INTERNATIONAL STANDARD

ISO/IEC 29192-3

First edition
2012-10-01

Information technology — Security techniques — Lightweight cryptography —

Part 3: Stream ciphers

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

Partie 3: Chiffrements à flot

Reference number
ISO/IEC 29192-3:2012(E)

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

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.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

ISO/IEC 29192-3 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee 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*

Introduction

This part of ISO/IEC 29192 specifies keystream generators for lightweight stream ciphers tailored for implementation in constrained environments. ISO/IEC 29192-1 specifies the requirements for lightweight cryptography. A stream cipher is an encryption mechanism that uses a keystream generator to generate a keystream to encrypt a plaintext in bitwise or block-wise manner.

The International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC) draw attention to the fact that it is claimed that compliance with this document may involve the use of patents.

ISO and IEC take no position concerning the evidence, validity and scope of these patent rights.

The holders of these patent rights have assured 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 statement of the holders of these patent rights are registered with ISO and IEC. Information may be obtained from:

René-MichaelCordes/ErnstSchobesberger/M&C ConsultInvest&TradeGmbH
LogoDynamic Unit GmbH
Prinz Eugen Strasse 52/9,
A-1040 Vienna
Austria

Hitachi Ltd.
IP Licensing Department
Intellectual Property Group
Marunouchi Center Building
6-1, Marunouchi 1-chome,
Chiyoda-ku,
Tokyo, 100-8220
Japan

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 3: Stream ciphers

1 Scope

This part of ISO/IEC 29192 specifies two dedicated keystream generators for lightweight stream ciphers:

- Enocoro: a lightweight keystream generator with a key size of 80 or 128 bits;
- Trivium: a lightweight keystream generator with a key size of 80 bits.

2 Normative reference

The following referenced documents are indispensable for the application of this document. 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 terms and definitions given in ISO/IEC 29192-1 and the following apply.

3.1
big-endian
method of storage of multi-byte numbers with the most significant bytes at the lowest memory addresses

[ISO/IEC 18033-4:2011]

3.2
ciphertext
data which has been transformed to hide its information content

[ISO/IEC 18033-1:2005]

3.3
decryption
reversal of a corresponding encipherment

[ISO/IEC 18033-1:2005]

3.4
encryption
(reversible) transformation of data by a cryptographic algorithm to produce ciphertext, i.e. to hide the information content of the data

[ISO/IEC 18033-1:2005]

3.5
initialization value
value used in defining the starting point of an encryption process

[ISO/IEC 18033-4:2011]

3.6
key
sequence of symbols that controls the operation of a cryptographic transformation (e.g. encipherment, decipherment)

[ISO/IEC 18033-1:2005]

3.7
keystream function
function that takes as input, the current state of the keystream generator and (optionally) part of the previously generated ciphertext, and gives as output the next part of the keystream

[ISO/IEC 18033-4:2011]

3.8
keystream generator
state-based process (i.e., a finite state machine) that takes as input, a key, an initialization vector, and if necessary the ciphertext, and gives as output a keystream (i.e., a sequence of bits or blocks of bits) of arbitrary length

[ISO/IEC 18033-4:2011]

3.9
next-state function
function that takes as input, the current state of the keystream generator and (optionally) part of the previously generated ciphertext, and gives as output a new state for the keystream generator

[ISO/IEC 18033-4:2011]

3.10
plaintext
unenciphered information

[ISO/IEC 18033-1:2005]

3.11
secret key
key used with symmetric cryptographic techniques by a specified set of entities

[ISO/IEC 18033-1:2005]

3.12
state
internal state of a keystream generator

4 Symbols and operational terms

0x	Prefix for hexadecimal values.
$0^{(n)}$	n-bit variable where 0 is assigned to every bit.
AND	Bitwise logical AND operation.
a_i	Variable forming part of the internal state of a keystream generator.
b_i	Variable forming part of the internal state of a keystream generator.
C_i	Ciphertext block.
$F[x]$	The polynomial ring over the finite field F.
$GF(2^n)$	Finite field of 2^n elements.
Init	Function which generates the initial internal state of a keystream generator.
IV	Initialization vector.
K	Key.
Next	Next-state function of a keystream generator.
n	Block length.
OR	Bitwise logical OR operation.
Out	Output function combining keystream and plaintext in order to generate ciphertext.
P	Plaintext.
P_i	Plaintext block.
Strm	Keystream function of a keystream generator.
S_i	Internal state of a keystream generator.
Z	Keystream.
Z_i	Keystream block.
$\lceil x \rceil$	The smallest integer greater than or equal to the real number x.
$\neg x$	Bitwise complement operation.
$\bullet$	Polynomial multiplication.
$\|\|$	Bit concatenation.
$+_m$	Integer addition modulo 2^m.
$\oplus$	Bitwise XOR (eXclusive OR) operation.
$<<_n t$	t-bit left shift in an n-bit register.

$>>_n t$ — t-bit right shift in an n-bit register.

$<<<_n t$ — t-bit left circular rotation in an n-bit register.

$>>>_n t$ — t-bit right circular rotation in an n-bit register.

$\otimes$ — Multiplication operation for elements in the finite field GF(2^n).

NOTE An example of operation of multiplication of elements in the finite field GF(2^n) is given in Annex E.

5 General models for stream ciphers

5.1 General

This clause describes general models for stream ciphers [ISO/IEC 18033-4:2011].

5.2 Synchronous Keystream generators

A synchronous keystream generator is a finite-state machine. It is defined by:

1. An initialization function, *Init*, which takes as input a key K and an initialization vector IV, and outputs an initial state S_0 for the keystream generator. The initialization vector should be chosen so that no two messages are ever encrypted using the same key and the same IV.

2. A next-state function, *Next*, which takes as input the current state of the keystream generator S_i, and outputs the next state of the keystream generator S_{i+1}.

3. A keystream function, *Strm*, which takes as input a state of the keystream generator S_i, and outputs a keystream block Z_i.

When the synchronous keystream generator is first initialized, it will enter an initial state S_0 defined by

$$S_0 = \mathit{Init}(IV, K).$$

On demand the synchronous keystream generator will for i=0,1,...:

1. Output a keystream block $Z_i = \mathit{Strm}(S_i, K)$.

2. Update the state of the machine $S_{i+1} = \mathit{Next}(S_i, K)$.

Therefore to define a synchronous keystream generator it is only necessary to specify the functions *Init, Next* and *Strm*, including the lengths and alphabets of the key, the initialization vector, the state, and the output block.

5.3 Output functions

5.3.1 General model of output function

This subclause specifies a stream cipher output function, i.e. a technique to be used in a stream cipher to combine a keystream with plaintext to derive ciphertext.

An output function for a synchronous or a self-synchronizing stream cipher is an invertible function *Out* that combines a plaintext block P_i, a keystream block Z_i to give a ciphertext block C_i ($i \geq 0$). A general model for a stream cipher output function is now defined.

Encryption of a plaintext block P_i by a keystream block Z_i is given by:

$C_i = Out(P_i, Z_i)$,

and decryption of a ciphertext block C_i by a keystream block Z_i is given by:

$P_i = Out^{-1}(C_i, Z_i)$.

The output function shall be such that, for any keystream block Z_i, and plaintext block P_i, we have

$P_i = Out^{-1}(Out(P_i, Z_i), Z_i)$.

5.3.2 Binary-additive output function

A binary-additive stream cipher is a stream cipher in which the keystream, plaintext, and ciphertext blocks are binary digits, and the operation to combine plaintext with keystream is bitwise XOR. Let n be the bit length of P_i. This function is specified by

$Out(P_i, Z_i) = P_i \oplus Z_i$.

The operation Out^{-1} is specified by

$Out^{-1}(C_i, Z_i) = C_i \oplus Z_i$.

6 Dedicated keystream generators

6.1 *Enocoro*-128v2 keystream generator

6.1.1 Introduction to *Enocoro*-128v2

Enocoro-128v2 is a keystream generator which uses a 128-bit secret key K, a 64-bit initialization vector IV, and a state variable S_i ($i \geq 0$) consisting of 34 bytes, and outputs a keystream block Z_i of one byte at every iteration of the function *Strm*.

NOTE This keystream generator was originally proposed in [5].

