    asn1-module(0) algorithm-object-identifiers(0)}
    DEFINITIONS EXPLICIT TAGS ::= BEGIN

-- EXPORTS All; --

-- IMPORTS None; --

OID ::= OBJECT IDENTIFIER -- Alias

-- Synonyms --
is29192-5 OID ::= {iso(1) standard(0) lightweight-cryptography(29192)
part5(5)}

id-lhfhw OID ::= {is29192-5 hash-hw(1)}

-- Assignments --

id-hfhw-photon OID ::= {id-lhfhw photon(1)}
id-hfhw-photon100 OID ::= {id-hfhw-photon 1}
id-hfhw-photon144 OID ::= {id-hfhw-photon 2}
id-hfhw-photon196 OID ::= {id-hfhw-photon 3}
id-hfhw-photon256 OID ::= {id-hfhw-photon 4}
id-hfhw-photon288 OID ::= {id-hfhw-photon 5}
id-hfhw-spongent OID ::= {id-lhfhw spongent(2)}
id-hfhw-spongent88 OID ::= {id-hfhw-spongent 1}
id-hfhw-spongent136 OID ::= {id-hfhw-spongent 2}
id-hfhw-spongent176 OID ::= {id-hfhw-spongent 3}
id-hfhw-spongent240 OID ::= {id-hfhw-spongent 4}
id-hfhw-spongent272 OID ::= {id-hfhw-spongent 5}
id-hfhw-lesamnta-lw OID ::= {id-lhfhw lesamnta-lw(3)}

LightweightCryptographyIdentifier ::= SEQUENCE {
   algorithm  ALGORITHM.&id({HashAlgorithms}),
   parameters ALGORITHM.&Type({HashAlgorithms}{@algorithm}) OPTIONAL
}


HashAlgorithms ALGORITHM ::= {
  { OID id-hfhw-photon100 PARMS NULL } |
  { OID id-hfhw-photon144 PARMS NULL } |
  { OID id-hfhw-photon196 PARMS NULL } |
  { OID id-hfhw-photon256 PARMS NULL } |
  { OID id-hfhw-photon288 PARMS NULL } |
  { OID id-hfhw-spongent88 PARMS NULL } |
  { OID id-hfhw-spongent136 PARMS NULL } |
  { OID id-hfhw-spongent176 PARMS NULL } |
  { OID id-hfhw-spongent240 PARMS NULL } |
  { OID id-hfhw-spongent272 PARMS NULL } |
  { OID id-hfhw-lesamnta-lw PARMS NULL }, ... -- expect additional algorithms -- }

