

# **FOX**

## **Specifications**

### **Version 1.0**

**EPFL I&C Technical Report IC/2003/82**

**Pascal Junod and Serge Vaudenay**

**16-12-2003**



Laboratory for Security and Cryptography (LASEC)  
EPFL  
CH-1015 Lausanne, Switzerland

# Abstract

In this document, we describe the design of a new family of block ciphers, named FOX. The main goals of this design, besides a very high security level, are a large implementation flexibility on various platforms as well as high performances. The high-level structure is based on a Lai-Massey scheme, while the round functions are substitution-permutation networks. In addition, we propose a new design of strong and efficient key-schedule algorithms.

FOX is the result of a joint project with the company *Mediacrypt AG* in Zürich, Switerland (website <http://www.mediacrypt.com>); the design has furthermore benefited from expert reviews of Prof. Jacques Stern, École Normale Supérieure, Paris (France) and of Prof. David Wagner, University of California, Berkeley (USA).

FOX may be subject to patenting and licensing issues. Please contact Mediacrypt (email [info@mediacrypt.com](mailto:info@mediacrypt.com)) for more information about them.

# Contents

<b>1</b>	<b>Notations and Conventions</b>	<b>7</b>
1.1	Algorithm Names . . . . .	7
1.2	Hexadecimal Notation . . . . .	7
1.3	Mathematical Operations . . . . .	8
1.4	Prefixes, Indices and Suffixes . . . . .	8
1.5	Byte Ordering . . . . .	9
1.6	Finite Field $\text{GF}(2^8)$ . . . . .	9
1.6.1	Addition in $\text{GF}(2^8)$ . . . . .	10
1.6.2	Multiplication in $\text{GF}(2^8)$ . . . . .	10
<b>2</b>	<b>Algorithms Description</b>	<b>11</b>
2.1	FOX64/ $k/r$ Skeleton . . . . .	11
2.1.1	Encryption . . . . .	11
2.1.2	Decryption . . . . .	12
2.2	FOX128/ $k/r$ Skeleton . . . . .	12
2.2.1	Encryption . . . . .	12
2.2.2	Decryption . . . . .	12
2.3	Internal Functions . . . . .	13
2.3.1	Definitions of <code>lmor64</code> , <code>lmid64</code> , <code>lmio64</code> . . . . .	13
2.3.2	Definitions of <code>elmor128</code> , <code>elmid128</code> , <code>elmio128</code> . . . . .	14
2.3.3	Definitions of <code>or</code> and <code>io</code> . . . . .	16
2.3.4	Definition of <code>f32</code> . . . . .	16
2.3.5	Definition of <code>f64</code> . . . . .	18
2.3.6	Definition of <code>sigma4</code> , <code>sigma8</code> and <code>sbox</code> . . . . .	19
2.3.7	Definition of <code>mu4</code> . . . . .	19
2.3.8	Definition of <code>mu8</code> . . . . .	20
2.4	Key Schedule . . . . .	21
2.4.1	General Overview . . . . .	21
2.4.2	Definition of <code>KS64</code> . . . . .	21
2.4.3	Definition of <code>KS64h</code> . . . . .	23

2.4.4	Definition of KS128 . . . . .	23
2.4.5	Definition of P . . . . .	23
2.4.6	Definition of pad . . . . .	25
2.4.7	Definition of M . . . . .	25
2.4.8	Definition of D . . . . .	25
2.4.9	Definition of LFSR . . . . .	26
2.4.10	Definition of NL64 . . . . .	26
2.4.11	Definition of NL64h . . . . .	28
2.4.12	Definition of NL128 . . . . .	28
2.4.13	Definition of mix64 . . . . .	31
2.4.14	Definition of mix64h . . . . .	31
2.4.15	Definition of mix128 . . . . .	33
<b>3</b>	<b>Design Rationales . . . . .</b>	<b>34</b>
3.1	Skeletons . . . . .	34
3.1.1	Lai-Massey Scheme . . . . .	34
3.1.2	Extended Lai-Massey Scheme . . . . .	36
3.2	sbox Transformation . . . . .	38
3.3	mu4 and mu8 Transformations . . . . .	40
3.3.1	Linear Multipermutations . . . . .	40
3.3.2	Security properties of f32 and f64 . . . . .	41
3.4	Key Schedule . . . . .	43
3.4.1	General Rationales . . . . .	43
3.4.2	P-Part . . . . .	44
3.4.3	M-Part . . . . .	44
3.4.4	L-Part . . . . .	44
3.4.5	NLx-Part . . . . .	45
<b>4</b>	<b>Implementation Aspects . . . . .</b>	<b>46</b>
4.1	32/64-bit Platforms . . . . .	46
4.1.1	Subkeys Precomputation . . . . .	46
4.1.2	Implementation of f32 and f64 Using Table-Lookups . . . . .	46
4.1.3	Key-Schedule Algorithms . . . . .	49
4.2	8-bit Platforms . . . . .	50
4.2.1	Four Memory Usage Strategies . . . . .	50
4.2.2	Implementation of f32 . . . . .	51
4.2.3	Implementation of f64 . . . . .	51
4.2.4	Implementation of talpha and dalpha . . . . .	52
4.3	Hardware Implementations . . . . .	53
<b>A</b>	<b>Test Vectors . . . . .</b>	<b>55</b>

<b>B</b>	<b>Reference Implementations</b>	<b>57</b>
B.1	File README . . . . .	57
B.2	File Makefile . . . . .	57
B.3	File fox_portable.h . . . . .	58
B.4	File fox_error.h . . . . .	59
B.5	File fox_cst.h . . . . .	60
B.6	File fox_cst.c . . . . .	61
B.7	File fox_ctx.h . . . . .	62
B.8	File fox_ctx.c . . . . .	64
B.9	File fox64.h . . . . .	79
B.10	File fox64.c . . . . .	80
B.11	File fox128.h . . . . .	82
B.12	File fox128.c . . . . .	83
B.13	File fox_util.h . . . . .	86
B.14	File fox_util.c . . . . .	86

# Preface

This document presents the specifications and rationales of the block cipher family FOX; it is organized as follows: in §1, the conventions and notations used throughout this document are described. §2 describes formally the cipher family, while §3 gives the mathematical foundations and rationales behind FOX. §4 discusses several issues related to the implementation of FOX. In Appendix, a reference implementation written in C is given; its sole goal is to help to understand how FOX is defined and to furnish test vectors.

# Chapter 1

## Notations and Conventions

The notations, conventions and symbols used throughout this document are described here.

### 1.1 Algorithm Names

The name of the family of block ciphers described in this document is FOX. The different members of this family are denoted as follows:

Name	Block size	Key size	Rounds number
FOX64	64	128	16
FOX128	128	256	16
FOX64/ $k/r$	64	$k$	$r$
FOX128/ $k/r$	128	$k$	$r$

The following conditions must hold in the case of FOX64/ $k/r$  and FOX128/ $k/r$ :

- $12 \leq r \leq 255$
- $0 \leq k \leq 256$  with  $k$  divisible by 8.

### 1.2 Hexadecimal Notation

The hexadecimal notation will be intensively used in this document to write binary strings in a compact way. Numbers written in hexadecimal notations begins with the prefix 0x. For instance, 0x01234567 is a 32-bit value. The following table gives the correspondance between decimal digits, hexadecimal digits and binary values.

Decimal	0	1	2	3	4	5	6	7
Binary	0000	0001	0010	0011	0100	0101	0110	0111
Hexadecimal	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7
Decimal	8	9	10	11	12	13	14	15
Binary	1000	1001	1010	1011	1100	1101	1110	1111
Hexadecimal	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF

### 1.3 Mathematical Operations

Here is a list of some mathematical operations used throughout this document together with their meanings.

Mathematical Symbols		
Operation	Description	Example
$\lfloor a \rfloor$	“Floor” function	$\lfloor 12.34 \rfloor = 12$
$\lceil a \rceil$	“Ceil” function	$\lceil 12.34 \rceil = 13$
$a \oplus b$	Bitwise exclusive-OR	$0xABCD \oplus 0x1234 = 0xB9F9$
$a \wedge b$	Bitwise AND	$0xABCD \wedge 0x1234 = 0x0204$
$a \vee b$	Bitwise OR	$0xABCD \vee 0x1234 = 0xBBFD$
$a \ll n$	Logical left shift of $n$ positions	$0x03 \ll 1 = 0x06$
$a \gg n$	Logical right shift of $n$ positions	$0x03 \gg 1 = 0x01$
$\bar{a}$	Logical negation	$\overline{0xA} = 0x5$
$a  b$	Concatenation	$0xABCD  0x1234 = 0xABCD1234$
$a \oplus b$	Addition in $\text{GF}(2^8)$	$0x02 \oplus 0x06 = 0x04$
$a \cdot b$	Multiplication in $\text{GF}(2^8)$	$0x02 \cdot 0x60 = 0xC0$
$\triangleq$	Assignment operator	

Note that the  $\text{GF}(2^8)$  representation is defined in §1.5.

### 1.4 Prefixes, Indices and Suffixes

Here are some generic conventions used in the notation:

- A variable  $x$  written with the suffix  $_{(n)}$  (*i.e.*  $x_{(n)}$ ) indicates that  $x$  has a length of  $n$  bits. For instance,  $y_{(1)}$  is a single-bit variable and  $F_{(64)}$  is a 64-bit value. The suffix will be omitted if the context is clear.
- A variable  $x$  written with the suffix  $_{[a\dots n]}$  (*i.e.*  $x_{[a\dots b]}$ ) indicates the bit subset of the variable  $x$  beginning at position  $a$  (inclusive) and ending at position  $b$  (inclusive).
- Indexed variables are denoted as follows:  $x_i$  is a variable  $x$  indexed by  $i$ . A variable  $x$  indexed by  $i$  with a length of  $\ell$  bits is denoted  $x_{i(\ell)}$ . A C-like notation is used for indexing which means that indices begin with 0.



- The suffix  $L$  is used to denote the left half of a variable. For instance,  $x_L$  is the left half of the variable  $x$ .
- The suffix  $R$  is used to denote the right half of a variable. For instance,  $x_R$  is the right half of the variable  $x$ .
- The suffixes  $LL$ ,  $LR$ ,  $RL$ ,  $RR$  are used to denote *quarters* of a variable. For instance,  $X \triangleq X_{LL}||X_{LR}||X_{RL}||X_{RR}$ .
- In general, the input of a function  $f$  is denoted  $X$  and its output  $Y$ .

## 1.5 Byte Ordering

In this document, a big-endian ordering is assumed. The index of the most significant part in a variable is equal to 0, while the index corresponding to the least significant part is the largest one.

Here is an example: a 128-bit value  $Q_{(128)}$  can be written as

$$\begin{aligned}
 Q_{(128)} &= R_{0(64)}||R_{1(64)} \\
 &= S_{0(32)}||S_{1(32)}||S_{2(32)}||S_{3(32)} \\
 &= T_{0(8)}||T_{1(8)}||\dots||T_{14(8)}||T_{15(8)} \\
 &= U_{0(1)}||U_{1(1)}||\dots||U_{126(1)}||U_{127(1)}
 \end{aligned}$$

## 1.6 Finite Field $\text{GF}(2^8)$

Some of the mathematical operations used in FOX are the addition and the multiplication in the finite field with 256 elements, which is denoted  $\text{GF}(2^8)$ . We describe now the *representation* of  $\text{GF}(2^8)$  used in the FOX definition. Let be the following irreducible polynomial  $P(\alpha)$  over  $\text{GF}(2) = \{0, 1\}$ :

$$P(\alpha) \triangleq \alpha^8 + \alpha^7 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + 1 \quad (1.1)$$

Elements of the field are polynomials in  $\alpha$  of degree at most 7 with coefficients in  $\text{GF}(2)$ . Let  $s$  be an 8-bit binary string

$$s \triangleq s_{0(1)}||s_{1(1)}||s_{2(1)}||s_{3(1)}||s_{4(1)}||s_{5(1)}||s_{6(1)}||s_{7(1)} \quad (1.2)$$

The corresponding field element is

$$s_{0(1)}\alpha^7 + s_{1(1)}\alpha^6 + s_{2(1)}\alpha^5 + s_{3(1)}\alpha^4 + s_{4(1)}\alpha^3 + s_{5(1)}\alpha^2 + s_{6(1)}\alpha + s_{7(1)} \quad (1.3)$$

### 1.6.1 Addition in $\text{GF}(2^8)$

The addition in  $\text{GF}(2^8)$ , denoted  $\oplus$  is the usual addition of polynomials where the respective coefficients are added modulo 2. For instance,

$$(\alpha^7 + \alpha^6 + \alpha^3 + \alpha^2 + 1) \oplus (\alpha^6 + \alpha^5 + \alpha + 1) = \alpha^7 + \alpha^5 + \alpha^3 + \alpha^2 + \alpha \quad (1.4)$$

Note that the addition  $a \oplus b$  of two elements of  $\text{GF}(2^8)$  is equivalent to a bitwise exclusive-OR operation of their representation as an 8-bit binary string.

### 1.6.2 Multiplication in $\text{GF}(2^8)$

The multiplication in  $\text{GF}(2^8)$ , denoted “.”, is the usual multiplication of polynomials where the result is taken modulo the polynomial defined in (1.1) and coefficients are reduced modulo 2. The reduction modulo  $P(\alpha)$  can be computed by taking the rest of the Euclidean division of the product by  $P(\alpha)$ . For instance,

$$\begin{aligned} (\alpha^5 + \alpha^4 + \alpha^3) \cdot (\alpha^3 + \alpha + 1) &= \alpha^8 + \alpha^6 + \alpha^5 \\ &\equiv \alpha^7 + \alpha^4 + \alpha + 1 \pmod{P(\alpha)} \end{aligned}$$

## Chapter 2

# Algorithms Description

In this part of the document, we describe precisely both versions of FOX, *i.e.* the one having a 64-bit block size (FOX64/ $k/r$ ) and the one with a block size of 128 bits (FOX128/ $k/r$ ).

This chapter is organized as follows: in §2.1, the high-level structure of FOX64/ $k/r$ , which is a *Lai-Massey scheme*, is formally described, together with the encryption and decryption operations. In §2.2, the same is done for FOX128/ $k/r$ , which is built on an *Extended Lai-Massey scheme*. In §2.3, the *internal functions* f32 and f64 used in both algorithms are formally defined, together with their building blocks. Finally, in §2.4, the key-schedule algorithm is described.

### 2.1 FOX64/ $k/r$ Skeleton

In this section, we describe the high-level structure of the 64-bit version of FOX in a top-down approach.

The 64-bit version of FOX is the  $(r - 1)$ -times iteration of a *round function* denoted  $\text{Imor64}$ , followed by the application of a slightly modified version of  $\text{Imor64}$ , denoted by  $\text{Imid64}$ .  $\text{Imio64}$  is a function used during the decryption operation. Formally,  $\text{Imor64}$ ,  $\text{Imio64}$  and  $\text{Imid64}$  take all a 64-bit input  $X_{(64)}$ , a 64-bit *round key*  $RK_{(64)}$  and return a 64-bit output  $Y_{(64)}$ :

$$\text{Imor64, Imio64, Imid64} : \begin{cases} \{0, 1\}^{64} \times \{0, 1\}^{64} & \rightarrow \{0, 1\}^{64} \\ (X_{(64)}, RK_{(64)}) & \mapsto Y_{(64)} \end{cases} \quad (2.1)$$

#### 2.1.1 Encryption

The encryption  $C_{(64)}$  by FOX64/ $k/r$  of a 64-bit plaintext  $P_{(64)}$  is formally defined as

$$C_{(64)} \triangleq \text{Imid64}(\text{Imor64}(\dots (\text{Imor64}(P_{(64)}, RK_{0(64)}), \dots, RK_{r-2(64)}), RK_{r-1(64)}) \quad (2.2)$$

where

$$RK_{(r-64)} \triangleq RK_{0(64)} || RK_{1(64)} || \dots || RK_{r-1(64)} \quad (2.3)$$

is the subkey stream produced by the key schedule algorithm out of the key  $K_{(k)}$ .