The state variable S_i is sub-divided into a 2-byte variable:

$a^{(i)} = (a_0^{(i)}, a_1^{(i)})$,

where $a_j^{(i)}$ is a byte (for $j = 0, 1$), and a 32-byte variable:

$b^{(i)} = (b_0^{(i)}, b_1^{(i)}, \ldots, b_{31}^{(i)})$,

where $b_j^{(i)}$ is a byte (for $j = 0, 1, \ldots, 31$).

The *Init* function, defined in detail in 6.1.2, takes as input the 128-bit key K and the 64-bit initializing vector IV, and produces the initial value of the state variable $S_0 = (a^{(0)}, b^{(0)})$.

The *Next* function, defined in detail in 6.1.3, takes as input the 34-byte state variable $S_i = (a^{(i)}, b^{(i)})$ and produces as output the next value of the state variable $S_{i+1} = (a^{(i+1)}, b^{(i+1)})$.

The *Strm* function, defined in detail in 6.1.4, takes as input the 34-byte state variable $S_i = (a^{(i)}, b^{(i)})$ and produces as output the keystream block Z_i.

Enocoro-128v2 uses operations over the finite field $GF(2^8)$. In the polynomial representation, $GF(2^8)$ is realized as $GF(2)[x] / \phi_{8432}(x)$, where $\phi_{8432}(x)$ is an irreducible polynomial of degree 8 defined over $GF(2)$. The *Enocoro*-128v2 keystream generator uses the following irreducible polynomial:

$\phi_{8432}(x) = x^8 + x^4 + x^3 + x^2 + 1.$

6.1.2 Initialization function *Init*

The initialization of *Enocoro*-128v2 is divided into six steps. During the initialization of *Enocoro*-128v2, the state is updated as sketched in Figure 1.

The initialization function *Init* is as follows:

Input: 128-bit key K, 64-bit initialization vector IV.

Output: Initial value of the state variable $S_0 = (a^{(0)}, b^{(0)})$.

a) Use the key K to set part of the state variable $b_j^{(-96)}$ as follows:

— Set $(K_0||K_1||...||K_{15}) = K$, where K_j is 8 bits for j=0,1,2,...,15.

— For j=0,1,2,...,15, set $b_j^{(-96)} = K_j$.

b) Use the initialization vector IV to set part of the state variable $b_j^{(-96)}$ as follows:

— Set $(I_0||I_1||...||I_7) = IV$, where I_j is 8 bits for j=0,1,2,...,7.

— For j=0,1,2,...,7, set $b_{j+16}^{(-96)} = I_j$.

c) Use the constants $C_0, C_1, ..., C_9$ to set part of the state variable $a_j^{(-96)}$ and $b_j^{(-96)}$ as follows:

— Set $b_{24}^{(-96)} = C_0 =$ 0x66,

— Set $b_{25}^{(-96)} = C_1 =$ 0xe9,

— Set $b_{26}^{(-96)} = C_2 =$ 0x4b,

— Set $b_{27}^{(-96)} = C_3 =$ 0xd4,

— Set $b_{28}^{(-96)} = C_4 =$ 0xef,

— Set $b_{29}^{(-96)} = C_5 =$ 0x8a,

— Set $b_{30}^{(-96)} = C_6 =$ 0x2c,

— Set $b_{31}^{(-96)} = C_7 =$ 0x3b,

— Set $a_0^{(-96)} = C_8 =$ 0x88,

— Set $a_1^{(-96)} = C_9 =$ 0x4c.

d) Set an 8-bit counter ctr = 1.

e) Perform the following steps for i=-96,-95,...,-1:

— $b_{31}^{(i)} = b_{31}^{(i)} \oplus \mathrm{ctr}$,

— ctr = 0x02$\otimes$ ctr,

— Set $S_{i+1} = \mathrm{Next}(S_i)$.

f) Output S_0.

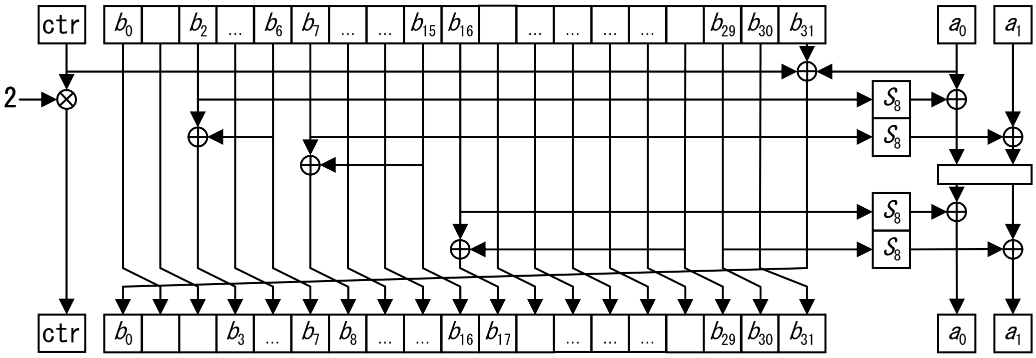

Figure 1 — State update during the initialization of *Enocoro*-128v2

6.1.3 Next-state function *Next*

The next-state function of *Enocoro*-128v2 is defined using the functions ρ and λ defined in 6.1.5 and 6.1.6 respectively. The next-state function *Next* of *Enocoro*-128v2 is as follows:

Input: State variable $S_i = (a^{(i)}, b^{(i)})$.

Ouput: Next value of the state variable $S_{i+1} = (a^{(i+1)}, b^{(i+1)})$.

— Set $a^{(i+1)} = \rho(a^{(i)}, b^{(i)})$.

— Set $b^{(i+1)} = \lambda(b^{(i)}, a_0{}^{(i)})$.

— Set $S_{i+1} = (a^{(i+1)}, b^{(i+1)})$.

— Output S_{i+1}.

6.1.4 Keystream function Strm

The keystream function *Strm* is as follows:

Input: State variable S_i.

Output: Keystream block Z_i.

— Set $Z_i = a_1{}^{(i)}$.

— Output Z_i.

The state is updated and the keystream is generated as sketched in Figure 2.

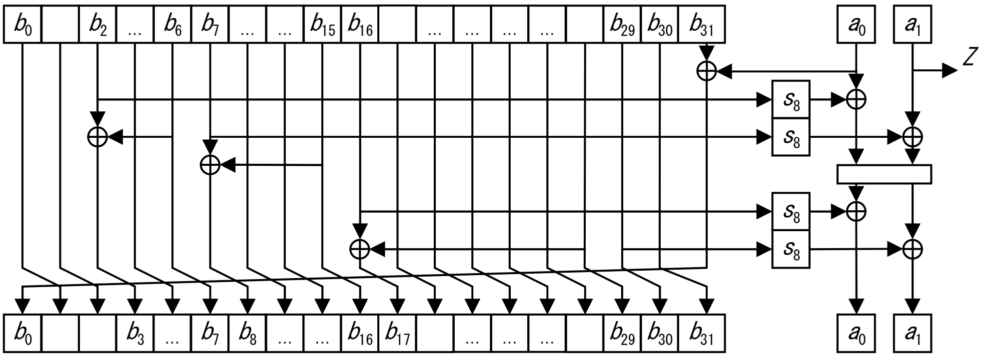

Figure 2 — State update during the keystream generation of *Enocoro*-128v2

6.1.5 Function *ρ*

The function ρ is composed of XORs, a non-linear transformation using the function S_8, and a linear transformation using the matrix L_{8432}. The function ρ is as follows:

Input: State variable $S_i = (a^{(i)}, b^{(i)})$.

Output: The next value of the state variable $a^{(i+1)}$.

- Set $u_0 = a_0^{(i)} \oplus s_8[b_2^{(i)}]$.
- Set $u_1 = a_1^{(i)} \oplus s_8[b_7^{(i)}]$.
- Set $(v_0, v_1) = L_{8432}(u_0, u_1)$,
- Set $a_0^{(i+1)} = v_0 \oplus s_8[b_{16}^{(i)}]$,
- Set $a_1^{(i+1)} = v_1 \oplus s_8[b_{29}^{(i)}]$.
- Output $a^{(i+1)}$.

6.1.6 Function *λ*

The function λ is as follows:

Input: State variable $S_i = (a^{(i)}, b^{(i)})$.

Output: The next value of the state variable $b^{(i+1)}$.