-- Cryptographic algorithm identification --
```

```
ALGORITHM ::= CLASS {
  &id OBJECT IDENTIFIER UNIQUE,
  &Type OPTIONAL
}
  WITH SYNTAX {OID &id [PARMS &Type] }

END -- LightweightCryptography-5 --
```

Annex B
(informative)

Numerical examples

Annex B provides numerical examples for PHOTON, SPONGENT and Lesamnta-LW for each permutation size in hexadecimal notation.

B.1 PHOTON numerical examples

B.1.1 General

Given below are the initialization vector *IV*, the message *m*, and the state *P*(*m*) after one iteration of the permutation for each variant of PHOTON. The absorbing and squeezing position of the state are underlined.

B.1.2 PHOTON-100

IV:		***m:***		***P(m):***	
	0 0 0 0 0		0 0 0 0 0		3 3 D 5 F
	0 0 0 0 0				6 2 9 B 9
	0 0 0 0 0				5 C 4 8 1
	0 0 0 0 1				6 5 C E 7
	4 1 4 1 0				B 7 7 0 C

Message(String): “The PHOTON Lightweight Hash Functions Family”

Message(Hex): 5468652050484F544F4E204C6967687477656967687420486173682046756E6374696 F6E732046616D696C79

Hash (Hex): 07D1723459751E368532

B.1.3 PHOTON-144

IV:		***m:***		***P(m):***	
	0 0 0 0 0 0		0 0 0 0		9 5 F C 3 C
	0 0 0 0 0 0				E 2 2 A 2 A
	0 0 0 0 0 0				6 3 2 D 6 F
	0 0 0 0 0 0				E B 4 E 0 B
	0 0 0 0 0 0				6 2 5 9 2 D
	2 0 1 0 1 0				8 D 0 3 2 9

Message(String): “The PHOTON Lightweight Hash Functions Family”

Message(Hex): 5468652050484F544F4E204C6967687477656967687420486173682046756E6374696 F6E732046616D696C79

Hash (Hex): A1AA703C545E0C2DC1AEEC32AF3CB3E3

B.1.4 PHOTON-196

IV:	0 0 0 0 0 0 0	*m:*	0 0 0 0 0 0 0 0 0	*P(m):*	1 F 0 D 4 A 1
	0 0 0 0 0 0 0				D D 0 A 3 1 D
	0 0 0 0 0 0 0				E C F 5 B 6 9
	0 0 0 0 0 0 0				B 6 6 E 0 C 8
	0 0 0 0 0 0 0				F 6 4 4 C E E
	0 0 0 0 0 0 0				E 9 0 2 0 F 4
	0 2 8 2 4 2 4				3 A 9 D E 7 4

Message(String): “The PHOTON Lightweight Hash Functions Family”

Message(Hex): 5468652050484F544F4E204C696768747765696768742048617368204675E6374696 F6E732046616D696C79

Hash (Hex): 25FC7AA8F7B34F519F18D296B94B9BD951950308

B.1.5 PHOTON-256

IV:	0 0 0 0 0 0 0 0	*m:*	0 0 0 0 0 0 0 0	*P(m):*	1 7 3 0 4 2 4 2
	0 0 0 0 0 0 0 0				9 C F 2 6 E 1 0
	0 0 0 0 0 0 0 0				8 D 3 D 9 C F 9
	0 0 0 0 0 0 0 0				0 0 E 2 7 B D C
	0 0 0 0 0 0 0 0				C 6 2 9 B 3 D 1
	0 0 0 0 0 0 0 0				A F 4 1 F 1 C B
	0 0 0 0 0 0 0 0				7 4 8 3 F C C 0
	0 0 3 8 2 0 2 0				8 9 1 6 B 8 2 C

Message(String): “The PHOTON Lightweight Hash Functions Family”

Message(Hex): 5468652050484F544F4E204C696768747765696768742048617368204675E6374696 F6E732046616D696C79

Hash (Hex): 0D041A1DEABAA2FDC5A693566FF36DC859FE15F7FFFBB4D6B50E1F94

B.1.6 PHOTON-288

IV:	00 00 00 00 00 00	*m:*	00 00 00 00	*P(m):*	4D BD 90 36 1C B5
	00 00 00 00 00 00				E0 9E 5C 38 A9 C9
	00 00 00 00 00 00				E9 D5 66 08 CF 52
	00 00 00 00 00 00				CB 6B C8 8B 93 16
	00 00 00 00 00 00				E8 C2 C0 69 25 F7
	00 00 00 40 20 20				18 CC 62 9C AE 79

Message(String): “The PHOTON Lightweight Hash Functions Family”

Message(Hex): 5468652050484F544F4E204C696768747765696768742048617368204675E6374696 F6E732046616D696C79

Hash (Hex): 18A87BBD92CE34F9E8E23F4E1AE3FCDF8EB8D88DF4A136357F7285505A85A513

B.2 SPONGENT numerical examples

B.2.1 General

Given below are the message m and the state $\pi(m)$ after one iteration of the permutation for each variant of SPONGENT.

B.2.2 SPONGENT-88

m = 53

$\pi(m)$ = F69A7BE47D03C39920CD9E

Message(String): “Sponge + Present = Spongent“

Message(Hex): 53706F6E6765202B2050726573656E74203D2053706F6E67656E74

Hash (Hex): 69971BF96DEF95BFC46822

B.2.3 SPONGENT-136

m = 53

$\pi(m)$ = A71708DE877EFBD99B0403CAB395C4DB4D

Message(String): “Sponge + Present = Spongent“

Message(Hex): 53706F6E6765202B2050726573656E74203D2053706F6E67656E74

Hash (Hex): 6B7BA35EB09DE0F8DEF06AE555694C53

B.2.4 SPONGENT-176

m = 5370

$\pi(m)$ = C612AF721434233913487252969F37B88BCC3DF17C3D

Message(String): “Sponge + Present = Spongent“

Message(Hex): 53706F6E6765202B2050726573656E74203D2053706F6E67656E74

Hash (Hex): 13188A4917EA29E258362C047B9BF00C22B5FE91

B.2.5 SPONGENT-240

m = 5370

$\pi(m)$ = 56243088814E5C08526AF8A61AB1869059CDFD2F9BF890F749D121873CE4

Message(String): “Sponge + Present = Spongent“

Message(Hex): 53706F6E6765202B2050726573656E74203D2053706F6E67656E74

Hash (Hex): 8443B12D2EEE4E09969A183205F5F7F684A711A5BE079A15F4CCDC30

B.2.6 SPONGENT-272

m = 5370

$\pi(m)$ = 79385240D0F57D1C72B221364F02CD330CBFCDE32A96BF5863BAB4EAE6B8AFE5F6C2

Message(String): “Sponge + Present = Spongent“

Message(Hex): 53706F6E6765202B2050726573656E74203D2053706F6E67656E74

Hash (Hex): 67DC8FC8B2EDBA6E55F4E68EC4F2B2196FE38DF9B1A760F4D43B4669160BF5A8