### 2.1.2 Decryption

The decryption  $P_{(64)}$  by FOX64/ $k/r$  of a 64-bit ciphertext  $C_{(64)}$  is formally defined as

$$P_{(64)} \triangleq \text{Imid64}(\text{Imio64}(\dots(\text{Imio64}(C_{(64)}, RK_{r-1(64)}), \dots, RK_{1(64)}), RK_{0(64)}) \quad (2.4)$$

where

$$RK_{(r \cdot 64)} \triangleq RK_{0(64)} || RK_{1(64)} || \dots || RK_{r-1(64)} \quad (2.5)$$

is the subkey stream produced by the key schedule algorithm out of the key  $K_{(k)}$ , as for the encryption.

## 2.2 FOX128/ $k/r$ Skeleton

In this section, we describe the high-level structure of the 128-bit version of FOX in a top-down approach.

The 128-bit version of FOX is the  $(r-1)$ -times iteration of a *round function* denoted *elmor128*, followed by the application of a slightly modified version of *elmor128*, denoted by *elmid128*. *elmio128* is a function used during the decryption operation. Formally, *elmor128*, *elmio128* and *elmid128* all take a 128-bit input  $X_{(128)}$ , a 128-bit *round key*  $RK_{(128)}$  and return a 128-bit output  $Y_{(128)}$ :

$$\text{elmor128, elmio128, elmid128} : \begin{cases} \{0, 1\}^{128} \times \{0, 1\}^{128} & \rightarrow \{0, 1\}^{128} \\ (X_{(128)}, RK_{(128)}) & \mapsto Y_{(128)} \end{cases} \quad (2.6)$$

### 2.2.1 Encryption

The encryption  $C_{(128)}$  by FOX128/ $k/r$  of a 128-bit plaintext  $P_{(128)}$  is formally defined as

$$C_{(128)} \triangleq \text{elmid128}(\text{elmor128}(\dots(\text{elmor128}(P_{(128)}, RK_{0(128)}), \dots, RK_{r-2(128)}), RK_{r-1(128)}) \quad (2.7)$$

where

$$RK_{(r \cdot 128)} \triangleq RK_{0(128)} || RK_{1(128)} || \dots || RK_{r-1(128)} \quad (2.8)$$

is the subkey stream produced by the key schedule algorithm out of the key  $K_{(k)}$ .

### 2.2.2 Decryption

The decryption  $P_{(128)}$  by FOX128/ $k/r$  of a 128-bit ciphertext  $C_{(128)}$  is formally defined as

$$P_{(128)} \triangleq \text{elmid128}(\text{elmio128}(\dots(\text{elmio128}(C_{(128)}, RK_{r-1(128)}), \dots, RK_{1(128)}), RK_{0(128)}) \quad (2.9)$$

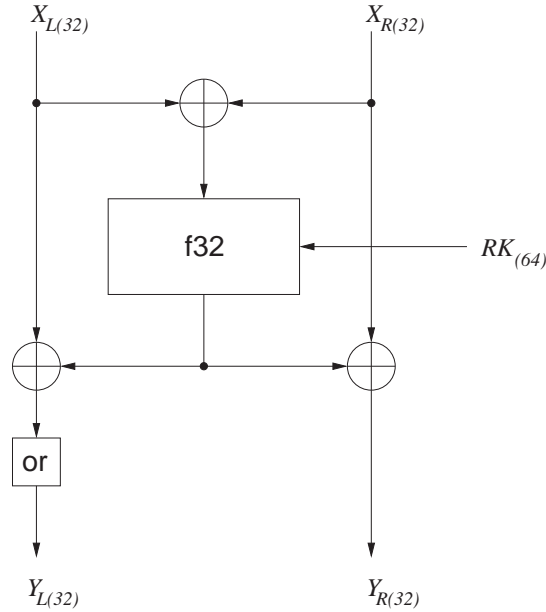


Figure 2.1: lmor64 Round Function

where

$$RK_{(r \cdot 128)} \triangleq RK_{0(128)} || RK_{1(128)} || \dots || RK_{r-1(128)} \quad (2.10)$$

is the subkey stream produced by the key schedule algorithm out of the key  $K_{(k)}$ , as for the encryption operation.

## 2.3 Internal Functions

In this section, we describe formally all the functions used internally in the core of both algorithms FOX64/ $k/r$  and FOX128/ $k/r$ .

### 2.3.1 Definitions of lmor64, lmid64, lmio64

In the 64-bit version of the algorithm, one uses three slightly different round functions.

The first one, **lmor64**, illustrated in Fig. 2.1, is built as a Lai-Massey scheme combined with an orthomorphism **or**, as described in [Vau00b]. This function transforms a 64-bit input  $X_{(64)}$  split in two parts  $X_{(64)} \triangleq X_{L(32)} || X_{R(32)}$  and a 64-bit round key  $RK_{(64)}$  in a 64-bit output

$Y_{(64)} \triangleq Y_{L(32)} || Y_{R(32)}$  as follows:

$$\begin{aligned} Y_{(64)} &= Y_{L(32)} || Y_{R(32)} = \text{Imor64} (X_{L(32)} || X_{R(32)}) \\ &\triangleq \text{or} (X_{L(32)} \oplus \text{f32} (X_{L(32)} \oplus X_{R(32)}, RK_{(64)})) || \\ &\quad (X_{R(32)} \oplus \text{f32} (X_{L(32)} \oplus X_{R(32)}, RK_{(64)})) \end{aligned}$$

The  $\text{Imid64}$  function is a slightly modified version of  $\text{Imor64}$ , namely it is the same one without the orthomorphism  $\text{or}$ :

$$\begin{aligned} Y_{(64)} &= Y_{L(32)} || Y_{R(32)} = \text{Imid64} (X_{L(32)} || X_{R(32)}) \\ &\triangleq (X_{L(32)} \oplus \text{f32} (X_{L(32)} \oplus X_{R(32)}, RK_{(64)})) || \\ &\quad (X_{R(32)} \oplus \text{f32} (X_{L(32)} \oplus X_{R(32)}, RK_{(64)})) \end{aligned}$$

Finally,  $\text{Imio64}$  is defined by

$$\begin{aligned} Y_{(64)} &= Y_{L(32)} || Y_{R(32)} = \text{Imio64} (X_{L(32)} || X_{R(32)}) \\ &\triangleq \text{io} (X_{L(32)} \oplus \text{f32} (X_{L(32)} \oplus X_{R(32)}, RK_{(64)})) || \\ &\quad (X_{R(32)} \oplus \text{f32} (X_{L(32)} \oplus X_{R(32)}, RK_{(64)})) \end{aligned}$$

### 2.3.2 Definitions of $\text{elmor128}$ , $\text{elmid128}$ , $\text{elmio128}$

In the 128-bit version of the algorithm, one uses also three slightly different round functions.

The first one,  $\text{elmor128}$ , illustrated in Fig. 2.2, is built as an *Extended Lai-Massey scheme* combined with two orthomorphisms  $\text{or}$ . This function transforms a 128-bit input  $X_{(128)}$  split in four parts  $X_{(128)} \triangleq X_{LL(32)} || X_{LR(32)} || X_{RL(32)} || X_{RR(32)}$  and a 128-bit round key  $RK_{(128)}$  in a 128-bit output  $Y_{(128)} \triangleq Y_{LL(32)} || Y_{LR(32)} || Y_{RL(32)} || Y_{RR(32)}$  as follows:

$$\begin{aligned} Y_{(128)} &= Y_{LL(32)} || Y_{LR(32)} || Y_{RL(32)} || Y_{RR(32)} = \text{elmor128} (X_{LL(32)} || X_{LR(32)} || X_{RL(32)} || X_{RR(32)}) \\ &\triangleq \text{or} \left( X_{LL(32)} \oplus \text{f64} ((X_{LL(32)} \oplus X_{LR(32)}) || (X_{RL(32)} \oplus X_{RR(32)}), RK_{(128)})_{L(32)} \right) || \\ &\quad \left( X_{LR(32)} \oplus \text{f64} ((X_{LL(32)} \oplus X_{LR(32)}) || (X_{RL(32)} \oplus X_{RR(32)}), RK_{(128)})_{L(32)} \right) || \\ &\quad \text{or} \left( X_{RL(32)} \oplus \text{f64} ((X_{LL(32)} \oplus X_{LR(32)}) || (X_{RL(32)} \oplus X_{RR(32)}), RK_{(128)})_{R(32)} \right) || \\ &\quad \left( X_{RR(32)} \oplus \text{f64} ((X_{LL(32)} \oplus X_{LR(32)}) || (X_{RL(32)} \oplus X_{RR(32)}), RK_{(128)})_{R(32)} \right) \end{aligned}$$

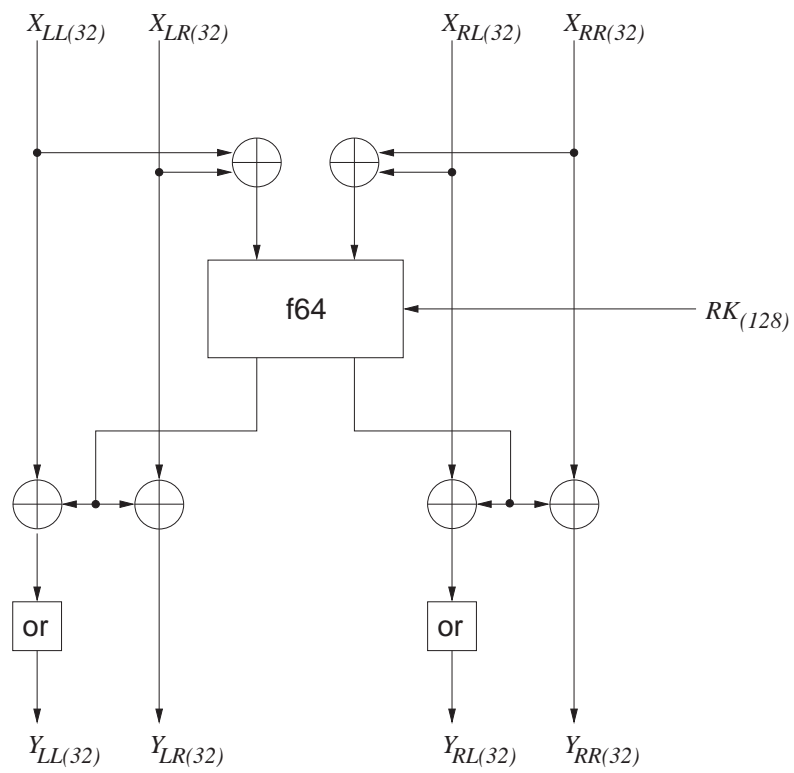


Figure 2.2: Round function elmor128

The `elmid128` function is a slightly modified version of `elmor128`, namely it is the same one without the orthomorphism `or`:

$$\begin{aligned}
Y_{(128)} &= Y_{LL(32)} \parallel Y_{LR(32)} \parallel Y_{RL(32)} \parallel Y_{RR(32)} = \text{elmid128} (X_{LL(32)} \parallel X_{LR(32)} \parallel X_{RL(32)} \parallel X_{RR(32)}) \\
&\triangleq \left( X_{LL(32)} \oplus \text{f64} ((X_{LL(32)} \oplus X_{LR(32)}) \parallel (X_{RL(32)} \oplus X_{RR(32)}), RK_{(128)})_{L(32)} \right) \parallel \\
&\quad \left( X_{LR(32)} \oplus \text{f64} ((X_{LL(32)} \oplus X_{LR(32)}) \parallel (X_{RL(32)} \oplus X_{RR(32)}), RK_{(128)})_{L(32)} \right) \parallel \\
&\quad \left( X_{RL(32)} \oplus \text{f64} ((X_{LL(32)} \oplus X_{LR(32)}) \parallel (X_{RL(32)} \oplus X_{RR(32)}), RK_{(128)})_{R(32)} \right) \parallel \\
&\quad \left( X_{RR(32)} \oplus \text{f64} ((X_{LL(32)} \oplus X_{LR(32)}) \parallel (X_{RL(32)} \oplus X_{RR(32)}), RK_{(128)})_{R(32)} \right)
\end{aligned}$$

Finally, `elmio128` is defined by

$$\begin{aligned}
Y_{(128)} &= Y_{LL(32)} \parallel Y_{LR(32)} \parallel Y_{RL(32)} \parallel Y_{RR(32)} = \text{elmio128} (X_{LL(32)} \parallel X_{LR(32)} \parallel X_{RL(32)} \parallel X_{RR(32)}) \\
&\triangleq \text{io} \left( X_{LL(32)} \oplus \text{f64} ((X_{LL(32)} \oplus X_{LR(32)}) \parallel (X_{RL(32)} \oplus X_{RR(32)}), RK_{(128)})_{L(32)} \right) \parallel \\
&\quad \left( X_{LR(32)} \oplus \text{f64} ((X_{LL(32)} \oplus X_{LR(32)}) \parallel (X_{RL(32)} \oplus X_{RR(32)}), RK_{(128)})_{L(32)} \right) \parallel \\
&\quad \text{io} \left( X_{RL(32)} \oplus \text{f64} ((X_{LL(32)} \oplus X_{LR(32)}) \parallel (X_{RL(32)} \oplus X_{RR(32)}), RK_{(128)})_{R(32)} \right) \parallel \\
&\quad \left( X_{RR(32)} \oplus \text{f64} ((X_{LL(32)} \oplus X_{LR(32)}) \parallel (X_{RL(32)} \oplus X_{RR(32)}), RK_{(128)})_{R(32)} \right)
\end{aligned}$$

### 2.3.3 Definitions of `or` and `io`

The orthomorphism `or` is a function taking a 32-bit input  $X_{(32)} \triangleq X_{L(16)} \parallel X_{R(16)}$  and returning a 32-bit output  $Y_{(32)} \triangleq Y_{L(16)} \parallel Y_{R(16)}$ . It is defined as

$$Y_{L(16)} \parallel Y_{R(16)} = \text{or} (X_{L(16)} \parallel X_{R(16)}) \triangleq X_{R(16)} \parallel (X_{L(16)} \oplus X_{R(16)}) \quad (2.11)$$

`or` is in fact a one-round Feistel scheme with the identity function as round function. The inverse function of `or`, denoted `io`, is defined as

$$Y_{L(16)} \parallel Y_{R(16)} = \text{io} (X_{L(32)} \parallel X_{R(32)}) \triangleq (X_{L(16)} \oplus X_{R(16)}) \parallel X_{L(16)} \quad (2.12)$$

### 2.3.4 Definition of `f32`

The function `f32` builds the core of `FOX64/k/r`. It is built of three main parts: a *substitution* part, denoted `sigma4`, a *diffusion* part, denoted `mu4`, and a *round key addition* part (see Fig. 2.3).