- Set $b_j^{(i+1)} = b_{j-1}^{(i)}$, for $j \neq 0, 3, 8, 17$,
- Set $b_0^{(i+1)} = b_{31}^{(i)} \oplus a_0^{(i)}$,
- Set $b_3^{(i+1)} = b_2^{(i)} \oplus b_6^{(i)}$,
- Set $b_8^{(i+1)} = b_7^{(i)} \oplus b_{15}^{(i)}$,
- Set $b_{17}^{(i+1)} = b_{16}^{(i)} \oplus b_{28}^{(i)}$,
- Output $b^{(i+1)}$.

6.1.7 Function L_{8432}

The function L_{8432} is the internal function of the ρ function. Denote the input and the output to the L_{8432} function by U and V respectively. The function L_{8432} is as follows:

Input: 16-bit string U.

Output: 16-bit string V.

— Set $(u_0, u_1) = U$, where u_i is an 8-bit string and an element of $GF(2^8)$.

— Set

$$\begin{pmatrix} v_0 \\ v_1 \end{pmatrix} = L_{8432}(u_0, u_1) = \begin{pmatrix} 1 & 1 \\ 1 & 0x02 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \end{pmatrix}.$$

— Set $V = v_0 \,||\, v_1$.

— Output V.

6.1.8 Function S_8

Function S_8 uses operations over the finite field $GF(2^4)$. In the polynomial representation, $GF(2^4)$ is realized as $GF(2)[x] / \phi_{41}(x)$, where $\phi_{41}(x)$ is an irreducible polynomial of degree 4 defined over $GF(2)$. The *Enocoro*-128v2 keystream generator uses the following irreducible polynomial:

$\phi_{41}(x) = x^4 + x + 1$,

The function S_8 is a permutation which maps 8-bit inputs to 8-bit outputs. It has an SPS (Substitution, permutation, substitution) structure and it consists of 4 small Sboxes s_4 which map 4-bit inputs to 4-bit outputs and a linear transformation l defined by a 2-by-2 matrix over $GF(2^4)$. The linear transformation l is defined as

$$l(x, y) = \begin{pmatrix} 1 & 0x4 \\ 0x4 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix}, \quad x, y \in GF(2^4)$$

Denote the input and the output to the S_8 function by X and Z respectively. The function S_8 is as follows:

Input: 8-bit string X.

Output: 8-bit string Z.

— Set $(x_0, x_1) = X$, where x_i is a 4-bit string and an element of $GF(2^4)$.

— Set

$$y_0 = s_4[s_4[x_0] \oplus 0x4 \otimes s_4[x_1] \oplus 0xa],$$
$$y_1 = s_4[0x4 \otimes s_4[x_0] \oplus s_4[x_1] \oplus 0x5],$$

where 0x4, 0x5, 0xa are the hexadecimal expressions of elements of $GF(2^4)$, and s_4 is defined as

$s_4[16] = \{1, 3, 9, 10, 5, 14, 7, 2, 13, 0, 12, 15, 4, 8, 6, 11\}$.

— *Set* $Z = (y_0 \,||\, y_1) \lll_8 1$.

— Output Z.

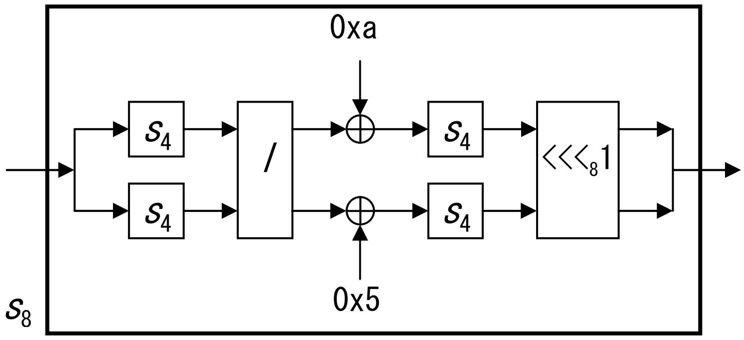

Figure 3 — Sbox S_8

Function S_8 is defined using a substitution table as follows:

S_8 [256] = {

```
99, 82, 26, 223, 138, 246, 174, 85, 137, 231, 208, 45, 189, 1, 36, 120,
27, 217, 227, 84, 200, 164, 236, 126, 171, 0, 156, 46, 145, 103, 55, 83,
78, 107, 108, 17, 178, 192, 130, 253, 57, 69, 254, 155, 52, 215, 167, 8,
184, 154, 51, 198, 76, 29, 105, 161, 110, 62, 197, 10, 87, 244, 241, 131,
245, 71, 31, 122, 165, 41, 60, 66, 214, 115, 141, 240, 142, 24, 170, 193,
32, 191, 230, 147, 81, 14, 247, 152, 221, 186, 106, 5, 72, 35, 109, 212,
30, 96, 117, 67, 151, 42, 49, 219, 132, 25, 175, 188, 204, 243, 232, 70,
136, 172, 139, 228, 123, 213, 88, 54, 2, 177, 7, 114, 225, 220, 95, 47,
93, 229, 209, 12, 38, 153, 181, 111, 224, 74, 59, 222, 162, 104, 146, 23,
202, 238, 169, 182, 3, 94, 211, 37, 251, 157, 97, 89, 6, 144, 116, 44,
39, 149, 160, 185, 124, 237, 4, 210, 80, 226, 73, 119, 203, 58, 15, 158,
112, 22, 92, 239, 33, 179, 159, 13, 166, 201, 34, 148, 250, 75, 216, 101,
133, 61, 150, 40, 20, 91, 102, 234, 127, 206, 249, 64, 19, 173, 195, 176,
242, 194, 56, 128, 207, 113, 11, 135, 77, 53, 86, 233, 100, 190, 28, 187,
183, 48, 196, 43, 255, 98, 65, 168, 21, 140, 18, 199, 121, 143, 90, 252,
205, 9, 79, 125, 248, 134, 218, 16, 50, 118, 180, 163, 63, 68, 129, 235
```

};

6.2 *Enocoro*-80 keystream generator

6.2.1 Introduction to *Enocoro*-80

Enocoro-80 is a keystream generator which uses an 80-bit secret key K, a 64-bit initialization vector IV, and a state variable S_i ($i \geq 0$) consisting of 22 bytes, and outputs a keystream block Z_i at every iteration of the function *Strm*.

NOTE This keystream generator was originally proposed in [6].

The state variable S_i is sub-divided into a 2-byte variable:

$$a^{(i)} = (a_0^{(i)}, a_1^{(i)}),$$

where $a_j^{(i)}$ is a byte (for $j = 0, 1$), and a 20-byte variable:

$$b^{(i)} = (b_0^{(i)}, b_1^{(i)}, \ldots, b_{19}^{(i)}),$$

where $b_j^{(i)}$ is a byte (for $j = 0, 1, \ldots, 19$).

The *Init* function, defined in detail in 6.2.2, takes as input the 80-bit key K and the 64-bit initializing vector IV, and produces the initial value of the state variable $S_0 = (a^{(0)}, b^{(0)})$.

The *Next* function, defined in detail in 6.2.3, takes as input the 22-byte state variable $S_i = (a^{(i)}, b^{(i)})$ and produces as output the next value of the state variable $S_{i+1} = (a^{(i+1)}, b^{(i+1)})$.

The *Strm* function, defined in detail in 6.3.5, takes as input the 22-byte state variable $S_i = (a^{(i)}, b^{(i)})$ and produces as output the keystream block Z_i.

Function L_{8431} uses operations over the finite field $GF(2^8)$. In the polynomial representation, $GF(2^8)$ is realized as $GF(2)[x] / \phi_{8431}(x)$, where $\phi_{8431}(x)$ is an irreducible polynomial of degree 8 defined over $GF(2)$. The *Enocoro*-80 keystream generator uses the following irreducible polynomial:

$$\phi_{8431}(x) = x^8 + x^4 + x^3 + x + 1.$$

6.2.2 Initialization function *Init*

The initialization of *Enocoro*-80 involves five steps. The initialization function *Init* is as follows:

Input: 80-bit key K, 64-bit initialization vector IV.

Output: Initial value of the state variable $S_0 = (a^{(0)}, b^{(0)})$.

a) Use the key K to set part of the state variable $b_j^{(-40)}$ as follows:

— Set($K_0||K_1||...||K_9$) = K, where K_j is 8 bits for j=0,1,2...,9.