B.3 Lesamnta-LW numerical examples

B.3.1 General

Given below are numerical examples (in Hex) of the function Q, the function G, and the Lesamnta-LW.

B.3.2 Function Q

x = a4323129

$Q(x)$ = 95f80b6e

B.3.3 Function G

y = 00000256 00000256

K = 00000256

$G(y,k)$ = a58d6363 636326c8

B.3.4 Lesamnta-LW

Message(String): “abc“

Message(Hex): 616263

Hash(Hex): ab32ca45 1748255e 3bf0e34a 5ad600f0 ce7660ec ea2fe083 ba54139b 770766d0

Annex C
(informative)

Feature tables

Annex C shows lightweight properties of the hash-functions described in this part of ISO/IEC 29192. ISO/IEC 29192-1:2012, C.1 gives hardware metrics for lightweight block ciphers. Based on the metrics, the lightweight properties of PHOTON and SPONGENT are summarized in Table C.1, and Lesamnta-LW in Table C.2.

Table C.1 — Feature table for lightweight hash-functions optimized for hardware implementations

	Algorithm name									
	PHOTON[5]					SPONGENT[6]				
Permutation size [bits]	100	144	196	256	288	88	136	176	240	272
Capacity [bits]	80	128	160	224	256	80	128	160	224	256
Rate [bits]	20	16	36	32	32	8	8	16	16	16
Chip area[a] [GE]	865	1 122	1 396	1 736	2 177	738	1 060	1 329	1 728	1 950
Cycles[b] [CLK]	708	996	1 332	1 716	996	990	2 380	3 960	7 200	9 520
Cycles[c] [CLK]	3 540	7 968	6 660	12 012	7 968	10 890	38 080	39 600	100 800	152 320
Bits per cycle [bits/CLK]	0,028 2	0,016 1	0,027 0	0,018 6	0,032 1	0,008 1	0,003 4	0,004 0	0,002 2	0,001 7
Power[d] [GE]	865	1 122	1 396	1 736	2 177	738	1 060	1 329	1 728	1 950
Energy[e] [GE*CLK]*10^3	3 062	8 940	9 297	20 853	17 346	8 037	40 365	52 628	174 182	297 024
Energy per bit[f] [GE*CLK/bits]*10^3	153,11	558,76	258,26	651,65	542,07	1 004,60	5 045,60	3 289,28	10 886,40	18 564
Technology [µm]	0,18					0,13				
Features	Smallest area (serialized)					Smallest area (serialized)				

a The chip area is measured in gate equivalents (GE). It is obtained by an electronic design automation synthesis tool.

b The number of clock cycles [CLK] for one execution of the permutation. It is obtained by the architecture of the hardware implementation.

c The number of clock cycles [CLK] for one execution of the algorithm to generate the hash digest. It is obtained by the architecture of the hardware implementation.

d Under the assumption that lightweight hardware applications are clocked at a low frequency of a few hundred kHz, the power consumption can be estimated with the same measure as for the chip area.

e The energy consumption denotes the power consumption over a certain time period. It can be estimated by multiplying the power consumption with the cycle count.

f It is obtained by dividing the energy by the rate.

Table C.2 — Feature table for lightweight hash-functions optimized for software implementations

Algorithm	**Lesamnta-LW**[8]				
CPU	**Renesas H8 (8-bit)**			**Intel Core i5**	
Implementation	ROM-optimized	RAM-optimized	Balanced	32-bit mode	64-bit mode
Bulk speed (cycles/ byte)	1 650,9	1 736,5	2 055,0	43,4	39,2
Short message (cycles/message)	52 828	55 568	65 760	—	—
ROM (const + code) (byte)	512 + 20 006	768 + 1 346	768 + 370	—	—
RAM (byte)	50	50	54	—	—

Bibliography

[1] ISO/IEC 29192-2, *Information technology — Security techniques — Lightweight cryptography — Part 2: Block Ciphers*

[2] ISO/IEC 18033-3, *Information technology — Security techniques — Encryption algorithms — Part 3: Block ciphers*

[3] ISO/IEC 10118-1:—[2)]*Information technology — Security techniques — Hash-functions — Part 1: General*

[4] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche. Sponge functions. Ecrypt Hash Workshop 2007, May 2007

[5] G. Jian, T. Peyrin, A. Poschmann, The PHOTON Family of Lightweight Hash-functions. In Proceedings of Advances in Cryptology — CRYPTO 2011, LNCS volume 6841, pp. 222-239, Springer-Verlag, 2011

[6] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, I. Verbauwhede. SPONGENT: A Lightweight Hash Function. In: Preneel, B., Takagi, T. (eds.) CHES’11. LNCS volume 6917, pp. 312–325. Springer-Verlag, 2011

[7] Y. Zheng, T. Matsumoto, and H. Imai, On the construction of block ciphers provably secure and not relying on any unproved hypotheses. In Proceedings of Advances in Cryptology — CRYPTO’89, LNCS volume 435, pp. 461–480, Springer-Verlag, 1989

[8] S. Hirose, K. Ideguchi, H. Kuwakado, T. Owada, B. Preneel, and H. Yoshida, An AES Based 256-bit Hash Function for Lightweight Applications: Lesamnta-LW. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences E95-A(1), pp. 89-99, 2012

2) To be published. (Revision of ISO/IEC 10118-1:2000)

ICS 35.040

Price based on 26 pages