Formally, the `f32` function takes a 32-bit input  $X_{(32)}$ , a 64-bit round key  $RK_{(64)} \triangleq RK_{0(32)} \parallel RK_{1(32)}$  and returns a 32-bit output  $Y_{(32)}$ . The `f32` function is then formally defined as

$$\begin{aligned}
Y_{(32)} &= \text{f32} (X_{(32)}, RK_{(64)}) \\
&\triangleq \text{sigma4}(\text{mu4}(\text{sigma4}(X_{(32)} \oplus RK_{0(32)})) \oplus RK_{1(32)}) \oplus RK_{0(32)}
\end{aligned}$$



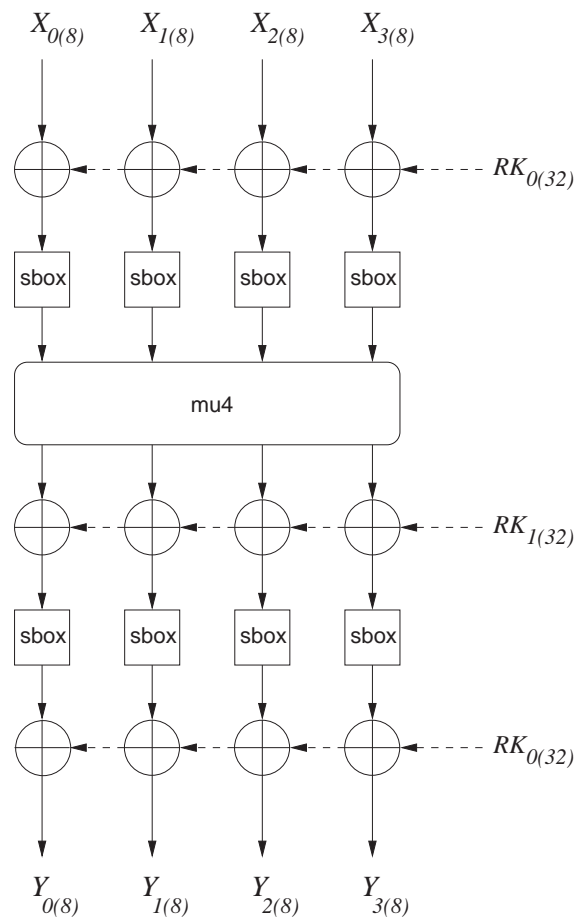


Figure 2.3: Function f32

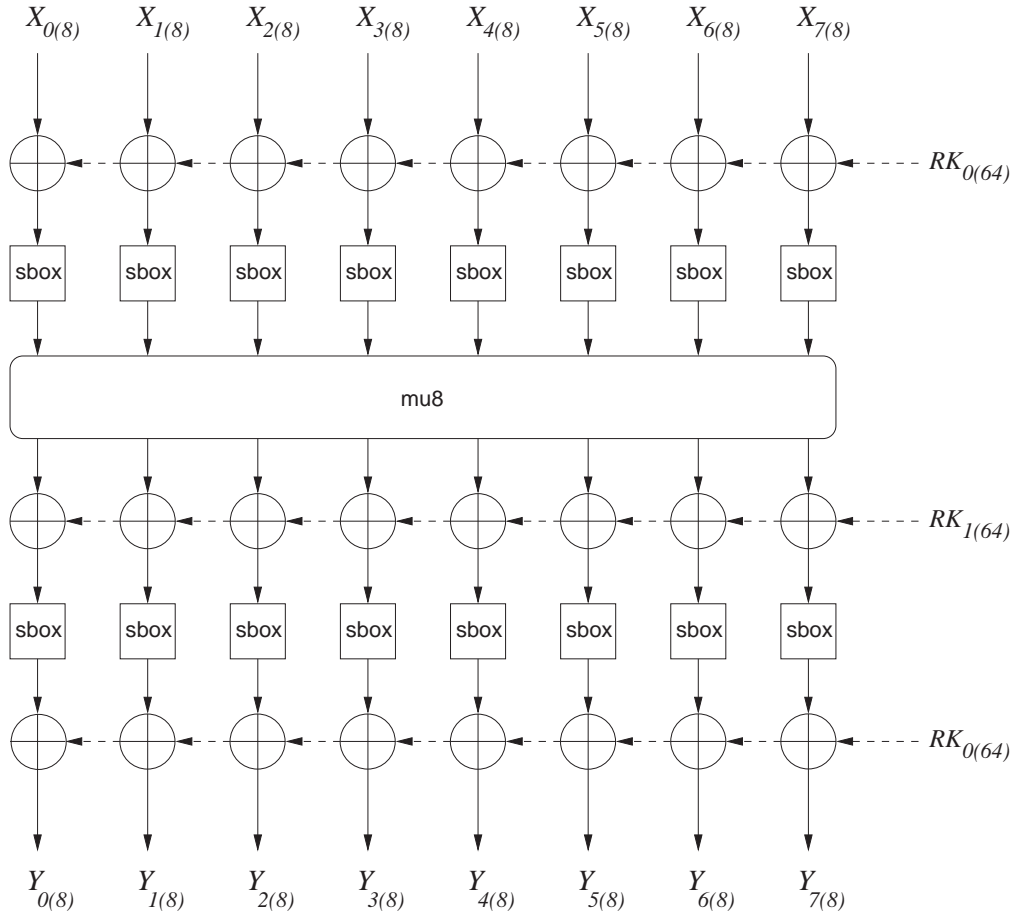


Figure 2.4: Function f64

### 2.3.5 Definition of f64

The function f64 builds the core of FOX128/ $k/r$ . It is built of three main parts: a *substitution* part, denoted *sigma8*, a *diffusion* part, denoted *mu8*, and a *round key addition* part (see Fig. 2.4).

Formally, the f64 function takes a 64-bit input  $X_{(64)}$ , a 128-bit round key  $RK_{(128)} \triangleq RK_{0(64)} || RK_{1(64)}$  and returns a 64-bit output  $Y_{(64)}$ . The f64 function is then defined as

$$\begin{aligned}
 Y_{(64)} &= \text{f64}(X_{(64)}, RK_{(128)}) \\
 &\triangleq \text{sigma8}(\text{mu8}(\text{sigma8}(X_{(64)} \oplus RK_{0(64)})) \oplus RK_{1(64)}) \oplus RK_{0(64)}
 \end{aligned}$$

### 2.3.6 Definition of sigma4, sigma8 and sbox

The function `sigma4` takes a 32-bit input  $X_{(32)} \triangleq X_{0(8)} || X_{1(8)} || X_{2(8)} || X_{3(8)}$  and returns a 32-bit output  $Y_{(32)}$ . It is defined as

$$\begin{aligned} Y_{(32)} &= \text{sigma4}(X_{0(8)} || X_{1(8)} || X_{2(8)} || X_{3(8)}) \\ &\triangleq \text{sbox}(X_{0(8)}) || \text{sbox}(X_{1(8)}) || \text{sbox}(X_{2(8)}) || \text{sbox}(X_{3(8)}) \end{aligned}$$

The function `sigma8` takes a 64-bit input

$$X_{(64)} \triangleq X_{0(8)} || X_{1(8)} || X_{2(8)} || X_{3(8)} || X_{4(8)} || X_{5(8)} || X_{6(8)} || X_{7(8)} \quad (2.13)$$

and returns a 64-bit output  $Y_{(64)}$ . It is defined as

$$\begin{aligned} Y_{(64)} &= \text{sigma8}(X_{0(8)} || X_{1(8)} || X_{2(8)} || X_{3(8)} || X_{4(8)} || X_{5(8)} || X_{6(8)} || X_{7(8)}) \\ &\triangleq \text{sbox}(X_{0(8)}) || \text{sbox}(X_{1(8)}) || \text{sbox}(X_{2(8)}) || \text{sbox}(X_{3(8)}) || \\ &\quad \text{sbox}(X_{4(8)}) || \text{sbox}(X_{5(8)}) || \text{sbox}(X_{6(8)}) || \text{sbox}(X_{7(8)}) \end{aligned}$$

Finally, the `sbox` function is the following lookup table:

sbox	0x.0	0x.1	0x.2	0x.3	0x.4	0x.5	0x.6	0x.7	0x.8	0x.9	0x.A	0x.B	0x.C	0x.D	0x.E	0x.F
0x0.	5D	DE	00	B7	D3	CA	3C	0D	C3	F8	CB	8D	76	89	AA	12
0x1.	88	22	4F	DB	6D	47	E4	4C	78	9A	49	93	C4	C0	86	13
0x2.	A9	20	53	1C	4E	CF	35	39	B4	A1	54	64	03	C7	85	5C
0x3.	5B	CD	D8	72	96	42	B8	E1	A2	60	EF	BD	02	AF	8C	73
0x4.	7C	7F	5E	F9	65	E6	EB	AD	5A	A5	79	8E	15	30	EC	A4
0x5.	C2	3E	E0	74	51	FB	2D	6E	94	4D	55	34	AE	52	7E	9D
0x6.	4A	F7	80	F0	D0	90	A7	E8	9F	50	D5	D1	98	CC	A0	17
0x7.	F4	B6	C1	28	5F	26	01	AB	25	38	82	7D	48	FC	1B	CE
0x8.	3F	6B	E2	67	66	43	59	19	84	3D	F5	2F	C9	BC	D9	95
0x9.	29	41	DA	1A	B0	E9	69	D2	7B	D7	11	9B	33	8A	23	09
0xA.	D4	71	44	68	6F	F2	0E	DF	87	DC	83	18	6A	EE	99	81
0xB.	62	36	2E	7A	FE	45	9C	75	91	0C	0F	E7	F6	14	63	1D
0xC.	0B	8B	B3	F3	B2	3B	08	4B	10	A6	32	B9	A8	92	F1	56
0xD.	DD	21	BF	04	BE	D6	FD	77	EA	3A	C8	8F	57	1E	FA	2B
0xE.	58	C5	27	AC	E3	ED	97	BB	46	05	40	31	E5	37	2C	9E
0xF.	0A	B1	B5	06	6C	1F	A3	2A	70	FF	BA	07	24	16	C6	61

One should read this table as follows: to compute `sbox(0x4C)`, one selects first the row named `0x4.` (*i.e.* the fifth row), and then one selects the column named `0x.C` (*i.e.* the thirteenth column) and we get finally

$$\text{sbox}(0x4C) = 0x15 \quad (2.14)$$

### 2.3.7 Definition of mu4

The diffusive part of `f32` is a linear  $(4, 4)$ -multipermutation defined on  $GF(2^8)$ . Formally, it is a function taking a 32-bit input  $X_{(32)} \triangleq X_{0(8)} || X_{1(8)} || X_{2(8)} || X_{3(8)}$  and returning a 32-bit output  $Y_{(32)} \triangleq Y_{0(8)} || Y_{1(8)} || Y_{2(8)} || Y_{3(8)}$ . `f32` is defined as

$$\begin{pmatrix} Y_{0(8)} \\ Y_{1(8)} \\ Y_{2(8)} \\ Y_{3(8)} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \alpha \\ 1 & c & \alpha & 1 \\ c & \alpha & 1 & 1 \\ \alpha & 1 & c & 1 \end{pmatrix} \times \begin{pmatrix} X_{0(8)} \\ X_{1(8)} \\ X_{2(8)} \\ X_{3(8)} \end{pmatrix} \quad (2.15)$$

where

$$c \triangleq \alpha^{-1} + 1 = \alpha^7 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + 1 \quad (2.16)$$

All the additions and multiplications are defined in  $\text{GF}(2^8)$  using the representation described in §1.6.

### 2.3.8 Definition of mu8

The diffusive part of f64 is a linear (8,8)-multipermutation defined on  $\text{GF}(2^8)$ . Formally, it is a function taking a 64-bit input

$$X_{(64)} \triangleq X_{0(8)} || X_{1(8)} || X_{2(8)} || X_{3(8)} || X_{4(8)} || X_{5(8)} || X_{6(8)} || X_{7(8)} \quad (2.17)$$

and returning a 64-bit output

$$Y_{(64)} \triangleq Y_{0(8)} || Y_{1(8)} || Y_{2(8)} || Y_{3(8)} || Y_{4(8)} || Y_{5(8)} || Y_{6(8)} || Y_{7(8)} \quad (2.18)$$

f64 is defined as

$$\begin{pmatrix} Y_{0(8)} \\ Y_{1(8)} \\ Y_{2(8)} \\ Y_{3(8)} \\ Y_{4(8)} \\ Y_{5(8)} \\ Y_{6(8)} \\ Y_{7(8)} \end{pmatrix} \triangleq \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & a \\ 1 & a & b & c & d & e & f & 1 \\ a & b & c & d & e & f & 1 & 1 \\ b & c & d & e & f & 1 & a & 1 \\ c & d & e & f & 1 & a & b & 1 \\ d & e & f & 1 & a & b & c & 1 \\ e & f & 1 & a & b & c & d & 1 \\ f & 1 & a & b & c & d & e & 1 \end{pmatrix} \times \begin{pmatrix} X_{0(8)} \\ X_{1(8)} \\ X_{2(8)} \\ X_{3(8)} \\ X_{4(8)} \\ X_{5(8)} \\ X_{6(8)} \\ X_{7(8)} \end{pmatrix} \quad (2.19)$$

where

$$\begin{aligned} a &\triangleq \alpha + 1 \\ b &\triangleq \alpha^7 + \alpha \\ c &\triangleq \alpha \\ d &\triangleq \alpha^2 \\ e &\triangleq \alpha^7 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 \\ f &\triangleq \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha \end{aligned}$$

All the additions and multiplications are defined in  $\text{GF}(2^8)$  using the representation described in §1.6.

## 2.4 Key Schedule

The key schedule is the algorithm which derives the subkey material

$$RK_{(r,64)} \triangleq RK_{0(64)} || RK_{1(64)} || \dots || RK_{r-1(64)} \quad (2.20)$$

and

$$RK_{(r,128)} \triangleq RK_{0(128)} || RK_{1(128)} || \dots || RK_{r-1(128)} \quad (2.21)$$

(for FOX64 and FOX128, respectively) out of the key  $K_{(k)}$ .

### 2.4.1 General Overview

A FOX key  $K_{(k)}$  must have a bit-length  $k$  such that  $0 \leq k \leq 256$ , and  $k$  must be a multiple of 8. Depending on the key length and the block size, a member of the FOX block cipher family may use one among three different key-schedule algorithm versions, denoted respectively KS64, KS64h and KS128. A constant,  $ek$ , depends on these values as well. The following table defines precisely the relation between the key size, the block size, the constant  $ek$  and the key-schedule algorithm version.

Cipher	Block size	Key size	Key-Schedule Version	$ek$
FOX64	64	$0 \leq k \leq 128$	KS64	128
FOX64	64	$136 \leq k \leq 256$	KS64h	256
FOX128	128	$0 \leq k \leq 256$	KS128	256

The three different versions of the key-schedule algorithm are constituted of four main parts: a padding part, denoted P, expanding  $K_{(k)}$  into  $ek$  bits, a mixing part, denoted M, a diversification part, denoted D, whose core consists mainly in a linear feedback shift register denoted LFSR, and finally, a non-linear part, denoted NLx (see Fig. 2.5 and Alg. 1 for a high-level overview of the key-schedule algorithm design).

As outlined above, the key-schedule algorithm definition depends on a the number of rounds  $r$ , on the key length  $k$  and on the cipher (FOX64 or FOX128). In fact, NLx is the only part which differs between the different versions, and we will denote the three variants NL64, NL64h and NL128.

### 2.4.2 Definition of KS64

This key-schedule algorithm is designed to be used by FOX64 with keys smaller or equal to 128 bits. It takes the following parameters as input: a key  $K$  of length  $k$  bits, with  $0 \leq k \leq 128$  and a number of rounds  $r$ . It returns in output  $r$  64-bit subkeys. KS64 is formally defined in Alg. 2.