— For j=0,1,2...,9, set $b_j^{(-40)} = K_j$.

b) Use the initialization vector IV to set part of the state variable $b_j^{(-40)}$ as follows:

— Set ($I_0||I_1||,...,||I_7$) = IV, where I_j is 8 bits for j=0,1,2...,7.

— For j=0,1,2...,7, set $b_{j+10}^{(-40)} = I_j$.

c) Set the constants C_0, C_1, C_2, C_3 to set part of the state variable $a_j^{(-40)}$ and $b_j^{(-40)}$ as follows:

— Set $b_{18}^{(-40)} = C_0$ = 0x66,

— Set $b_{19}^{(-40)} = C_1$ = 0xe9,

— Set $a_0^{(-40)} = C_2$ = 0x4b,

— Set $a_1^{(-40)} = C_3$ = 0xd4.

d) Perform the following steps for i=-40,-39...,-1:

— Set S_{i+1} =Next(S_i).

e) Output S_0

6.2.3 Next-state function *Next*

The next-state function of *Enocoro-80* is defined using the functions ρ and λ defined in 6.2.5 and 6.2.6 respectively. The next-state function *Next* of *Enocoro*-80 is as follows:

Input: State variable $S_i = (a^{(i)}, b^{(i)})$.

Ouput: Next value of the state variable $S_{i+1} = (a^{(i+1)}, b^{(i+1)})$.

- Set $a^{(i+1)} = \rho(a^{(i)}, b^{(i)})$,
- Set $b^{(i+1)} = \lambda(b^{(i)}, a_0^{(i)})$,
- Set $S_{i+1} = (a^{(i+1)}, b^{(i+1)})$,
- Output S_{i+1}.

6.2.4 Keystream function *Strm*

The keystream function *Strm* is as follows:

Input: State variable S_i.

Output: Keystream block Z_i.

- Set $Z_i = a_1^{(i)}$.

Output Z_i.

6.2.5 Function *ρ*

The function ρ is composed of XORs, a non-linear transformation using the function S_8, a linear transformation using the matrix L_{8431}. The function S_8 is described in 6.1.8. The function ρ is as follows:

Input: State variable $S_i = (a^{(i)}, b^{(i)})$.

Output: The next value of the state variable $a^{(i+1)}$.

- Set $u_0 = a_0^{(i)} \oplus s_8[b_1^{(i)}]$,
- Set $u_1 = a_1^{(i)} \oplus s_8[b_4^{(i)}]$,
- Set $(v_0, v_1) = L_{8431}(u_0, u_1)$,
- Set $a_0^{(i+1)} = v_0 \oplus s_8[b_6^{(i)}]$,
- Set $a_1^{(i+1)} = v_1 \oplus s_8[b_{16}^{(i)}]$.
- Output $a^{(i+1)}$.

6.2.6 Function *λ*

The function λ is as follows:

Input: State variable $S_i = (a^{(i)}, b^{(i)})$.

Output: The next value of the state variable $b^{(i+1)}$.

- Set $b_j^{(i+1)} = b_{j-1}^{(i)}$, for $j \neq 0, 2, 5, 7$,
- Set $b_0^{(i+1)} = b_{19}^{(i)} \oplus a_0^{(i)}$,
- Set $b_2^{(i+1)} = b_1^{(i)} \oplus b_3^{(i)}$,

— Set $b_5^{(i+1)} = b_4^{(i)} \oplus b_5^{(i)}$,

— Set $b_7^{(i+1)} = b_6^{(i)} \oplus b_{15}^{(i)}$,

— Output $b^{(i+1)}$.

6.2.7 Function L_{8431}

The function L_{8431} is the internal function of the ρ function. Denote the input and the output to the L_{8431} function by U and V respectively. The function L_{8431} is as follows:

Input: 16-bit string U.

Output: 16-bit string V.

— Set $(u_0, u_1) = U$, where u_i is an 8-bit string and an element of $GF(2^8)$.

— Set

$$\begin{pmatrix} v_0 \\ v_1 \end{pmatrix} = L_{8431}(u_0, u_1) = \begin{pmatrix} 1 & 1 \\ 1 & 0x02 \end{pmatrix} \begin{pmatrix} u_0 \\ u_1 \end{pmatrix}.$$

where 0x02 is the hexadecimal representation of the element of $GF(2^8)$ which is realized as $GF(2)[x] / \phi_{8431}(x)$.

— Set $V = v_0 \parallel v_1$.

— Output V.

6.3 Trivium keystream generator

6.3.1 Overview

TRIVIUM is a keystream generator which takes as input an 80-bit secret key $K = (K_0, \ldots, K_{79})$, an 80-bit initialization vector $IV = (IV_0, \ldots, IV_{79})$, and generates up to 2^{64} bits of keystream $z_0, z_1, \ldots, z_{N-1}$.

NOTE This keystream generator was originally proposed in [7].

The keystream bits z_i are computed by combining the elements of three internal bit sequences $\{a_i\}$, $\{b_i\}$, and $\{c_i\}$, which themselves are generated by iterating three interconnected nonlinear recurrence relations. The exact relations are specified in 6.3.4.

The first 288 sequence bits involved in the recursion are initialized using the secret key, the initialization vector, and some constant bits. The next 1152 triplets (a_i, b_i, c_i), starting from index $i = -1152$, are computed recursively, but without producing any output. These 1152 initial iterations are referred to as blank rounds.

Each subsequent iteration, starting from $i = 0$, outputs one keystream bit z_i, which is computed by XORing together a subset of six sequence bits. This is repeated until all requested keystream bits have been generated.

In the following sections, the complete keystream generation algorithm is described more formally using the framework introduced in 5. The internal state S_i is defined in 6.3.2, and the functions *Init*, *Next*, and *Strm* are specified in 6.3.3, 6.3.4, and 6.3.5.

6.3.2 Internal State

Since each new triplet (a_i, b_i, c_i) only depends on a limited number of earlier sequence bits, there is no need to keep the entire sequences in memory. At any point in time i, it suffices for the algorithm to maintain an internal state S_i consisting of the following 288 sequence bits:

$$S_i = (a_{i-1}, \dots, a_{i-93}, b_{i-1}, \dots, b_{i-84}, c_{i-1}, \dots, c_{i-111}).$$

NOTE In a straightforward hardware implementation of TRIVIUM, this internal state would be stored in shift registers, as sketched in Figure4. The bits in the registers (represented by boxes in the figure) are shifted in a clockwise direction after each iteration.

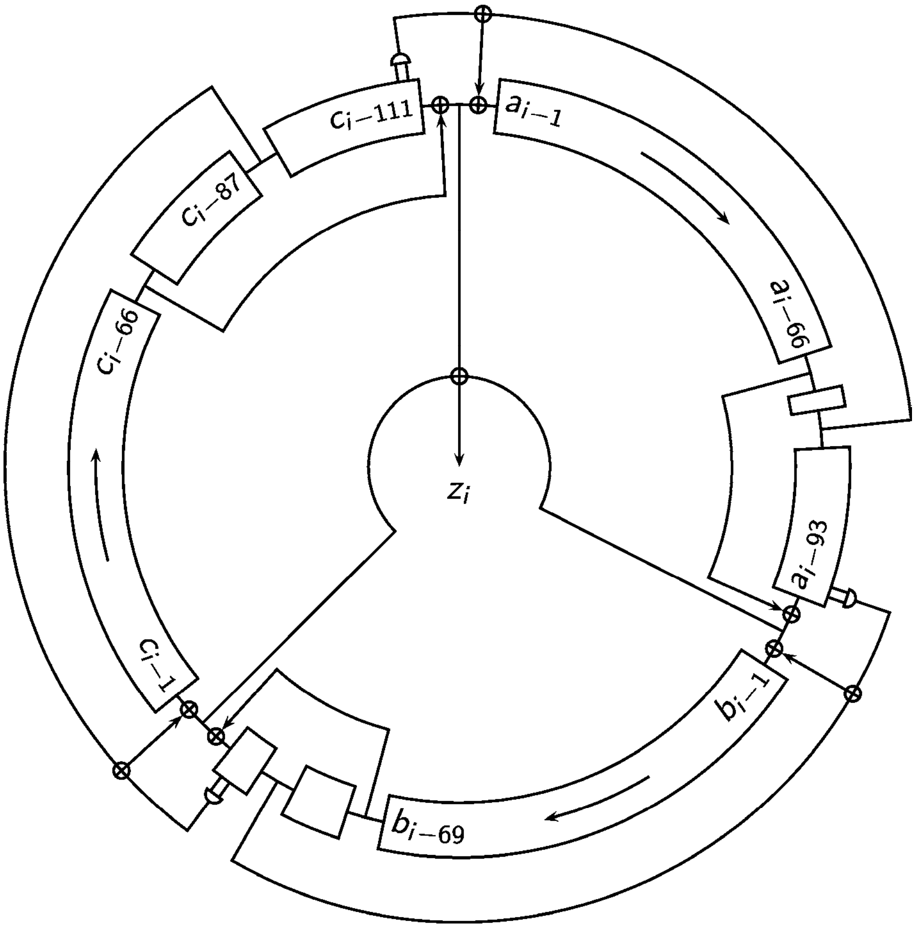

Figure 4 — An implementation of TRIVIUM using shift registers

6.3.3 Initialization function *Init*

The internal state of TRIVIUM is initialized using the following *Init* function.