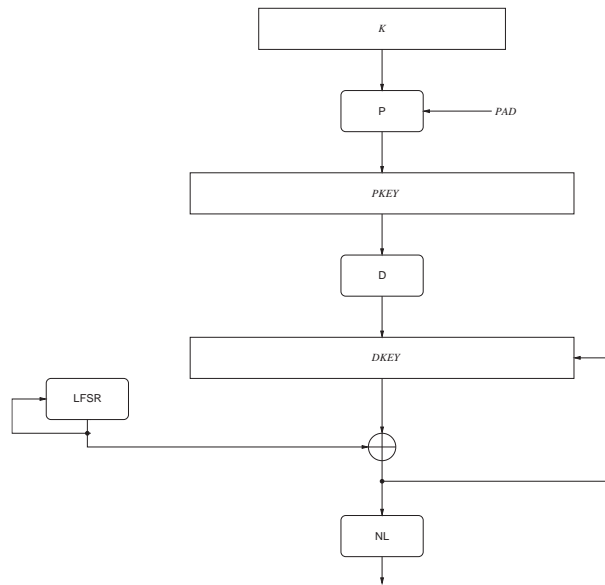


Figure 2.5: Key-Schedule Algorithm (High-Level Overview)

**Algorithm 1** Key-Schedule Algorithm (High-Level Description)

---

```

/* Preprocessing */
PKEY ← P(K)
MKEY ← M(PKEY)
/* Initialization of the loop */
i ← 1
/* Loop */
while i ≤ r do
  DKEY ← D(MKEY, i, r)
  Output RKi-1(x) ← NLx(DKEY)
  i ← i + 1
end while

```

---

**Algorithm 2** Key-Schedule Algorithm KS64

---

```

/* Preprocessing */
if  $k < ek$  then
   $PKEY \triangleq P(K)$ 
   $MKEY \triangleq M(PKEY)$ 
else
   $PKEY \triangleq K$ 
   $MKEY \triangleq PKEY$ 
end if
/* Initialization of the loop */
 $i \triangleq 1$ 
/* Loop */
while  $i \leq r$  do
   $DKEY \triangleq D(MKEY, i, r)$ 
  Output  $RK_{i-1(64)} \triangleq NL64(DKEY)$ 
   $i \triangleq i + 1$ 
end while

```

---

**2.4.3 Definition of KS64h**

This key schedule algorithm is designed to be used by FOX64 with keys larger than 128 bits. It takes the following parameters as input: a key  $K$  of length  $k$  bits, with  $136 \leq k \leq 256$  and a number of rounds  $r$ . It returns in output  $r$  64-bit subkeys. KS64h is formally defined in Alg. 3.

**2.4.4 Definition of KS128**

This key schedule algorithm is designed to be used by FOX128. It takes the following parameters as input: a key  $K$  of length  $k$  bits, with  $0 \leq k \leq 256$  and a number of rounds  $r$ . It returns in output  $r$  128-bit subkeys. KS128 is formally defined in Alg. 4.

**2.4.5 Definition of P**

The P-part, taking  $ek$  and  $k$  as input, is basically a function expanding a bit string by  $\frac{ek-k}{8}$  bytes. More precisely, then P concatenates the input key  $K$  with the first  $ek - k$  bits of the constant pad, giving  $PKEY$  as output. The P function is defined formally in Alg. 5. The pad constant value is defined in the following section.

---

**Algorithm 3** Key-Schedule Algorithm KS64h

---

```

/* Preprocessing */
if  $k < ek$  then
   $PKEY \triangleq P(K)$ 
   $MKEY \triangleq M(PKEY)$ 
else
   $PKEY \triangleq K$ 
   $MKEY \triangleq PKEY$ 
end if
/* Initialization of the loop */
 $i \triangleq 1$ 
/* Loop */
while  $i \leq r$  do
   $DKEY \triangleq D(MKEY, i, r)$ 
  Output  $RK_{i-1(64)} \triangleq NL64h(DKEY)$ 
   $i \triangleq i + 1$ 
end while

```

---



---

**Algorithm 4** Key-Schedule Algorithm KS128

---

```

/* Preprocessing */
if  $k < ek$  then
   $PKEY \triangleq P(K)$ 
   $MKEY \triangleq M(PKEY)$ 
else
   $PKEY \triangleq K$ 
   $MKEY \triangleq PKEY$ 
end if
/* Initialization of the loop */
 $i \triangleq 1$ 
/* Loop */
while  $i \leq r$  do
   $DKEY \triangleq D(MKEY, i, r)$ 
  Output  $RK_{i-1(128)} \triangleq NL128(DKEY)$ 
   $i \triangleq i + 1$ 
end while

```

---



---

**Algorithm 5** P-Part

---

```

Output  $PKEY \triangleq K || \text{pad}_{[0 \dots ek-k-1]}$ 

```

---



### 2.4.6 Definition of pad

The constant pad is defined as being the first 256 bits of the hexadecimal development of  $e - 2$ :

$$e - 2 = \sum_{n=0}^{+\infty} \frac{1}{n!} - 2$$

Thus,

$$\text{pad} \triangleq \text{0xB7E151628AED2A6ABF7158809CF4F3C762E7160F38B4DA56A784D9045190CFEF}$$

### 2.4.7 Definition of M

The M-part is used to mix the padded key  $PKEY$ , such that the constant words depend on the randomness provided by the key, too. This is done with help of a Fibonacci recursion. It takes as input a key  $PKEY$  with length  $ek$  (expressed in bits). More formally, the padded key  $PKEY$  is seen as an array of  $\frac{ek}{8}$  bytes  $PKEY_{i(8)}$ ,  $0 \leq i \leq \frac{ek}{8} - 1$ , and is mixed according to

$$MKEY_{i(8)} \triangleq PKEY_{i(8)} \oplus (MKEY_{i-1(8)} + MKEY_{i-2(8)} \bmod 2^8) \quad 0 \leq i \leq \frac{ek}{8} - 1 \quad (2.22)$$

with the convention that

$$MKEY_{-2(8)} \triangleq \text{0x6A} \quad \text{and} \quad MKEY_{-1(8)} \triangleq \text{0x76} \quad (2.23)$$

### 2.4.8 Definition of D

The D-part is a diversification part. It takes a key  $MKEY$  having a length in bits equal to  $ek$ , the total round number  $r$ , and the current round number  $i$ , with  $1 \leq i \leq r$ ; it modifies  $MKEY$  with help of the output of a 24-bit Linear Shift Feedback Register (LFSR) denoted LFSR. More precisely,  $MKEY$  is seen as an array of  $\lfloor \frac{ek}{24} \rfloor$  24-bit values  $MKEY_{j(24)}$ , with  $0 \leq j \leq \lfloor \frac{ek}{24} \rfloor - 1$  concatenated with one residue byte  $MKEYRB_{(8)}$  (if  $ek = 128$ ) or two residue bytes  $MKEYRB_{(16)}$  (if  $ek = 256$ ), and is modified according to

$$DKEY_{j(24)} \triangleq MKEY_{j(24)} \oplus \text{LFSR} \left( (i-1) \cdot \left\lceil \frac{ek}{24} \right\rceil + j, r \right) \quad \text{for } 0 \leq j \leq \left\lfloor \frac{ek}{24} \right\rfloor - 1$$

and the  $DKEYRB_{(8)}$  value ( $DKEYRB_{(16)}$ ) is obtained by XORing the most 8 (16) significant bits of  $\text{LFSR}((i-1) \cdot \lceil \frac{ek}{24} \rceil + \lfloor \frac{ek}{24} \rfloor, r)$  with  $MKEYRB_{(8)}$  ( $MKEYRB_{(16)}$ ), respectively. The remaining 16 (8) bits of the LFSR routine output are discarded.

### 2.4.9 Definition of LFSR

The diversification part D needs a stream of pseudo-random values; it is produced by a 24-bit linear feedback shift register, denoted LFSR. This algorithm takes two inputs, the total number of rounds  $r$  and a number of preliminary clocking  $c$ . It is based on the following primitive polynomial of degree 24 over GF(2):

$$\xi^{24} + \xi^4 + \xi^3 + \xi + 1 \quad (2.24)$$

The register is initially seeded with the value  $0x6A||r_{(8)}||\overline{r_{(8)}}$ , where  $r_{(8)}$  is expressed as an 8-bit value. LFSR is described formally in Alg. 6.

---

#### Algorithm 6 LFSR Algorithm

---

```

/* Initialization */
reg  $\triangleq$  0x6A||r|| $\bar{r}$ 
/* Pre-Clocking */
p  $\triangleq$  0
while p < c do
  p  $\triangleq$  p + 1
  if (reg  $\wedge$  0x800000)  $\neq$  0x000000 then
    reg  $\triangleq$  (reg  $\ll$  1)  $\oplus$  0x00001B
  else
    reg  $\triangleq$  (reg  $\ll$  1)
  end if
end while
Output reg

```

---

### 2.4.10 Definition of NL64

The NL64-part takes a single input: the 128-bit value  $DKEY$  corresponding to the current round.

Basically, the  $DKEY$  value passes through a substitution layer (made of four parallel  $\sigma_4$  functions), a diffusion layer (made of four parallel  $\mu_4$  functions) and a mixing layer called  $\text{mix}_{64}$ . Then, the constant  $\text{pad}_{[0\dots 127]}$  is XORed and the result is flipped if and only if  $k = ek$ . The result passes through a second substitution layer, it is hashed down to 64 bits and the resulting value is encrypted first with a  $\text{Imor}_{64}$  round function, where the subkey is the left half of the  $DKEY$  value and second by a  $\text{Imid}_{64}$  function, where the subkey is the right half of  $DKEY$ . The resulting value is defined to be the 64-bit round key. Fig. 2.6 illustrates the NL64 process and Alg. 7 describes it formally.

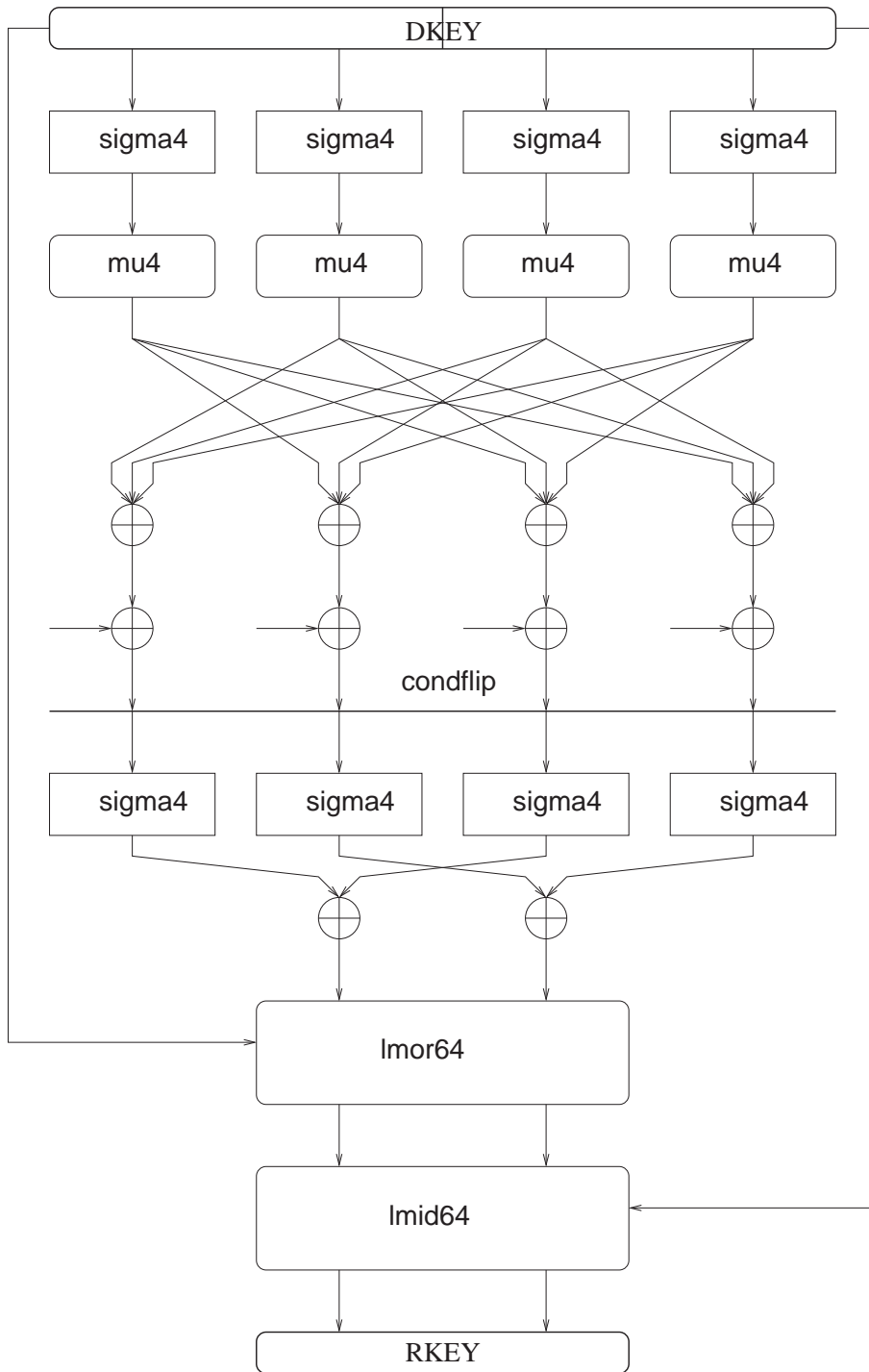


Figure 2.6: NL64 Part

**Algorithm 7** NL64 Part

---


$$t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} \triangleq DKEY$$

$$t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} \triangleq \text{sigma4}(t_{0(32)}) || \text{sigma4}(t_{1(32)}) || \text{sigma4}(t_{2(32)}) || \text{sigma4}(t_{3(32)})$$

$$t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} \triangleq \text{mu4}(t_{0(32)}) || \text{mu4}(t_{1(32)}) || \text{mu4}(t_{2(32)}) || \text{mu4}(t_{3(32)})$$

$$t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} \triangleq \text{mix64}(t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)})$$

$$t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} \triangleq (t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)}) \oplus \text{pad}_{[0..127]}$$
**if**  $k = ek$  **then**

$$t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} \triangleq \overline{t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)}}$$
**end if**

$$t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} \triangleq \text{sigma4}(t_{0(32)}) || \text{sigma4}(t_{1(32)}) || \text{sigma4}(t_{2(32)}) || \text{sigma4}(t_{3(32)})$$

$$t_{0(32)} || t_{1(32)} \triangleq (t_{0(32)} \oplus t_{2(32)}) || (t_{1(32)} \oplus t_{3(32)})$$

$$t_{0(32)} || t_{1(32)} \triangleq \text{lmor64}(t_{0(32)} || t_{1(32)}, DKEY_{[64..127]})$$

$$t_{0(32)} || t_{1(32)} \triangleq \text{lmid64}(t_{0(32)} || t_{1(32)}, DKEY_{[0..63]})$$
Output  $t_{0(32)} || t_{1(32)}$  as round subkey.

---

**2.4.11 Definition of NL64h**

The NL64h-part takes a single input: the 256-bit value *DKEY* corresponding to the current round.

Basically, the *DKEY* value passes through a substitution layer (made of eight parallel **sigma4** functions), a diffusion layer (made of eight parallel **mu4** functions) and a mixing layer called **mix64h**. Then, the constant **pad** is XORed and the result is flipped if and only if  $k = ek$ . The result passes through a second substitution layer, it is hashed down to 64 bits and the resulting value is encrypted first with three **lmor64** round functions, where the respective subkeys are the three left quarters of the *DKEY* value and secondly by a **lmid64** function, where the subkey is the rightmost quarter of *DKEY*. The resulting value is defined to be the 64-bit round key. Fig. 2.7 illustrates the NL64h process and Alg. 8 describes it formally.

**2.4.12 Definition of NL128**

The NL128-part takes a single different input: the 256-bit value *DKEY* corresponding to the current round.

Basically, the *DKEY* value passes through a substitution layer (made of four parallel **sigma8** functions), a diffusion layer (made of four parallel **mu8** functions) and a mixing layer called **mix128**. Then, the constant **pad** is XORed and the result is flipped if and only if  $k = ek$ . The result passes through a second substitution layer, it is hashed down to 128 bits and the resulting value is encrypted first with a **elmor128** round function, where the subkey is the left half of the *DKEY* value and second by a **elmid128** function, where the subkey is the right half of *DKEY*.

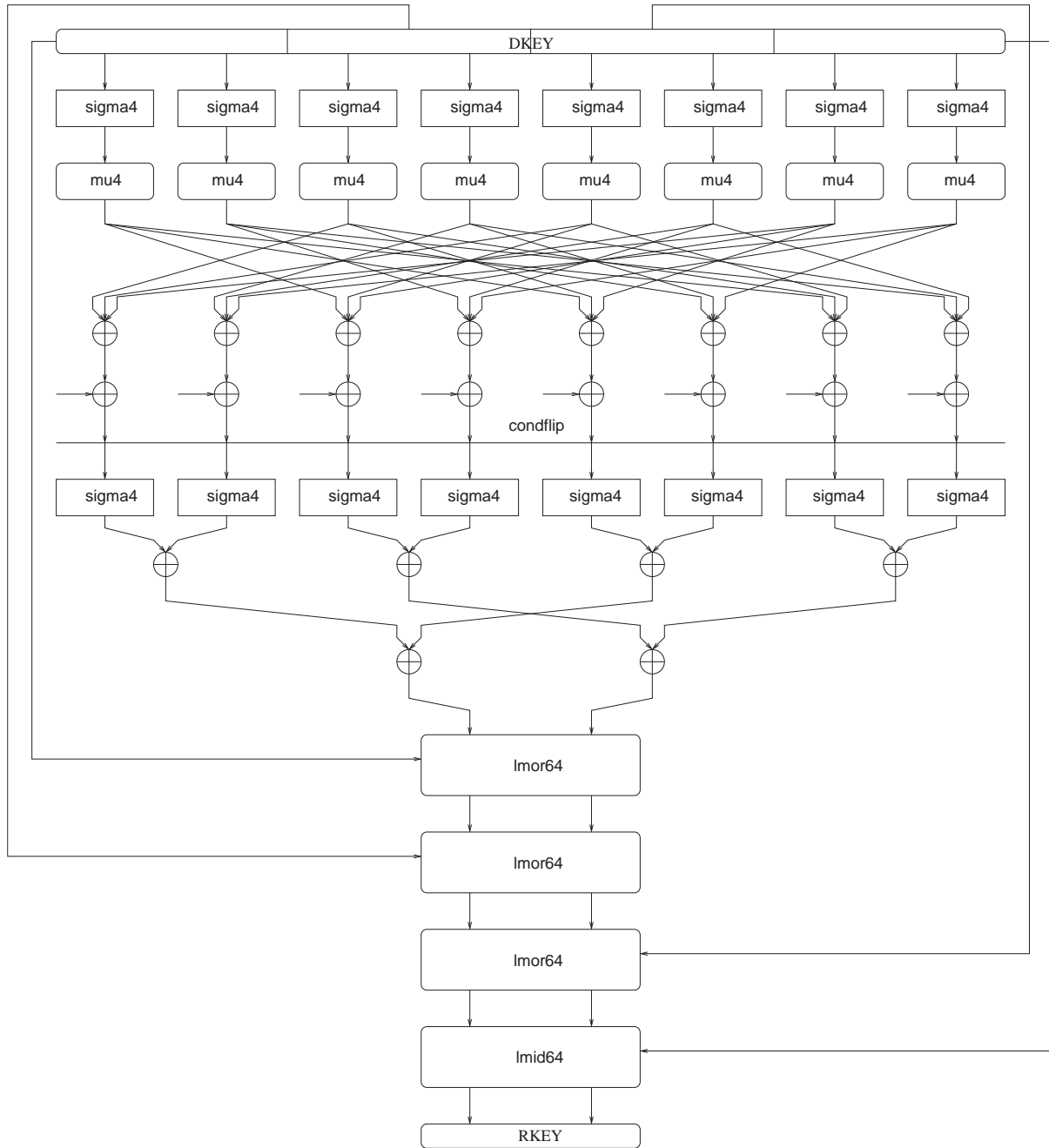


Figure 2.7: NL64h Part

**Algorithm 8 NL64h Part**


---

```

/* Initialization */
 $t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} || t_{4(32)} || t_{5(32)} || t_{6(32)} || t_{7(32)} \triangleq DKKEY$ 
/* Substitution Layer */
 $t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} \triangleq \text{sigma4}(t_{0(32)}) || \text{sigma4}(t_{1(32)}) || \text{sigma4}(t_{2(32)}) || \text{sigma4}(t_{3(32)})$ 
 $t_{4(32)} || t_{5(32)} || t_{6(32)} || t_{7(32)} \triangleq \text{sigma4}(t_{4(32)}) || \text{sigma4}(t_{5(32)}) || \text{sigma4}(t_{6(32)}) || \text{sigma4}(t_{7(32)})$ 
/* Diffusion Layer */
 $t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} \triangleq \text{mu4}(t_{0(32)}) || \text{mu4}(t_{1(32)}) || \text{mu4}(t_{2(32)}) || \text{mu4}(t_{3(32)})$ 
 $t_{4(32)} || t_{5(32)} || t_{6(32)} || t_{7(32)} \triangleq \text{mu4}(t_{4(32)}) || \text{mu4}(t_{5(32)}) || \text{mu4}(t_{6(32)}) || \text{mu4}(t_{7(32)})$ 
 $t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} || t_{4(32)} || t_{5(32)} || t_{6(32)} || t_{7(32)} \triangleq \text{mix64h}(t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} || t_{4(32)} || t_{5(32)} || t_{6(32)} || t_{7(32)})$ 
 $t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} || t_{4(32)} || t_{5(32)} || t_{6(32)} || t_{7(32)} \triangleq (t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} || t_{4(32)} || t_{5(32)} || t_{6(32)} || t_{7(32)}) \oplus \text{pad}$ 
if  $k = ek$  then
     $t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} || t_{4(32)} || t_{5(32)} || t_{6(32)} || t_{7(32)} \triangleq \overline{t_{0(32)}} || \overline{t_{1(32)}} || \overline{t_{2(32)}} || \overline{t_{3(32)}} || \overline{t_{4(32)}} || \overline{t_{5(32)}} || \overline{t_{6(32)}} || \overline{t_{7(32)}}$ 
end if
/* Substitution Layer */
 $t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} \triangleq \text{sigma4}(t_{0(32)}) || \text{sigma4}(t_{1(32)}) || \text{sigma4}(t_{2(32)}) || \text{sigma4}(t_{3(32)})$ 
 $t_{4(32)} || t_{5(32)} || t_{6(32)} || t_{7(32)} \triangleq \text{sigma4}(t_{4(32)}) || \text{sigma4}(t_{5(32)}) || \text{sigma4}(t_{6(32)}) || \text{sigma4}(t_{7(32)})$ 
/* Hashing Layer */
 $t_{0(32)} || t_{1(32)} || t_{2(32)} || t_{3(32)} \triangleq (t_{0(32)} \oplus t_{1(32)}) || (t_{2(32)} \oplus t_{3(32)}) || (t_{4(32)} \oplus t_{5(32)}) || (t_{6(32)} \oplus t_{7(32)})$ 
 $t_{0(32)} || t_{1(32)} \triangleq (t_{0(32)} \oplus t_{2(32)}) || (t_{1(32)} \oplus t_{3(32)})$ 
/* Encryption Layer */
 $t_{0(32)} || t_{1(32)} \triangleq \text{Imor64}(t_{0(32)} || t_{1(32)}, DKKEY_{[0...63]})$ 
 $t_{0(32)} || t_{1(32)} \triangleq \text{Imor64}(t_{0(32)} || t_{1(32)}, DKKEY_{[64...127]})$ 
 $t_{0(32)} || t_{1(32)} \triangleq \text{Imor64}(t_{0(32)} || t_{1(32)}, DKKEY_{[128...191]})$ 
 $t_{0(32)} || t_{1(32)} \triangleq \text{Imid64}(t_{0(32)} || t_{1(32)}, DKKEY_{[192...256]})$ 
Output  $t_{0(32)} || t_{1(32)}$  as round subkey.

```

---

The resulting value is defined to be the 128-bit round key. Fig. 2.8 illustrates the NL128 process and Alg. 9 describes it formally.

---

**Algorithm 9** NL128 Part
 

---

```

 $t_{0(64)} || t_{1(64)} || t_{2(64)} || t_{3(64)} \triangleq DKKEY$ 
 $t_{0(64)} || t_{1(64)} || t_{2(64)} || t_{3(64)} \triangleq \text{sigma8}(t_{0(64)}) || \text{sigma8}(t_{1(64)}) || \text{sigma8}(t_{2(64)}) || \text{sigma8}(t_{3(64)})$ 
 $t_{0(64)} || t_{1(64)} || t_{2(64)} || t_{3(64)} \triangleq \text{mu8}(t_{0(64)}) || \text{mu8}(t_{1(64)}) || \text{mu8}(t_{2(64)}) || \text{mu8}(t_{3(64)})$ 
 $t_{0(64)} || t_{1(64)} || t_{2(64)} || t_{3(64)} \triangleq \text{mix128}(t_{0(64)} || t_{1(64)} || t_{2(64)} || t_{3(64)})$ 
 $t_{0(64)} || t_{1(64)} || t_{2(64)} || t_{3(64)} \triangleq (t_{0(64)} || t_{1(64)} || t_{2(64)} || t_{3(64)}) \oplus \text{pad}$ 
if  $k = ek$  then
   $t_{0(64)} || t_{1(64)} || t_{2(64)} || t_{3(64)} \triangleq \overline{t_{0(64)}} || \overline{t_{1(64)}} || \overline{t_{2(64)}} || \overline{t_{3(64)}}$ 
end if
 $t_{0(64)} || t_{1(64)} || t_{2(64)} || t_{3(64)} \triangleq \text{sigma8}(t_{0(64)}) || \text{sigma8}(t_{1(64)}) || \text{sigma8}(t_{2(64)}) || \text{sigma8}(t_{3(64)})$ 
 $t_{0(64)} || t_{1(64)} \triangleq (t_{0(64)} \oplus t_{2(64)}) || (t_{1(64)} \oplus t_{3(64)})$ 
 $t_{0(64)} || t_{1(64)} \triangleq \text{elmor128}(t_{0(64)} || t_{1(64)}, DKKEY_{[128...255]})$ 
 $t_{0(64)} || t_{1(64)} \triangleq \text{elmid128}(t_{0(64)} || t_{1(64)}, DKKEY_{[0...127]})$ 
Output  $t_{0(64)} || t_{1(64)}$  as round subkey.

```

---

### 2.4.13 Definition of mix64

Given an input vector of four 32-bit values, denoted

$$X \triangleq X_{0(32)} || X_{1(32)} || X_{2(32)} || X_{3(32)} \quad (2.25)$$

the mix64 function consists in processing it by the following relations, resulting in an output vector denoted  $Y \triangleq Y_{0(32)} || Y_{1(32)} || Y_{2(32)} || Y_{3(32)}$ . More formally, mix64 is defined as

$$\begin{aligned} Y_{0(32)} &\triangleq X_{1(32)} \oplus X_{2(32)} \oplus X_{3(32)} \\ Y_{1(32)} &\triangleq X_{0(32)} \oplus X_{2(32)} \oplus X_{3(32)} \\ Y_{2(32)} &\triangleq X_{0(32)} \oplus X_{1(32)} \oplus X_{3(32)} \\ Y_{3(32)} &\triangleq X_{0(32)} \oplus X_{1(32)} \oplus X_{2(32)} \end{aligned}$$

### 2.4.14 Definition of mix64h

Given an input vector of eight 32-bit values, denoted

$$X \triangleq X_{0(32)} || X_{1(32)} || X_{2(32)} || X_{3(32)} || X_{4(32)} || X_{5(32)} || X_{6(32)} || X_{7(32)} \quad (2.26)$$

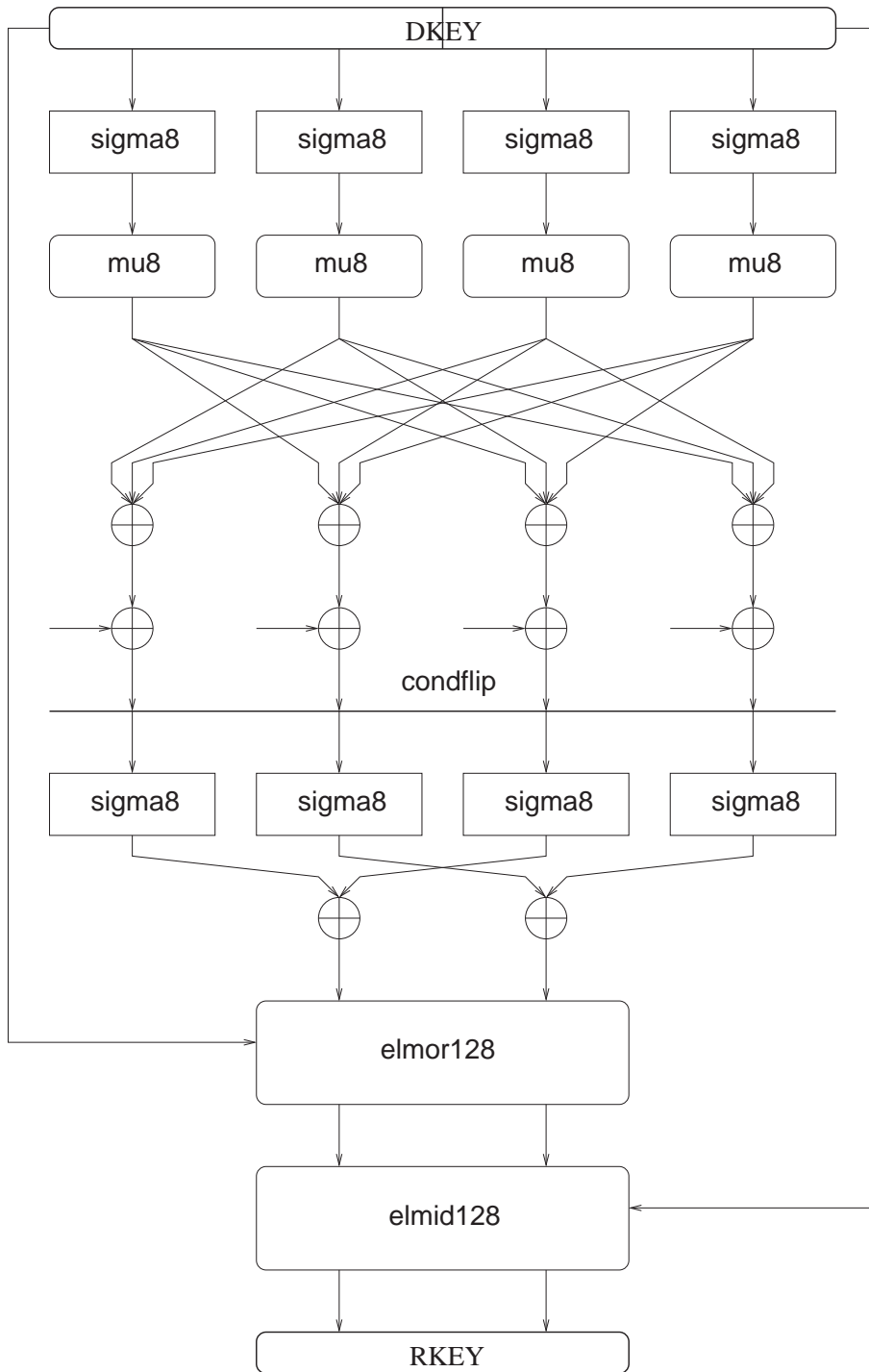


Figure 2.8: NL128 Part



the `mix64h` function consists in processing it by the following relations, resulting in an output vector denoted  $Y \triangleq Y_{0(32)} || Y_{1(32)} || Y_{2(32)} || Y_{3(32)} || Y_{4(32)} || Y_{5(32)} || Y_{6(32)} || Y_{7(32)}$ . More formally, `mix64h` is defined as

$$\begin{aligned}
 Y_{0(32)} &\triangleq X_{2(32)} \oplus X_{4(32)} \oplus X_{6(32)} \\
 Y_{1(32)} &\triangleq X_{3(32)} \oplus X_{5(32)} \oplus X_{7(32)} \\
 Y_{2(32)} &\triangleq X_{0(32)} \oplus X_{4(32)} \oplus X_{6(32)} \\
 Y_{3(32)} &\triangleq X_{1(32)} \oplus X_{5(32)} \oplus X_{7(32)} \\
 Y_{4(32)} &\triangleq X_{0(32)} \oplus X_{2(32)} \oplus X_{6(32)} \\
 Y_{5(32)} &\triangleq X_{1(32)} \oplus X_{3(32)} \oplus X_{7(32)} \\
 Y_{6(32)} &\triangleq X_{0(32)} \oplus X_{2(32)} \oplus X_{4(32)} \\
 Y_{7(32)} &\triangleq X_{1(32)} \oplus X_{3(32)} \oplus X_{5(32)}
 \end{aligned}$$

#### 2.4.15 Definition of `mix128`

Given an input vector of four 64-bit values, denoted  $X \triangleq X_{0(64)} || X_{1(64)} || X_{2(64)} || X_{3(64)}$ , the `mix64` function consists in processing it by the following relations, resulting in an output vector denoted  $Y \triangleq Y_{0(64)} || Y_{1(64)} || Y_{2(64)} || Y_{3(64)}$ . More formally, `mix128` is defined as

$$\begin{aligned}
 Y_{0(64)} &\triangleq X_{1(64)} \oplus X_{2(64)} \oplus X_{3(64)} \\
 Y_{1(64)} &\triangleq X_{0(64)} \oplus X_{2(64)} \oplus X_{3(64)} \\
 Y_{2(64)} &\triangleq X_{0(64)} \oplus X_{1(64)} \oplus X_{3(64)} \\
 Y_{3(64)} &\triangleq X_{0(64)} \oplus X_{1(64)} \oplus X_{2(64)}
 \end{aligned}$$

# Chapter 3

## Design Rationales

In this chapter, some explanations and justifications about the choices made during the design of FOX are provided. In §3.1, we recall some results of Vaudenay which provide a strong theoretical security foundation to both Lai-Massey and Extended Lai-Massey schemes as defined in §2.1 and §2.2, respectively. In §3.2, we describe how the non-linear part `sbox` has been designed, and its security characteristics; the diffusion part, *i.e.* the `mu4` and `mu8` functions, are described in §3.3. Finally, in §3.4, we provide some motivations in the context of the key-schedule algorithm.