Input: 80-bit key K, 80-bit initialization vector IV.

Output: Initial value of the internal state $S_0 = (a_{-1}, \dots, a_{-93}, b_{-1}, \dots, b_{-84}, c_{-1}, \dots, c_{-111})$.

a) Set $i = -1152$, and initialize the 288 bits of S_i as follows:

— Set $(a_{i-93}, \ldots, a_{i-1}) = (0, \ldots, 0, K_0, \ldots, K_{79})$.

— Set $(b_{i-84}, \ldots, b_{i-1}) = (0, \ldots, 0, IV_0, \ldots, IV_{79})$.

— Set $(c_{i-111}, \ldots, c_{i-1}) = (1, 1, 1, 0, \ldots, 0)$.

b) For $i = -1151, -1150, \ldots, -1, 0$:

— Set $S_i = \mathrm{Next}(S_{i-1})$.

c) Output S_0.

6.3.4 Next-state function *Next*

The next-state function *Next* is defined below.

Input: Internal state $S_i = (a_{i-1}, \ldots, a_{i-93}, b_{i-1}, \ldots, b_{i-84}, c_{i-1}, \ldots, c_{i-111})$.

Output: Next value of the internal state $S_{i+1} = (a_i, \ldots, a_{i-92}, b_i, \ldots, b_{i-83}, c_i, \ldots, c_{i-110})$.

a) Compute the bits a_i, b_i, and c_i:

— Set $a_i = c_{i-66} \oplus c_{i-111} \oplus c_{i-110} \cdot c_{i-109} \oplus a_{i-69}$.

— Set $b_i = a_{i-66} \oplus a_{i-93} \oplus a_{i-92} \cdot a_{i-91} \oplus b_{i-78}$.

— Set $c_i = b_{i-69} \oplus b_{i-84} \oplus b_{i-83} \cdot b_{i-82} \oplus c_{i-87}$.

b) Output $S_{i+1} = (a_i, \ldots, a_{i-92}, b_i, \ldots, b_{i-83}, c_i, \ldots, c_{i-110})$.

6.3.5 Keystream function *Strm*

The output function *Strm* is defined as follows.

Input: Internal state $S_i = (a_{i-1}, \ldots, a_{i-93}, b_{i-1}, \ldots, b_{i-84}, c_{i-1}, \ldots, c_{i-111})$.

Output: Keystream bit z_i.

Output $z_i = c_{i-66} \oplus c_{i-111} \oplus a_{i-66} \oplus a_{i-93} \oplus b_{i-69} \oplus b_{i-84}$.

Annex A
(normative)

Object Identifiers

This annex lists the object identifiers assigned to algorithms specified in this part of ISO/IEC 29192 and defines algorithm parameter structures.

```
LightweightCryptography-3 {
    iso(1) standard(0) lightweight-cryptography(29192) part3(3)
    asn1-module(0) algorithm-object-identifiers(0)}

DEFINITIONS EXPLICIT TAGS ::= BEGIN

-- EXPORTS All; --

-- IMPORTS None; --

OID ::= OBJECT IDENTIFIER -- Alias

-- Synonyms --
is29192-3 OID ::= {iso(1) standard(0) lightweight-cryptography(29192) part3(3)}
keystream-generators OID ::= {is29192-3 dedicated-keystream-generators(1)}
-- Lightweight dedicated keystream generators
enocoro-128v2 OID ::= {keystream-generators enocoro-128v2(1)}
enocoro-80 OID ::= {keystream-generators enocoro-80(2)}
trivium OID ::= {keystream-generators trivium(3)}

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

StreamAlgorithms ALGORITHM ::= {
{ OID enocoro-128v2 PARMS KeyLengthID } |
{ OID enocoro-80 PARMS KeyLengthID } |
{ OID trivium PARMS KeyLengthID },
... -- Expect additional algorithms --
}

KeyLength ::= INTEGER

KeyLengthID ::= CHOICE {
  int KeyLength,
  oid OID
}

-- Cryptographic algorithm identification --

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

END -- LightweightCryptography-3 --
```


Annex B
(informative)

Test vectors

B.1 Test vectors for *Enocoro*-128v2

B.1.1 Key, initialization vector, and keystream triplets

This clause provides numerical test vectors consisting of a 128-bit key, a 64-bit initialization vector, and the first 256 corresponding bits of keystream produced by *Enocoro*-128v2.

```
IV = 00 00 00 00 00 00 00 00
key = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Keystream =
63 d7 da 6b 55 73 7f cf 57 34 b6 77 3a e7 72 e8 e6 5c b3 bd a0 75 e6 b6 94 1c e3 e5 ca 28 2a 1e

IV = 00 10 20 30 40 50 60 70
Key = 00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
Keystream =
c8 c8 ee 43 3b 0d c0 40 e5 3b c5 06 ea 21 ad 82 20 05 88 89 b7 c8 45 b8 fb bc fc 26 66 d6 5a ce

IV = 80 90 a0 b0 c0 d0 e0 f0
Key = 0f 0e 0d 0c 0b 0a 09 08 07 06 05 04 03 02 01 00
Keystream =
f7 73 f9 b4 3f 1c b2 3c e4 19 8f 11 28 89 64 a3 e1 20 2e 6d ea 7d c8 07 7b 5d b1 5e cb 67 c8 6e
```

B.1.2 Sample internal states

The intermediate values of the state and buffer used to generate a keystream sequence are listed below.

```
IV  = 10 00 10 00 10 00 10 00
Key  = 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01 00
Keystream =
6c 1b 26 05 d1 97 f7 9f d4 60 4d 13 13 93 89 2e 29 6d 5d 50 f7 e6 07 10 ac 62 56 01 b3 e6 5e a6


round = -96
state = 88 4c
buffer = 01 00 01 00 01 00 01 00 01 00 01 00 01 00 01 00 10 00 10 00 10 00 10 00 66 e9 4b d4 ef 8a 2c
3b

round = -95
state = ee bf
buffer = b2 01 00 00 00 01 00 01 00 01 00 01 00 01 00 01 00 ff 00 10 00 10 00 10 00 66 e9 4b d4 ef 8a
2c

round = -94
state = 03 b6
buffer = c0 b2 01 00 00 00 01 00 00 00 01 00 01 00 01 00 01 d4 ff 00 10 00 10 00 10 00 66 e9 4b d4 ef
8a
```

```
round = -93
state = d6 29
buffer = 8d c0 b2 00 00 00 00 01 00 00 00 01 00 01 00 01 00 4a d4 ff 00 10 00 10 00 10 00 66 e9 4b d4
ef

round = -92
state = 92 8c
buffer = 31 8d c0 b2 00 00 00 00 00 00 00 00 01 00 01 00 01 e9 4a d4 ff 00 10 00 10 00 10 00 66 e9 4b
d4

     .

     .

     .

round = 0
state = 6a 6c
buffer = 7c 63 d7 8b 44 d5 a8 02 5f 29 87 6d 98 1f 46 e3 bc 45 72 32 9c d6 ca ed 0b fc 30 2f 8e 02 b4
85

round = 1
state = 61 1b
buffer = ef 7c 63 7f 8b 44 d5 a8 e1 5f 29 87 6d 98 1f 46 e3 32 45 72 32 9c d6 ca ed 0b fc 30 2f 8e 02
b4

round = 2
state = 42 26
buffer = d5 ef 7c b6 7f 8b 44 d5 ee e1 5f 29 87 6d 98 1f 46 cc 32 45 72 32 9c d6 ca ed 0b fc 30 2f 8e
02

round = 3
state = c8 05
buffer = 40 d5 ef 38 b6 7f 8b 44 ca ee e1 5f 29 87 6d 98 1f 76 cc 32 45 72 32 9c d6 ca ed 0b fc 30 2f
8e

round = 4
state = c7 d1
buffer = 46 40 d5 64 38 b6 7f 8b dc ca ee e1 5f 29 87 6d 98 e3 76 cc 32 45 72 32 9c d6 ca ed 0b fc 30
2f
```