### 3.1 Skeletons

#### 3.1.1 Lai-Massey Scheme

We recall briefly in this section results of Vaudenay [Vau00b] about the Lai-Massey scheme.

#### Pseudo-randomness and Super Pseudo-Randomness

A distinguisher  $\mathcal{A}$  is a probabilistic Turing machine with unlimited computation power. It has access to an oracle  $\mathcal{O}$  and can send it a *limited* number of queries. At the end, the distinguisher must output “0” or “1”. The advantage for distinguishing a random function  $f$  from a random function  $g$  is defined by

$$\text{Adv}(f, g) \triangleq \left| \Pr \left[ \mathcal{A}^{\mathcal{O}=f} = 1 \right] - \Pr \left[ \mathcal{A}^{\mathcal{O}=g} = 1 \right] \right| \quad (3.1)$$

From this point, we will make use of the following notation: given an orthomorphism  $\circ$  on a group  $(G, +)$  and given  $r$  functions  $f_1, f_2, \dots, f_r$  on  $G$ , we note a  $r$ -rounds Lai-Massey scheme using the  $r$  functions and the orthomorphism as defined in §2.1 by

$$\Lambda^\circ(f_1, \dots, f_r) \quad (3.2)$$

Then the following result is a Luby-Rackoff-like [LR88] result on the Lai-Massey scheme. We refer to [Vau00b] for a complete proof.

**Theorem 1** *Let  $f_1^*$ ,  $f_2^*$  and  $f_3^*$  be three independent random functions uniformly distributed on a group  $(G, +)$ . Let  $\circ$  be an orthomorphism<sup>1</sup> on  $G$ . For any distinguisher limited to  $d$  chosen plaintexts, where  $g = |G|$  denotes the cardinality of the group, between  $\Lambda^\circ(f_1^*, \dots, f_3^*)$  and a uniformly distributed random permutation  $c^*$ , we have*

$$\text{Adv}(\Lambda^\circ(f_1^*, \dots, f_3^*), c^*) \leq d(d-1)(g^{-1} + g^{-2}) \quad (3.3)$$

Super-pseudorandomness corresponds to cases where distinguishers can query chosen ciphertexts as well. In this scenario, the following result [Vau00b] asserts that rounds can provide super-pseudorandomness in a Lai-Massey scheme.

**Theorem 2** *Let  $f_1^*$ ,  $f_2^*$ ,  $f_3^*$  and  $f_4^*$  be four independent random functions uniformly distributed on a group  $(G, +)$ . Let  $\circ$  be an orthomorphism on  $G$ . For any distinguisher limited to  $d$  chosen plaintexts or ciphertexts, where  $g = |G|$  denotes the cardinality of the group, between  $\Lambda^\circ(f_1^*, \dots, f_r^*)$  and a uniformly distributed random permutation  $c^*$ , we have*

$$\text{Adv}(\Lambda^\circ(f_1^*, \dots, f_4^*), c^*) \leq d(d-1)(g^{-1} + g^{-2}) \quad (3.4)$$

In summary, the two previous theorems show that the Lai-Massey scheme, as defined in §2.1 provides pseudo-randomness on three rounds and super-pseudorandomness on four rounds, like for the Feistel scheme [Fei73].

### Decorrelation Inheritance

We recall here briefly some notions related to the concept of decorrelation (see [Vau03] for more details). Given an integer  $d$  and a random function  $f$  from a given set  $\mathcal{M}_1$  to a given set  $\mathcal{M}_2$ , one defines the  $d$ -wise distribution matrix  $[f]^d$  as a matrix in  $\mathbb{R}^{\mathcal{M}_1^d \times \mathcal{M}_2^d}$  by  $[f]_{(x_1, \dots, x_d)(y_1, \dots, y_d)}^d \triangleq \Pr[f(x_1) = y_1, \dots, f(x_d) = y_d]$ . For a matrix  $A$  in  $\mathbb{R}^{\mathcal{M}_1^d \times \mathcal{M}_2^d}$ , one defines the following matrix norm:

$$\|A\|_a \triangleq \max_{x_1} \sum_{y_1} \max_{x_2} \sum_{y_2} \cdots \max_{x_d} \sum_{y_d} |A_{(x_1, \dots, x_d)(y_1, \dots, y_d)}| \quad (3.5)$$

Finally, one defines

$$\text{Dec}_{\|\cdot\|_a}^d(c) \triangleq \| [c]^d - [c^*]^d \|_a \quad (3.6)$$

where  $c^*$  is a uniformly distributed random permutation. We know that  $\text{Adv}(c, c^*) \leq \frac{1}{2} \text{Dec}_{\|\cdot\|_a}^d$  for any distinguisher limited to  $d$  chosen plaintexts.

As pointed out in [Vau00b], it is possible to use results of [Vau00a] for showing that the decorrelation bias of the round functions of a Lai-Massey scheme is inherited by the whole structure. This is stated formally by the following theorem:

<sup>1</sup>An orthomorphism on a group  $(G, +)$  is a permutation  $\circ(x)$  such that  $\circ(x) - x$  is also a permutation.

**Theorem 3** *If  $f_1, \dots, f_r$  are  $r \geq 3$  independent random functions on a group  $(G, +)$  of order  $g$  such that  $\|f_i - f_i^*\|_a \leq \epsilon$  for  $1 \leq i \leq r$  and if  $\circ$  is an orthomorphism on  $G$ , we have*

$$\text{Dec}_{\|\cdot\|_a}^d(\Lambda^\circ(f_1, \dots, f_r)) \leq (3\epsilon + d(d-1)(2g^{-1} + g^{-2})) \lfloor \frac{r}{3} \rfloor \quad (3.7)$$

This result shows that provided the  $f_i$ 's are strong, so is the Lai-Massey scheme.

### 3.1.2 Extended Lai-Massey Scheme

One can show that an extended Lai-Massey scheme can be seen as a (classical) Lai-Massey scheme, and thus inherits all the “good properties” of a Lai-Massey scheme, as discussed in the previous section. For this, one needs the following result.

#### Lemma 1

*The application defined by*

$$\begin{cases} (\{0, 1\}^{32})^2 & \rightarrow (\{0, 1\}^{32})^2 \\ (x, y) & \mapsto (\text{or}(x), \text{or}(y)) \end{cases}$$

*is an orthomorphism, where  $\text{or}(\cdot)$  is the orthomorphism defined in Section 2.3.3.*

*Proof:* First, we show that this application is a permutation. This follows from the fact that the inverse application is given by

$$(x', y') \mapsto (\text{io}(x'), \text{io}(y'))$$

and that  $\text{io}$  is a permutation, too. Now, we have to check that

$$(x, y) \mapsto (\text{or}(x) \oplus x, \text{or}(y) \oplus y) \quad (3.8)$$

is also a permutation. This follows easily from the fact that (3.8) is an invertible application.

◇

From this lemma, we can deduce that an Extended Lai-Massey scheme is nothing but a Lai-Massey scheme. This is illustrated in Fig. 3.1.

### Resistance Towards Linear and Differential Cryptanalysis

We prove now some interesting properties of an Extended Lai-Massey scheme regarding differential and linear characteristics.

**Theorem 4** *In the Extended Lai-Massey scheme as defined in §2.2, any differential characteristic on two rounds must involve at least one f64-function.*

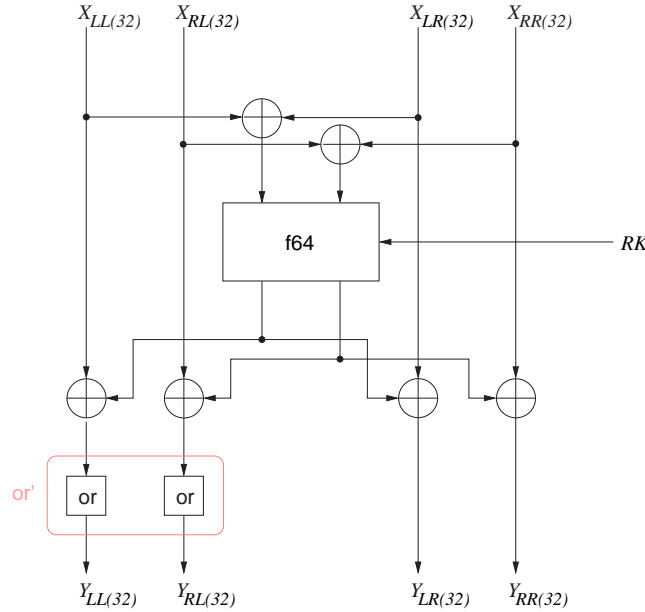


Figure 3.1: An alternative view of an Extended Lai-Massey scheme

*Proof:* We follow a top-down approach. If we stack up two rounds of an Extended Lai-Massey scheme (see Fig. 3.2 for a detailed illustration of one round) and we force a differential characteristic at the input of the first **f64**-function to be equal to 0, then a differential characteristic at the input of the two rounds must have the form  $(a, b, a, b, c, d, c, d)$  with  $a, b, c, d \in \{0, 1\}^{16}$  and  $a, b, c, d$  are not all equal to 0. At the end of the first round, the differential characteristic sounds  $(b, a \oplus b, a, b, d, c \oplus d, c, d)$ . At the input of the second **f64**-function, the differential characteristic is equal to  $(a \oplus b, a, c \oplus d, c)$ . We proceed by contraposition. If the input of the second **f64**-function is equal to zero, we have  $a = c = 0$ . As  $a \oplus b$  and  $c \oplus d$  must be both equal to 0, the we conclude that  $a = b = c = d = 0$ . This is a contradiction to our primary assumption about  $a, b, c$  and  $d$ , and the theorem follows.  $\diamond$

**Theorem 5** *In the Extended Lai-Massey scheme as defined in Fig. 2.2, any linear characteristic on two rounds must involve at least one **f64**-function.*

*Proof:* We follow a bottom-up approach. By forcing a linear characteristic to be equal to  $(0, 0, 0, 0, 0, 0, 0, 0)$  at the end of the second **f64**-function, we note that the output linear characteristic must have the form  $(a, a \oplus d, a \oplus d, d, b, b \oplus c, b \oplus c, c)$  with  $a, b, c, d \in \{0, 1\}^{16}$  and  $a, b, c, d$  not all equal to 0. If we consider now the first **f64**-function, we note that a linear characteristic at its output must have the form  $(d, a \oplus d, b, b \oplus c)$ , which implies that  $a = b = 0$  and then that

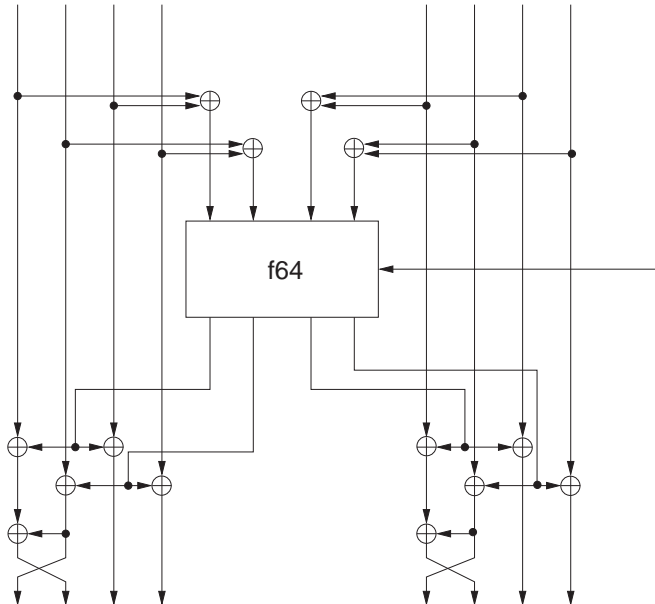


Figure 3.2: Extended Lay-Massey Scheme (detailed description)

$x$	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
$S_1(x)$	0x2	0x5	0x1	0x9	0xE	0xA	0xC	0x8	0x6	0x4	0x7	0xF	0xD	0xB	0x0	0x3
$S_2(x)$	0xB	0x4	0x1	0xF	0x0	0x3	0xE	0xD	0xA	0x8	0x7	0x5	0xC	0x2	0x9	0x6
$S_3(x)$	0xD	0xA	0xB	0x1	0x4	0x3	0x8	0x9	0x5	0x7	0x2	0xC	0xF	0x0	0x6	0xE

Figure 3.3: Small S-boxes  $S_1$ ,  $S_2$  and  $S_3$

$c = d = 0$ , which is a contradiction to our assumption. The theorem follows. ◇

### 3.2 sbx Transformation

In this section, we describe the motivations behind the definition of the S-box `sbx` and some of its security properties.

As defined in §2.3.6, `sbx` is a permutation mapping 8-bit inputs to 8-bit outputs. It consists in a Lai-Massey scheme with 3 rounds taking three different substitution boxes as round function; these “small” S-boxes are denoted  $S_1$ ,  $S_2$  and  $S_3$ , and their content is given in Fig. 3.3. The structure of `sbx` is depicted in Fig. 3.4. The orthomorphism `or4` used in the Lai-Massey scheme is a single round of a 4-bit Feistel scheme with the identity function as round function (see Fig. 3.5).

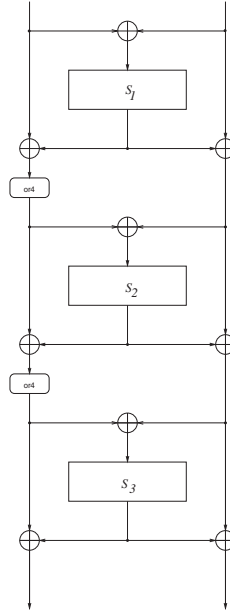


Figure 3.4: Structure of sbox

We describe now the generation process of the **sbox** transformation. First a set of three different candidates for small substitution boxes, each having a  $LP_{\max}$  and a  $DP_{\max}$  smaller than  $2^{-2}$  where pseudo-randomly chosen, with the common notations

$$DP^{\text{sbox}}(a, b) \triangleq \Pr[\text{sbox}(X \oplus a) = \text{sbox}(X) \oplus b] \quad (3.9)$$

$$LP^{\text{sbox}}(\mathbf{a}, \mathbf{b}) \triangleq (2 \cdot \Pr[\mathbf{a} \cdot X = \mathbf{b} \cdot \text{sbox}(X)] - 1)^2 \quad (3.10)$$

with  $\cdot$  being the inner dot-product on  $GF(2)^n$  and where

$$DP_{\max}^{\text{sbox}} \triangleq \max_{a \neq 0, b} DP^{\text{sbox}}(a, b) \quad (3.11)$$

$$LP_{\max}^{\text{sbox}} \triangleq \max_{\mathbf{a}, \mathbf{b} \neq \mathbf{0}} LP^{\text{sbox}}(\mathbf{a}, \mathbf{b}) \quad (3.12)$$

Then, the candidate **sbox** mapping was evaluated and tested regarding its  $LP_{\max}$  and  $DP_{\max}$  values, which are required to be smaller or equal to  $2^{-4}$ . This process has been iterated until a good candidate was found.

The chosen **sbox** satisfy

$$DP_{\max}^{\text{sbox}} = LP_{\max}^{\text{sbox}} = 2^{-4} \quad (3.13)$$

and its algebraic degree is equal to 6.

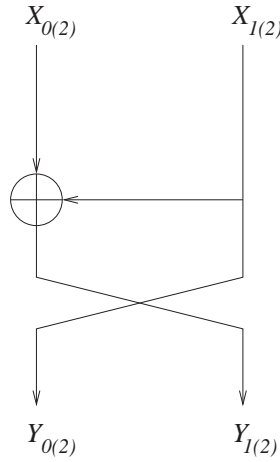


Figure 3.5: The orthomorphism or4

### 3.3 mu4 and mu8 Transformations

In this section, we describe the rationales behind the main diffusion parts of both algorithms, namely the mu4 and mu8 functions.

#### 3.3.1 Linear Multipermutations

Both mu4 and mu8 are *linear multipermutations*. This kind of construction was early recognized as being optimal for which regards its diffusion properties [SV95].

A linear application defined by a matrix  $A$  is a multipermutation if and only if  $\det(A) \neq 0$  and if the determinant of each submatrix of  $A$  is different of zero as well. It is well-known that one can construct linear multipermutations out of MDS linear codes (*i.e.* Maximum Distance Separable codes). Although this approach leads to very elegant constructions, they are not all very efficient to implement, especially on low-end smartcard, which have usually very few available memory and computational power.

Both mu4 and mu8 have the structure of the following matrix:

$$\begin{pmatrix} 1 & 1 & \dots & 1 & \xi \\ 1 & \zeta_1 & \dots & \zeta_n & 1 \\ \zeta_1 & \zeta_2 & \dots & 1 & 1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \zeta_{n-1} & \zeta_n & \dots & \zeta_{n-2} & 1 \\ \zeta_n & 1 & \dots & \zeta_{n-1} & 1 \end{pmatrix} \tag{3.14}$$

where  $\xi \in \{\zeta_i : 1 \leq i \leq n\}$ . In order to be efficiently implementable, the elements of the matrix, which are elements of  $\text{GF}(2^8)$ , should have a low Hamming weight, since the only really efficient



operations are the addition, the multiplication by  $\alpha$  and the division by  $\alpha$ : indeed, one should note that

$$\begin{aligned}\alpha^7 + \alpha &= \alpha^{-1} + \alpha^{-2} \\ \alpha^7 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 &= \alpha^{-1} \\ \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + \alpha &= \alpha^{-2}\end{aligned}$$

Thus, a search among possible combinations of “acceptable” elements  $\{\zeta_i : 1 \leq i \leq n\}$  has been done, until good matrices were found.

### 3.3.2 Security properties of f32 and f64

In this part, we prove some important results about the security of both f32 and f64 functions towards linear and differential cryptanalysis. As these functions may be viewed as classical *Substitution-Permutation Network* constructions, we will refer to some well-known results on their resistance towards linear and differential cryptanalysis proved in [HLL<sup>+</sup>01].

#### Differential Cryptanalysis

In the following, we denote  $\delta \triangleq \delta_1\delta_2\delta_3\delta_4$ ,  $d \triangleq d_1d_2d_3d_4$ ,  $D \triangleq D_1D_2D_3D_4$  and  $\Delta \triangleq \Delta_1\Delta_2\Delta_3\Delta_4$  a differential at input, after the first layer of S-boxes, after the mu4 application and at output, respectively (see Fig. 3.6).

Then the probability of this differential is given by

$$\text{DP}^{\text{f32}}(\delta, \Delta) = \sum_{\delta=\delta_1, \dots, \delta_n} \left( \prod_i \text{DP}^{\text{sbox}}(\delta_i, d_i) \prod_i \text{DP}^{\text{sbox}}(D_i, \Delta_i | \delta) \right) \quad (3.15)$$

Note that the value of (3.15), provided the round keys are uniformly distributed, does not depend on their values. As the mu4 construction is a (4, 4)-multipermutation, one is ensured that at last  $n_d = 5$  S-boxes before and after mu4 will be active. Then, by Theorem 1 of [HLL<sup>+</sup>01], we have

#### Theorem 6

$$\text{DP}_{\max}^{\text{f32}} \leq \left( \text{DP}_{\max}^{\text{sbox}} \right)^4 \quad (3.16)$$

We can extend these computations to the f64-function (as defined in Fig. 2.4), noting that  $n_d = 9$ , in this case, and we get

#### Theorem 7

$$\text{DP}_{\max}^{\text{f64}} \leq \left( \text{DP}_{\max}^{\text{sbox}} \right)^8 \quad (3.17)$$

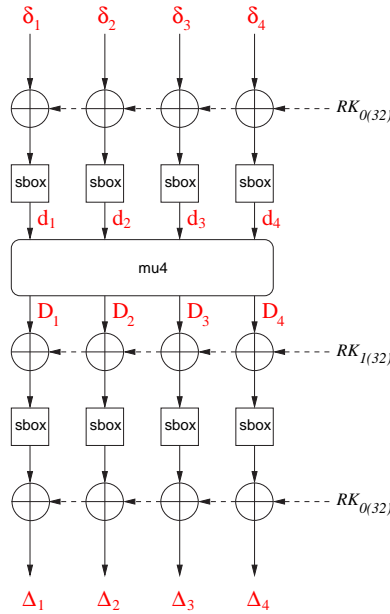


Figure 3.6: Differential Characteristic of a f32-function.

By taking into account Theorem 4, we obtain the following result, which can be used to define requirements on the minimal number of rounds needed to resist differential cryptanalysis.

**Theorem 8** *The probability of a (single-path) differential characteristic in FOX64/k/r is upper bounded by*

$$\left( DP_{\max}^{\text{sbox}} \right)^{2r} \tag{3.18}$$

Similarly, we obtain

**Theorem 9** *The probability of a (single-path) differential characteristic in FOX128/k/r is upper bounded by*

$$\left( DP_{\max}^{\text{sbox}} \right)^{4r} \tag{3.19}$$

### Linear Cryptanalysis

The computations regarding linear cryptanalysis are very similar to the ones done in the previous section. We have thus the following theorem.

**Theorem 10**

$$LP_{\max}^{\text{f32}} \leq \left( LP_{\max}^{\text{sbox}} \right)^4 \tag{3.20}$$

We can extend these computations to the `f64`-function (as defined in Fig. 2.4) and get

**Theorem 11**

$$\text{LP}_{\max}^{\text{f64}} \leq \left(\text{LP}_{\max}^{\text{sbox}}\right)^8 \quad (3.21)$$

By taking into account Theorem 4, we obtain the following result, which can be used to define requirements on the minimal number of rounds needed to resist linear cryptanalysis.

**Theorem 12** *The probability of a (single-path) linear characteristic in FOX64/ $k/r$  is upper bounded by*

$$\left(\text{LP}_{\max}^{\text{sbox}}\right)^{2r} \quad (3.22)$$

Similarly, we obtain

**Theorem 13** *The probability of a (single-path) linear characteristic in FOX128/ $k/r$  is upper bounded by*

$$\left(\text{LP}_{\max}^{\text{sbox}}\right)^{4r} \quad (3.23)$$

## 3.4 Key Schedule

In this part, we describe the motivations behind each one of the four parts constituting the key schedule algorithm.

### 3.4.1 General Rationales

The FOX key-schedule algorithm was designed with the following rationales in mind:

- The function taking a key  $K$  and the round number  $r$  in output and returning  $r$  subkeys should be a cryptographic pseudo-random, collision resistant and one-way function.
- The computational complexity of the key-schedule algorithm should not be penalized when the key-size is equal to 128 for FOX64 or to 256 for FOX128.
- The computational complexity of the key-schedule algorithm should not be penalized in FOX64 when the key size is smaller than 128 bits.
- The sequence of subkeys should be generated in any direction without a complexity penalty.
- All the bytes of  $MKEY$  should be randomized even when the key size is strictly smaller than  $ek$ .
- $MKEY$  should be expanded by XORing constants depending on  $r$  and  $ek$  with no overlap on these constants sequences.

- As FOX is a variable-round cipher, the key-schedule algorithm should resist *related-cipher attacks* as described by Wu in [Wu02]. More precisely, the subkey sequence must depend on the actual round number of the algorithm instance for which the sequence will be used.

### 3.4.2 P-Part

The goal of the P-part is to transform the user-provided key, which may have any length multiple of 8 smaller or equal than 256, in a fixed-size of 128-bit or 256-bit value. The chosen padding constant  $e - 2$  was checked regarding the following property.

**Lemma 1** *It is impossible to find two values of  $K$  with a length strictly smaller than 256 bits which lead to the same value of  $PKEY$ .*

*Proof:* In order for two different inputs to produce the same output during the padding operation, one has to concatenate the smaller one with a padding value which is contained in the one used for the larger input; this is only possible if the first  $\ell$  bytes of the padding constant are present in another location. The lemma follows from the fact that the first byte 0xB7 is unique in the constant.  $\diamond$

Note that in order to avoid that a padded key and non-padded key generates the same subkey sequence, a conditional negation has been incorporated in the NLx part of the key-schedule algorithm.

### 3.4.3 M-Part

When using small keys, a large part of the key-schedule state is known to a potential adversary: it is the padding constant. The goal of the M-part is hence to mix the entropy on all bytes. The following lemma insures that, when fed with two different inputs, the M-part will return two different outputs.

**Lemma 2** *The M-part is a permutation.*

*Proof:* The lemma follows from the fact that the M-part is an invertible application.  $\diamond$

### 3.4.4 L-Part

The goal of the L-part is to diversify the  $DKEY$  register (which serves as input for the NLx-part) at each round. The main design goals are its simplicity and its reversibility: as a LFSR step is equivalent to the multiplication by a constant in a finite field, the inverse operation is a division by the same constant. It is thus possible to evaluate the L in both directions. It was furthermore checked that the outputs (being 144 or 264 bits) for all  $12 \leq r \leq 255$  and all round numbers  $1 \leq i \leq r$  are unique.

### 3.4.5 NLx-Part

The goal of the NL part is to generate a pseudo-random stream of data as “cryptographically secure” as possible and as fast as possible. For this, it re-uses the round functions in its core, and it needs only a few supplementary operations.

# Chapter 4

## Implementation Aspects

In this chapter, we handle several issues linked to implementation aspects.

### 4.1 32/64-bit Platforms

Most modern CPUs architecture are 32- or 64-bit ones. In this section, we list several ways to optimize an implementation of FOX in terms of speed (*i.e.* of throughput).

#### 4.1.1 Subkeys Precomputation

Most of the time, block ciphers are used to encrypt *several* blocks of data, so it is very time-sparing to precompute the subkeys once for all and to store them in a table. Typically, one needs 128 bytes of memory to store all the subkeys for an implementation of FOX64 with 16 rounds and twice as much for FOX128.

#### 4.1.2 Implementation of f32 and f64 Using Table-Lookups

The f32 and f64 functions can be implemented very efficiently using a combinations of table-lookups and XORs. We will focus on the f32 function, but the considerations are similar for which concerns f64.

Let  $X_{0(8)}||X_{1(8)}||X_{2(8)}||X_{3(8)}$  be an input of f32. We denote by  $T_{0(8)}||T_{1(8)}||T_{2(8)}||T_{3(8)}$  the temporary result obtained after the mu4 application. Let  $RK_{0(8)}||RK_{1(8)}||RK_{2(8)}||RK_{3(8)}$  denote the first half of the round key. Finally, let  $V_{i(8)} \triangleq X_{i(8)} \oplus RK_{i(8)}$  for  $0 \leq i \leq 3$ . We have

$$\begin{pmatrix} T_{0(8)} \\ T_{1(8)} \\ T_{2(8)} \\ T_{3(8)} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \alpha \\ 1 & c & \alpha & 1 \\ c & \alpha & 1 & 1 \\ \alpha & 1 & c & 1 \end{pmatrix} \times \begin{pmatrix} \text{sbx}(V_{0(8)}) \\ \text{sbx}(V_{1(8)}) \\ \text{sbx}(V_{2(8)}) \\ \text{sbx}(V_{3(8)}) \end{pmatrix} \quad (4.1)$$

This equation may be rewritten as

$$\begin{pmatrix} T_{0(8)} \\ T_{1(8)} \\ T_{2(8)} \\ T_{3(8)} \end{pmatrix} = \text{sbox}(V_{0(8)}) \times \begin{pmatrix} 1 \\ 1 \\ c \\ \alpha \end{pmatrix} \oplus \text{sbox}(V_{1(8)}) \times \begin{pmatrix} 1 \\ c \\ \alpha \\ 1 \end{pmatrix} \oplus \\ \text{sbox}(V_{2(8)}) \times \begin{pmatrix} 1 \\ \alpha \\ 1 \\ c \end{pmatrix} \oplus \text{sbox}(V_{3(8)}) \times \begin{pmatrix} \alpha \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

Thus, one may precompute 4 tables of 256 4-bytes elements defined by

$$\begin{aligned} \text{TBSM}_0[\mathbf{a}] &\triangleq \begin{pmatrix} 1 \cdot \text{sbox}(\mathbf{a}) \\ 1 \cdot \text{sbox}(\mathbf{a}) \\ c \cdot \text{sbox}(\mathbf{a}) \\ \alpha \cdot \text{sbox}(\mathbf{a}) \end{pmatrix}, & \text{TBSM}_1[\mathbf{a}] &\triangleq \begin{pmatrix} 1 \cdot \text{sbox}(\mathbf{a}) \\ c \cdot \text{sbox}(\mathbf{a}) \\ \alpha \cdot \text{sbox}(\mathbf{a}) \\ 1 \cdot \text{sbox}(\mathbf{a}) \end{pmatrix} \\ \text{TBSM}_2[\mathbf{a}] &\triangleq \begin{pmatrix} 1 \cdot \text{sbox}(\mathbf{a}) \\ \alpha \cdot \text{sbox}(\mathbf{a}) \\ 1 \cdot \text{sbox}(\mathbf{a}) \\ c \cdot \text{sbox}(\mathbf{a}) \end{pmatrix}, & \text{TBSM}_3[\mathbf{a}] &\triangleq \begin{pmatrix} \alpha \cdot \text{sbox}(\mathbf{a}) \\ 1 \cdot \text{sbox}(\mathbf{a}) \\ 1 \cdot \text{sbox}(\mathbf{a}) \\ 1 \cdot \text{sbox}(\mathbf{a}) \end{pmatrix} \end{aligned}$$

and write

$$\begin{pmatrix} T_{0(8)} \\ T_{1(8)} \\ T_{2(8)} \\ T_{3(8)} \end{pmatrix} = \text{TBSM}_0[V_{0(8)}] \oplus \text{TBSM}_1[V_{1(8)}] \oplus \text{TBSM}_2[V_{2(8)}] \oplus \text{TBSM}_3[V_{3(8)}] \quad (4.2)$$