B.2 Test vector for *Enocoro*-80

B.2.1 Key, initialization vector, and keystream triplets

This clause provides numerical test vectors consisting of an 80-bit key, a 64-bit initialization vector, and the first 128 corresponding bits of keystream produced by *Enocoro*-80.

```
Key = 00 00 00 00 00 00 00 00 00 00
IV = 00 00 00 00 00 00 00 00
Keystream = c9 22 79 45 6e be 3b ff d8 d4 73 12 3e ce b9 57

Key = 00 01 02 03 04 05 06 07 08 09
IV = 00 10 20 30 40 50 60 70
Keystream = 9b 0a 97 39 4b 58 72 73 3d bf 9e e5 0c 33 73 3e
```


B.2.2 Sample internal states

The intermediate values of the state and buffer used to generate a keystream are listed below.

```
Key = 00 00 00 00 00 00 00 00 00 00
IV = 00 00 00 00 00 00 00 00
Keystream = c9 22 79 45 6e be 3b ff d8 d4 73 12 3e ce b9 57

round = -40
state = 4b d4
buffer = 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 66 e9

round = -39
state = fc 3e
buffer = a2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 66

round = -38
state = a1 46
buffer = 9a a2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

round = -37
state = 47 28
buffer = a1 9a a2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

round = -36
state = 0e d3
buffer = 47 a1 9a a2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

     .

     .

     .

round = 0
state = 20 c9
buffer = 33 cb 98 13 d3 e0 5a 45 fc f3 ae b2 a7 0f 05 f7 0a 24 d2 4e

round = 1
state = 43 22
buffer = 6e 33 d8 98 13 33 e0 ad 45 fc f3 ae b2 a7 0f 05 f7 0a 24 d2

round = 2
state = 44 79
buffer = 91 6e ab d8 98 20 33 e5 ad 45 fc f3 ae b2 a7 0f 05 f7 0a 24

round = 3
state = e8 45
buffer = 60 91 b6 ab d8 b8 20 3c e5 ad 45 fc f3 ae b2 a7 0f 05 f7 0a

round = 4
state = 40 6e
buffer = e2 60 3a b6 ab 60 b8 87 3c e5 ad 45 fc f3 ae b2 a7 0f 05 f7
```


B.3 Test vector for Trivium

B.3.1 Key, initialization vector, and keystream triplets

This clause provides a numerical test vector consisting of an 80-bit key, an 80-bit initialization vector, and the first 128 corresponding bits of keystream produced by TRIVIUM.

Note that TRIVIUM is specified at a bit level, and is indifferent to the order in which these bits are grouped into bytes. In order to simplify the verification of this test vector on software platforms with different endianness conventions, each group of eight bits is printed in two different hexadecimal formats. The first format maps the first bit of each byte to the most significant bit, and is better suited for big-endian platforms; the second one uses the reverse ordering, and is more natural on little-endian platforms.

```
80-bit Key:                                      [MSB first]   [LSB first]

  0...31:  11110000 01000110 10101101 00010000   F0 46 AD 10   0F 62 B5 08
 32...63:  11011010 01110101 10000000 00101010   DA 75 80 2A   5B AE 01 54
 64...79:  11100101 01011111                     E5 5F         A7 FA

80-bit IV:                                       [MSB first]   [LSB first]

  0...31:  00010100 11110001 01101111 10111010   14 F1 6F BA   28 8F F6 5D
 32...63:  00100011 11010100 01001001 10011111   23 D4 49 9F   C4 2B 92 F9
 64...79:  00000110 11100011                     06 E3         60 C7

First 128 bits of keystream:                     [MSB first]   [LSB first]

  0...31:  00100101 00011100 00110110 10110110   25 1C 36 B6   A4 38 6C 6D
 32...63:  01101110 00100100 00011001 11111100   6E 24 19 FC   76 24 98 3F
 64...95:  01010111 10110001 01111101 11001110   57 B1 7D CE   EA 8D BE 73
 96..127:  00101000 10100111 01111111 11111000   28 A7 7F F8   14 E5 FE 1F
```

B.3.2 Internal Sequence Bits

The values of the internal sequence bits a_i, b_i, and c_i which were computed in order to generate the previous test vector, are listed below.

```
   i: -1280    -1272    -1264    -1256    -1248    -1240    -1232    -1224
       |        |        |        |        |        |        |        |
a[i]:                                         00000 00000000 11110000 01000110
b[i]:                                                   0000 00010100 11110001
c[i]:                         1110000 00000000 00000000 00000000 00000000 00000000

   i: -1216    -1208    -1200    -1192    -1184    -1176    -1168    -1160
       |        |        |        |        |        |        |        |
a[i]:  10101101 00010000 11011010 01110101 10000000 00101010 11100101 01011111
b[i]:  01101111 10111010 00100011 11010100 01001001 10011111 00000110 11100011
c[i]:  00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000

[the 1152 blank rounds start here]

   i: -1152    -1144    -1136    -1128    -1120    -1112    -1104    -1096
       |        |        |        |        |        |        |        |
a[i]:  01010101 01101000 10000110 11010011 10101100 00000001 01010111 00101010
b[i]:  11111000 10001000 00001010 00000000 00000111 11010100 01001010 10010111
c[i]:  10001010 00101110 11001110 00000011 00000000 00000001 10111100 10001101
```

```
   i: -1088    -1080    -1072    -1064    -1056    -1048    -1040    -1032
       |        |        |        |        |        |        |        |
a[i]:  11011000 00100000 11110111 10110110 01011101 01100001 01110001 11110101
b[i]:  11001111 10001110 11101000 10001110 01101100 01000100 00110100 00110110
c[i]:  00101111 00110010 00000000 11001100 11011101 00000010 10100100 11011001

   i: -1024    -1016    -1008    -1000    -992     -984     -976     -968
       |        |        |        |        |        |        |        |
a[i]:  10110001 00000011 10000111 10001001 00011100 10010001 11000100 01011101
b[i]:  01111100 01011101 11011010 10001001 11110100 01101000 00100010 10011110
c[i]:  11111001 10101100 11100000 00000011 01100001 11101010 11010111 11011111

   i: -960     -952     -944     -936     -928     -920     -912     -904
       |        |        |        |        |        |        |        |
a[i]:  11010000 01011000 11111110 00111001 11011011 01101010 00001100 10101111
b[i]:  11010011 10010100 10000001 00101010 10100101 00101111 11101100 11111110
c[i]:  11110001 11101100 10110101 11100001 11111001 11001011 11100100 10011001

   i: -896     -888     -880     -872     -864     -856     -848     -840
       |        |        |        |        |        |        |        |
a[i]:  11010010 11110110 00101110 01011110 00000001 01101001 11001101 10111010
b[i]:  10011010 11111011 01010010 10101010 11110001 10111000 00100101 01000011
c[i]:  10100001 00011001 01100110 10110011 11000100 01010000 00010110 11000111

   i: -832     -824     -816     -808     -800     -792     -784     -776
       |        |        |        |        |        |        |        |
a[i]:  11111010 00010101 11011101 01000110 11001000 01101000 00001001 10101111
b[i]:  10111000 00010111 00000101 11111100 01010011 00000000 00000110 10111110
c[i]:  01100100 11001110 10000010 01100110 10010000 01101010 00010101 10011111

   i: -768     -760     -752     -744     -736     -728     -720     -712
       |        |        |        |        |        |        |        |
a[i]:  11010011 00001010 10000110 11010011 10101101 11001110 00001110 10000111
b[i]:  00100100 11011101 01101001 11010101 01110101 01011100 00100100 00010101
c[i]:  00111111 10111001 01101011 01100111 11001111 11111100 00001000 01100101

   i: -704     -696     -688     -680     -672     -664     -656     -648
       |        |        |        |        |        |        |        |
a[i]:  10110100 10101011 00101111 00111011 01000001 11010111 11110001 11011110
b[i]:  10101001 01111010 01101111 01000111 11110100 01110010 00101000 01111100
c[i]:  00100101 11100001 10110111 01101111 00000111 11010001 01110101 11101001

   i: -640     -632     -624     -616     -608     -600     -592     -584
       |        |        |        |        |        |        |        |
a[i]:  10101011 00001100 10010100 10111010 00001010 00110000 10101110 00110011
b[i]:  10001011 11001100 11011010 01101000 11001000 00010011 10110111 01000101
c[i]:  10010111 00011010 01100011 10101000 01010011 11111001 11001100 00000111

   i: -576     -568     -560     -552     -544     -536     -528     -520
       |        |        |        |        |        |        |        |
a[i]:  01100111 01111000 11101101 11101100 10101000 01111000 11000000 00000101
b[i]:  00011101 11110000 10011101 00101000 10010011 11001011 10101101 00010010
c[i]:  01100100 00111101 01111101 00110000 10101101 00101001 01010001 10011100

   i: -512     -504     -496     -488     -480     -472     -464     -456
       |        |        |        |        |        |        |        |
a[i]:  10001001 01100100 00110000 00010110 11110110 00011110 01011010 11111100
b[i]:  01010111 01001111 11111111 00000110 00000010 01110111 10001100 11001010
c[i]:  10000100 11000011 11011000 11000010 00110100 10110110 10110101 01111110
```



```
   i: -448     -440     -432     -424     -416     -408     -400     -392
       |        |        |        |        |        |        |        |
a[i]:  00010111 10011010 10101101 11100010 10011101 11000100 01010100 00000001
b[i]:  10010010 11011110 01010001 00000110 00001110 10111100 00011111 10011000
c[i]:  00011011 11001000 01100011 01000100 10110111 10010010 11011000 00100100

   i: -384     -376     -368     -360     -352     -344     -336     -328
       |        |        |        |        |        |        |        |
a[i]:  10010101 11001010 00100100 01000011 11010010 00000110 11101101 00111000
b[i]:  11000111 11111111 00111000 01100001 00101110 10111100 11011100 11100101
c[i]:  10100000 00100000 10100111 10000001 11000101 10100011 10001000 11111111

   i: -320     -312     -304     -296     -288     -280     -272     -264
       |        |        |        |        |        |        |        |
a[i]:  11101010 00101111 10001011 11100101 10011111 10110001 10010101 00010111
b[i]:  01101101 00110011 00110110 11110000 10100011 01010100 00100011 10011011
c[i]:  00100001 10010100 00101100 00000011 01111101 10110100 11110110 00000000

   i: -256     -248     -240     -232     -224     -216     -208     -200
       |        |        |        |        |        |        |        |
a[i]:  01011100 00110001 11111000 00010100 11100001 10010110 11110110 00000000
b[i]:  11011001 01100101 10111100 11111110 11101101 10100010 10001011 00101000
c[i]:  10110010 10101110 00111000 00100011 11011010 10110001 10101101 10010010

   i: -192     -184     -176     -168     -160     -152     -144     -136
       |        |        |        |        |        |        |        |
a[i]:  11101110 01000101 00011110 00100010 00101101 10100001 00111001 10111000
b[i]:  01010110 11111111 10110011 01001000 10101000 00001110 11001011 10100001
c[i]:  11110101 01111100 11100011 00011100 11010111 11001101 00100100 00010110

   i: -128     -120     -112     -104     -96      -88      -80      -72
       |        |        |        |        |        |        |        |
a[i]:  00001010 01100100 11100101 11010001 01011011 10111000 00101011 11010010
b[i]:  00001011 11001001 11101100 01111101 11110100 01100011 11011111 01000100
c[i]:  00001001 00011110 01011101 10000000 10100011 11110010 11000101 01110011

   i: -64      -56      -48      -40      -32      -24      -16      -8
       |        |        |        |        |        |        |        |
a[i]:  10001100 01001111 00000001 11110100 11101010 00011101 01100010 01001110
b[i]:  11010001 00010111 11001000 10010011 10110000 00110010 11101101 11110000
c[i]:  11101110 11001100 01110011 10111101 01110100 11010000 11111100 11110011

[the keystream generation starts here]

   i:  0        8        16       24       32       40       48       56
       |        |        |        |        |        |        |        |
a[i]:  10110000 11010000 00101100 10000111 01110001 00001100 00111011 10010111
b[i]:  00110000 11000001 11100010 10110110 11111010 01010001 10001001 11110011
c[i]:  10001101 00110110 00000100 10001000 10001001 11101111 10110010 10000001

z[i]:  00100101 00011100 00110110 10110110 01101110 00100100 00011001 11111100

   i:  64       72       80       88       96       104      112      120
       |        |        |        |        |        |        |        |
a[i]:  11111101 01010000 01101011 11100100 00000011 10011000 10110111 10000000
b[i]:  01001010 10011011 11011010 10010001 11011000 00011001 00000011 11111110
c[i]:  00010111 10011100 11101111 00010011 11111101 11110100 01101100 11111001

z[i]:  01010111 10110001 01111101 11001110 00101000 10100111 01111111 11111000
```

B.3.3 Internal State

As mentioned in 6.3.2, a typical implementation of TRIVIUM will only need to maintain an internal state of 288 bits. The content of the internal state S_0 after the 1152 blank rounds is printed below.

```
   i: -128     -120     -112     -104     -96      -88      -80      -72
       |        |        |        |        |        |        |        |
a[i]:                                         11011 10111000 00101011 11010010
b[i]:                                                   0011 11011111 01000100
c[i]:                      1011101 10000000 10100011 11110010 11000101 01110011

   i: -64      -56      -48      -40      -32      -24      -16      -8
       |        |        |        |        |        |        |        |
a[i]:  10001100 01001111 00000001 11110100 11101010 00011101 01100010 01001110
b[i]:  11010001 00010111 11001000 10010011 10110000 00110010 11101101 11110000
c[i]:  11101110 11001100 01110011 10111101 01110100 11010000 11111100 11110011
```

B.3.4 Parallelism

The following test vector illustrates how TRIVIUM's parallelism enables implementers to compute 64 bits of b_i at once, using three 64-bit XOR operations and one 64-bit AND operation.

```
   i:  0        8        16       24       32       40       48       56
       |        |        |        |        |        |        |        |
 (1):  10100011 00010011 11000000 01111101 00111010 10000111 01011000 10010011
 (2):  11011101 11000001 01011110 10010100 01100010 01111000 00001111 10100111
 (3):  10111011 10000010 10111101 00101000 11000100 11110000 00011111 01001110
 (4):  01110111 00000101 01111010 01010001 10001001 11100000 00111110 10011101
 (5):  01111101 00010011 01000100 01011111 00100010 01001110 11000000 11001011
-----
 (6):  00110000 11000001 11100010 10110110 11111010 01010001 10001001 11110011

 (1) = a[i - 66]
 (2) = a[i - 93]
 (3) = a[i - 92]
 (4) = a[i - 91]
 (5) = b[i - 78]

 (6) = b[i] = (1) XOR (2) XOR [(3) AND (4)] XOR (5)
```

Annex C
(informative)

Guidance on implementation and use

C.1 Trivium

C.1.1 Parallelism

A useful feature of the recurrence relations used in TRIVIUM is that the bits computed at a given point in time only affect subsequent computations after a delay of at least 66 iterations. As a consequence, up to 66 consecutive iterations (the most natural choices are 8, 16, 32, or 64) can be computed in parallel without any interference. An illustration of this property is given in B.3.4.

NOTE Note that there are probably not many applications of TRIVIUM for which it would *not* make sense to exploit this parallelism to at least some extent. Parallel hardware implementations can achieve a significantly lower power consumption or higher throughput in exchange for a modest increase in area. In software, TRIVIUM's parallelism makes it possible to take advantage of the largest word size available on a given architecture.

C.1.2 Recommended Use of Initialization Values

This section provides recommendations on how to use initialization vectors in the most effective way. Thus this technique can be applied to all of the stream ciphers specified in this part of ISO/IEC 29192. In the remainder of this clause, we consider an example of Trivium. Given the fact that TRIVIUM uses a relatively short 80-bit secret key, an improper use of initialization vectors may reduce its security to a dangerously low level. It is important to note that TRIVIUM's initialization vector serves two purposes:

a) It allows data, encrypted with the same secret key, to be split into chunks which can be decrypted in arbitrary order.

b) It increases the security level against generic attacks.

In order to better reflect these two different purposes, it is useful to split the 80-bit initialization vector IV into two components, I and V:

$$(IV_{79}, \dots , IV_0) = (0, \dots , 0, I_{n-1}, \dots , I_0) \oplus (V_{79}, \dots , V_0).$$

The first component I is a simple n-bit counter which uniquely identifies each chunk of data. It is assumed to be publicly known, and if its value cannot be derived in any other way, then it needs to be transmitted for each chunk. Its length n depends on the maximum number of randomly accessible data chunks that the application should be able to encrypt under a single key. Note that there is often no need to make n very large, as illustrated in the following examples.

EXAMPLE 1 A 1× speed DVD drive reads data at 10Mbit/s and has a typical access time of 100ms. In order to access arbitrary parts of an encrypted disc without causing any additional delays, the decryption device will have to generate keystream at a speed of 10Mbit/s and be able to reach any point in the keystream within 100ms. This requirement can easily be met by reinitializing TRIVIUM with a different value of I after each chunk of 1Mbit. In this case, a 16-bit counter I would suffice to encrypt a 4.7GB disc.

EXAMPLE 2 In applications involving real-time communication (e.g., voice conversations), it typically does not make sense to decrypt data in any different order than the one used during encryption. The whole conversation can hence be encrypted as a single stream, eliminating the need for a counter, i.e., n = 0. Note however that in order to compensate for small synchronization differences, or for data packets arriving out-of-order or being dropped, the keystream bits will probably need to be generated at a slightly higher speed than the transmission rate and temporarily kept in a buffer.

The second component V can be used to increase TRIVIUM's resistance against generic attacks, and the most effective way to do so, is to treat it as an additional secret key. That is, whenever a new key is needed, both K and V are initialized simultaneously using a larger 160-bit secret key K':

$$(K_0, \dots , K_{79}, V_0, \dots , V_{79}) = (K'_0, \dots , K'_{79}, K'_{80}, \dots , K'_{159}).$$

It is important to realize that the use of such an extended key K' will not necessarily increase the security of TRIVIUM against dedicated attacks. The guess-and-determine attack proposed by Maximov and Biryukov [4], for instance, requires an effort roughly equivalent to an exhaustive search over a 90-bit key space, and this does not depend on how TRIVIUM is initialized. Generic bruteforce attacks, on the other hand, will necessitate considerably larger computational resources, especially when n can be kept relatively small. Considering the fact that the most efficient dedicated attacks require the adversary to intercept very large amounts of data (in the order of hundreds of petabytes in the case of [4]), the use of an extended key will in practice significantly increase the security margin of TRIVIUM.

Should users decide, because of specific constraints of the application, to deviate from the recommended procedure outlined above, then they should at least take measures to enforce the following rules.

a) Two different streams of data should never be encrypted with the same key K and the same initialization vector IV. A violation of this rule would expose the XOR of the two plaintexts to the adversary. If the two data streams contain some redundancy, then this information would often suffice to recover both of them.

b) The same initialization vector should not be reused with a large number of different keys. Suppose for instance that a single publicly known initialization vector would be used with 2^{24} (16 million) different 80-bit secret keys. In that case, recovering at least one of those 80-bit keys would not be harder than recovering a single 56-bit key.

C.2 Enocoro

C.2.1 Recommended Use for Extended Initialization Values

In some applications, initialization vector spaces are required to be larger than those of stream ciphers specified in this part of ISO/IEC 29192.

This section provides recommendations on how to use ciphers for extended initialization vectors in a secure way. Thus this technique can be applied to all of the stream ciphers specified in this part of ISO/IEC 29192. In the remainder of this clause, we consider an example of *Enocoro*-128v2 that uses a relatively short 64-bit initialization vector. Hereafter, we describe the case of using an extended 128-bit initialization vector *IV* for *Enocoro*-128v2.

According to this extension, the changes are made only at the procedures b) and c) in the Initialization function Init specified in 6.1.2 as follows:

b) Set the initialization vector *IV* into the part of the state variable $b_j^{(-96)}$ as follows:

— Set $(I_0||I_1||\dots||I_{15}) = IV$, where I_j is 8 bits for j =0,1,2,...,15.

— For j =0,1,2,...,15, set $b_{j+16}^{(-96)} = I_j$.

c) Set the constants into the part of the state variable $a_j^{(-96)}$ as follows:

— Set $a_0^{(-96)} = C_8$ = 0x88,

— Set $a_1^{(-96)} = C_9$ = 0x4c.

Annex D
(informative)

Feature Table

This annex shows lightweight properties of the cryptographic algorithms described in this document. ISO/IEC 29192-1:2012, Annex C gives hardware metrics for lightweight stream ciphers. Based on the metrics, the lightweight properties of Enocoro and Trivium are summarized in Table D.1.

Table D.1 — Lightweight properties of Enocoro and Trivium

	Algorithm name		
	Enocoro		Trivium
Key size [bits]	80	128	80
Chip area [GE]	2700	4100	2599
Cycles for initialization [CLK]	40	96	1152
Bits per cycles [bits/CLK]	8	8	1
Power[a] [GE]	2700	4100	2599
Energy[a] [GE*CLK]	2700	4100	2599
Energy per bit[a] [GE*CLK/bits]	338	513	2599
Technology [µm]	0,18	0.09	0,13
Reference	[6]	[5]	[3]

[a] Estimated using hardware metrics for lightweight cryptography in ISO/IEC 29192-1:2012, Annex C.

Annex E
(informative)

Computation over a finite field

EXAMPLE E.g. $C = A \otimes B$: In this operation, the finite field is represented as a selected irreducible polynomial $F(x)$ of degree n with binary coefficients, the n-bit blocks $A = \{a_{n-1}, a_{n-2}, ..., a_0\}$ and $B = \{b_{n-1}, b_{n-2}, ..., b_0\}$ (where the a_i and b_i are bits) are represented as the polynomials, $A(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + ... + a_0$ and $B(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + ... + b_0$ respectively, then let $C(x) = A(x) \bullet B(x) \bmod F(x)$, i.e. $C(x)$ is the polynomial of degree at most n–1 obtained by multiplying $A(x)$ and $B(x)$, dividing the result by $F(x)$, and then taking the remainder. If $C(x) = c_{n-1}x^{n-1} + c_{n-2}x^{n-2} + ... + c_0$ (where the c_i are bits) then let C be the n-bit block $\{c_{n-1}, c_{n-2}, ..., c_0\}$. For instance, we introduce a function *Xtime* which implements the multiplication by 2 in GF(2^8) using an irreducible polynomial $\phi_{8431}(x) = x^8 + x^4 + x^3 + x + 1$ which is represented as 0x11b. *Xtime* can be implemented as follows:

```
Xtime(X ) {
   if (X & 0x80) {
      X  = X <<8 1;
      X  = X ⊕ 0x1b;
   } else {
      X = X << 1;
   }
   return X;
}
```


Bibliography

[1] ISO/IEC 18033-1:2005, *Information technology — Security techniques — Encryption algorithms — Part 1: General*

[2] ISO/IEC 18033-4:2011, *Information technology — Security techniques — Encryption algorithms — Part 4: Stream ciphers*

[3] T. Good and M. Benaissa, “Hardware results of selected stream cipher candidates”, available at http://www.ecrypt.eu.org/stream/papersdir/2007/023.pdf

[4] A. Maximov and A. Biryukov, “Two Trivial Attacks on TRIVIUM”. In C. M. Adams, A. Miri, and M. J. Wiener, editors, Selected Areas in Cryptography, SAC 2007, volume 4876 of Lecture Notes in Computer Science, pages 36–55. Springer-Verlag, 2007

[5] Hitachi, Ltd, “Stream Cipher Enocoro Evaluation Report”. CRYPTREC submission package. http://www.hitachi.com/rd/yrl/crypto/enocoro/

[6] D. Watanabe, K. Ideguchi, J. Kitahara, K. Muto, H. Furuichi, T. Kaneko, “Enocoro-80: A Hardware Oriented Stream Cipher”. ARES 2008: pages 1294-1300

[7] C. De Canniere and B. Preneel, "Trivium Specifications". eSTREAM submission package. http://www.ecrypt.eu.org/stream/p3ciphers/trivium/trivium_p3.pdf

ICS 35.040

Price based on 28 pages