Similarly, after the second key-addition layer of f32, by denoting  $U_{0(8)}||U_{1(8)}||U_{2(8)}||U_{3(8)}$  this temporary result *before* the last substitution layer and by denoting  $W_{0(8)}||W_{1(8)}||W_{2(8)}||W_{3(8)}$ , the temporary result *after* the last substitution layer, one can use the same strategy with the following tables:

$$\begin{aligned} \text{TBS}_0[\mathbf{a}] &\triangleq \begin{pmatrix} \text{sbox}(\mathbf{a}) \\ 0 \\ 0 \\ 0 \end{pmatrix}, & \text{TBS}_1[\mathbf{a}] &\triangleq \begin{pmatrix} 0 \\ \text{sbox}(\mathbf{a}) \\ 0 \\ 0 \end{pmatrix} \\ \text{TBS}_2[\mathbf{a}] &\triangleq \begin{pmatrix} 0 \\ 0 \\ \text{sbox}(\mathbf{a}) \\ 0 \end{pmatrix}, & \text{TBS}_3[\mathbf{a}] &\triangleq \begin{pmatrix} 0 \\ 0 \\ 0 \\ \text{sbox}(\mathbf{a}) \end{pmatrix} \end{aligned}$$

and write

$$\begin{pmatrix} W_{0(8)} \\ W_{1(8)} \\ W_{2(8)} \\ W_{3(8)} \end{pmatrix} = \text{TBS}_0[U_{0(8)}] \oplus \text{TBS}_1[U_{1(8)}] \oplus \text{TBS}_2[U_{2(8)}] \oplus \text{TBS}_3[U_{3(8)}] \quad (4.3)$$

As outlined before, the process is similar for the implementation of the f64 function. In this case, we have to define two times 8 tables of 256 64-bit elements. The following table summarizes the size of the various tables for a fully-precomputed implementation :

	number of tables	width [bytes]	total size [bytes]
FOX64	$2 \times 4$	4	8'192
FOX128	$2 \times 8$	8	32'768

Depending on the target processor, the nearest cache (*i.e.* the fastest memory) size may be smaller than 32'768 bytes. In this case, one can spare half of the tables (at the cost of a few masking operations) by noting that all the TBS tables are “embedded” in the TBSM ones; this implementation strategy will be denoted *half-precomputed implementation*. This allows to reduce the fast memory needs to 4'096 and 16'384 bytes, respectively. The following table summarizes the L1 cache size of various processors:


Processor	cache size [kB]	Note	Impl
Alpha 21164	8	(data)	★
Alpha 21264	64	(data)	★
AMD Athlon XP	128	(data + code)	★
AMD Athlon MP	128	(data + code)	★
AMD Opteron	64	(data)	★
Intel Pentium III	16	(data)	★
Intel Pentium IV	8	(data)	★
Intel Xeon	8	(data)	★
Intel Itanium	16	(data)	★
Intel Itanium2	16	(data)	★
PowerPC G4	32	(data + code)	★
PowerPC G5	32	(data)	★
UltraSparc II	16	(data)	★
UltraSparc III	64	(data)	★

### Large and Very Large L1 Caches

For most modern microprocessors (denoted by ★ in the above table), a fully-precomputed implementation of FOX64 and FOX128 is probably the fastest possible solution. For the processors denoted by ★, a half-precomputed implementation is probably the best solution. The supplementary masking operations may be furthermore used to increase the instructions throughput on pipelined architectures.



### Small L1 Caches

Some microprocessors have a very small L1 data cache (they are denoted in  in the above table). In these cases, in the case of FOX128, even a half-precomputed implementation will result in many caches misses, inducing a performance penalty. For an Intel Pentium IV, a half-precomputed implementation of FOX64 is advantageous, while one can reduce the size of the precomputed data needed for a FOX128 implementation down to 8'192 bytes at the cost of at most 18 supplementary PSHUFW instructions. Although these operations will result in a performance penalty, the latter will be reduced since the highly-parrallelizable structure of the f64 function allows to fully use the pipeline and thus to improve the instructions throughput.

### Improving the Instruction Throughput

As most modern CPU architectures are pipelined ones, one can take this fact into account in order to improve performances of FOX implementations. There are two “dependency walls” in a FOX round function. The first one is just after the first subkey addition, the second one just after the second subkey addition. Inbetween, the additions of the table-lookup results may be done in any order, as an XOR is a commutative addition.

### 64-bit Capabilities of 32-bit Microprocessors

FOX128 is an excellent candidate for using the 64-bit instructions of actual 32-bit microprocessors. For instance, on the Intel architecture, the MMX/SSE/SSE2 instructions sets may be used to “emulate” a 64-bit microprocessor.

### Transforming the Extended Lai-Massey Scheme in a Lai-Massey Scheme

By expressing the Extended Lai-Massey scheme as in Fig. 3.1, one can compute very efficiently the two orthomorphisms as a single one on 64-bit architectures.

### Compilers / Data Structures

In order to get the best performances for FOX implementations written in a high-level language (like C), one can get big speed differences when using different compilers. Furthermore, the choice of the data structure of the precomputed tables and of the data to be encrypted plays an important role: implementing a simple way to access these data will result in a speed increase.

#### 4.1.3 Key-Schedule Algorithms

For applications needing a high key-agility, one can implement the various key-schedule algorithms using the same guidelines and tricks as for the core algorithm, since they share many common features.

## 4.2 8-bit Platforms

The resources representing the most important bottleneck in a block cipher implementation on a smartcard (which uses typically low-cost, 8-bit microprocessors) is of course the RAM usage. The amount of efficiently usable RAM available on a smartcard is typically in the order of 256 bytes. It may be a bit larger depending on the cases, but as this type of smart card is devoted to contain more than a simple encryption routine, FOX implementations on this kind of platforms will minimize the amount of necessary RAM. ROM is not so scarce as RAM on a smartcard, so the code size can be greater than the RAM usage. It is usually reasonable not to have a ROM size (instructions + possible precomputed tables) greater than 1024 bytes.

### 4.2.1 Four Memory Usage Strategies

Obviously, the most intensive computation are related to the evaluation of the `sbox` mapping and of the `mu4` and `mu8` mappings. We propose in the following table four different (the last one concerning uniquely FOX128) strategies using various amounts of precomputed data to implement these mappings. Note that the precomputed data may be stored in ROM and that the constants needed in the key-schedule algorithm are not taken into account.

Strategy	Precomputations	Data size
A	No precomputed data	24 B
B	<code>sbox(x)</code>	256 B
C	<code>sbox(x)</code> , <code>talpha(x)</code> , <code>dalpha(x)</code>	768 B
D	<code>sbox</code> , <code>stalpha(x)</code> , <code>sdalpha(x)</code> , <code>stalpha2(x)</code> , <code>sdalpha2(x)</code>	1280 B

Strategy A can be applied when extremely few memory is available. For this, one computes on-the-fly the `sbox` mapping, as it is described in §3.2, and all the operations in  $\text{GF}(2^8)$ . The sole needed constants are the small substitution boxes  $S_1$ ,  $S_2$  and  $S_3$  (see Fig. 3.3). Strategy A is clearly the slowest strategy.

A significant speed gain can be obtained if one precomputes the `sbox` mapping (Strategy B), the finite field operations being all computed dynamically.

A third possibility (Strategy C) is to precompute two more mappings: `talpha(x)` is a function mapping an element  $x$  to  $\alpha \cdot x$ , with the multiplication in  $\text{GF}(2^8)$ ; `dalpha(x)` is a function mapping an element  $x \in \text{GF}(2^8)$  to  $\alpha^{-1} \cdot x$ .

Finally, in the case of FOX128, a further speed gain may be obtained (Strategy D) by tabulating the five following mappings:

$$\begin{aligned}
 \text{sbox}(x) & : x \mapsto \text{sbox}(x) \\
 \text{stalpha}(x) & : x \mapsto \text{sbox}(x) \cdot \alpha \\
 \text{sdalpha}(x) & : x \mapsto \text{sbox}(x) \cdot \alpha^{-1} \\
 \text{stalpha2}(x) & : x \mapsto \text{sbox}(x) \cdot \alpha^2 \\
 \text{sdalpha2}(x) & : x \mapsto \text{sbox}(x) \cdot \alpha^{-2}
 \end{aligned}$$

### 4.2.2 Implementation of f32

The implementation of the  $\sigma_4/\mu_4$  layer is relatively straightforward:

$$\begin{aligned} Y_{0(8)} &= \text{sbox}(X_{0(8)}) \oplus \text{sbox}(X_{1(8)}) \oplus \text{sbox}(X_{2(8)}) \oplus \alpha \cdot \text{sbox}(X_{3(8)}) \\ Y_{1(8)} &= \text{sbox}(X_{0(8)}) \oplus \text{sbox}(X_{1(8)}) \oplus \text{sbox}(X_{3(8)}) \oplus \alpha \cdot \text{sbox}(X_{2(8)}) \oplus \alpha^{-1} \cdot \text{sbox}(X_{1(8)}) \\ Y_{2(8)} &= \text{sbox}(X_{0(8)}) \oplus \text{sbox}(X_{2(8)}) \oplus \text{sbox}(X_{3(8)}) \oplus \alpha \cdot \text{sbox}(X_{1(8)}) \oplus \alpha^{-1} \cdot \text{sbox}(X_{0(8)}) \\ Y_{3(8)} &= \text{sbox}(X_{1(8)}) \oplus \text{sbox}(X_{2(8)}) \oplus \text{sbox}(X_{3(8)}) \oplus \alpha \cdot \text{sbox}(X_{0(8)}) \oplus \alpha^{-1} \cdot \text{sbox}(X_{2(8)}) \end{aligned}$$

By carefully rewriting the above equations and by re-using some temporary results, one can easily minimize the number of **sbox**, **alpha**, **dalpha** evaluations and the number of  $\oplus$  operations. However, the resulting implementation is strongly dependent of the chosen strategy (as discussed in §4.2.1).

### 4.2.3 Implementation of f64

The implementation of the  $\sigma_8/\mu_8$  layer is not much complicated. By rewriting the operations as done above, one can easily obtain a fast implementation. For instance, in case of an

implementation following memory strategy C, one can obtain the following computations:

$$\begin{aligned}
Y_{0(8)} &= \text{sbox}(X_{0(8)}) \oplus \text{sbox}(X_{1(8)}) \oplus \text{sbox}(X_{2(8)}) \oplus \text{sbox}(X_{3(8)}) \oplus \\
&\quad \text{sbox}(X_{4(8)}) \oplus \text{sbox}(X_{5(8)}) \oplus \text{sbox}(X_{6(8)}) \oplus \alpha \cdot \text{sbox}(X_{7(8)}) \\
Y_{1(8)} &= \text{sbox}(X_{0(8)}) \oplus \text{sbox}(X_{1(8)}) \oplus \text{sbox}(X_{7(8)}) \oplus \\
&\quad \alpha \cdot (\text{sbox}(X_{1(8)}) \oplus \text{sbox}(X_{3(8)}) \oplus \alpha \cdot \text{sbox}(X_{4(8)}) \oplus \\
&\quad \alpha^{-1} \cdot (\text{sbox}(X_{2(8)}) \oplus \text{sbox}(X_{5(8)}) \oplus \alpha^{-1} \cdot (\text{sbox}(X_{2(8)}) \oplus \text{sbox}(X_{6(8)}))) \\
Y_{2(8)} &= \text{sbox}(X_{0(8)}) \oplus \text{sbox}(X_{6(8)}) \oplus \text{sbox}(X_{7(8)}) \oplus \\
&\quad \alpha \cdot (\text{sbox}(X_{0(8)}) \oplus \text{sbox}(X_{2(8)}) \oplus \alpha \cdot \text{sbox}(X_{3(8)}) \oplus \\
&\quad \alpha^{-1} \cdot (\text{sbox}(X_{1(8)}) \oplus \text{sbox}(X_{4(8)}) \oplus \alpha^{-1} \cdot (\text{sbox}(X_{1(8)}) \oplus \text{sbox}(X_{5(8)}))) \\
Y_{3(8)} &= \text{sbox}(X_{5(8)}) \oplus \text{sbox}(X_{6(8)}) \oplus \text{sbox}(X_{7(8)}) \oplus \\
&\quad \alpha \cdot (\text{sbox}(X_{1(8)}) \oplus \text{sbox}(X_{6(8)}) \oplus \alpha \cdot \text{sbox}(X_{2(8)}) \oplus \\
&\quad \alpha^{-1} \cdot (\text{sbox}(X_{0(8)}) \oplus \text{sbox}(X_{3(8)}) \oplus \alpha^{-1} \cdot (\text{sbox}(X_{0(8)}) \oplus \text{sbox}(X_{4(8)}))) \\
Y_{4(8)} &= \text{sbox}(X_{4(8)}) \oplus \text{sbox}(X_{5(8)}) \oplus \text{sbox}(X_{7(8)}) \oplus \\
&\quad \alpha \cdot (\text{sbox}(X_{0(8)}) \oplus \text{sbox}(X_{5(8)}) \oplus \alpha \cdot \text{sbox}(X_{1(8)}) \oplus \\
&\quad \alpha^{-1} \cdot (\text{sbox}(X_{2(8)}) \oplus \text{sbox}(X_{6(8)}) \oplus \alpha^{-1} \cdot (\text{sbox}(X_{3(8)}) \oplus \text{sbox}(X_{6(8)}))) \\
Y_{5(8)} &= \text{sbox}(X_{3(8)}) \oplus \text{sbox}(X_{4(8)}) \oplus \text{sbox}(X_{7(8)}) \oplus \\
&\quad \alpha \cdot (\text{sbox}(X_{4(8)}) \oplus \text{sbox}(X_{6(8)}) \oplus \alpha \cdot \text{sbox}(X_{0(8)}) \oplus \\
&\quad \alpha^{-1} \cdot (\text{sbox}(X_{1(8)}) \oplus \text{sbox}(X_{5(8)}) \oplus \alpha^{-1} \cdot (\text{sbox}(X_{2(8)}) \oplus \text{sbox}(X_{5(8)}))) \\
Y_{6(8)} &= \text{sbox}(X_{2(8)}) \oplus \text{sbox}(X_{3(8)}) \oplus \text{sbox}(X_{7(8)}) \oplus \\
&\quad \alpha \cdot (\text{sbox}(X_{3(8)}) \oplus \text{sbox}(X_{5(8)}) \oplus \alpha \cdot \text{sbox}(X_{6(8)}) \oplus \\
&\quad \alpha^{-1} \cdot (\text{sbox}(X_{0(8)}) \oplus \text{sbox}(X_{4(8)}) \oplus \alpha^{-1} \cdot (\text{sbox}(X_{1(8)}) \oplus \text{sbox}(X_{4(8)}))) \\
Y_{7(8)} &= \text{sbox}(X_{1(8)}) \oplus \text{sbox}(X_{2(8)}) \oplus \text{sbox}(X_{7(8)}) \oplus \\
&\quad \alpha \cdot (\text{sbox}(X_{2(8)}) \oplus \text{sbox}(X_{4(8)}) \oplus \alpha \cdot \text{sbox}(X_{5(8)}) \oplus \\
&\quad \alpha^{-1} \cdot (\text{sbox}(X_{3(8)}) \oplus \text{sbox}(X_{6(8)}) \oplus \alpha^{-1} \cdot (\text{sbox}(X_{0(8)}) \oplus \text{sbox}(X_{3(8)})))
\end{aligned}$$

This computation flow (consisting of  $71 \oplus$ , 15  $\alpha$  and 15  $\alpha^{-1}$  evaluations) is obviously not optimal in terms of operations; by using redundant temporary computations, one can spare a few more operations.

#### 4.2.4 Implementation of $\alpha$ and $\alpha^{-1}$

We give here a constant-time implementation of  $\alpha$  and  $\alpha^{-1}$ . The routines  $\alpha_2$  and  $\alpha^{-1}_2$  can be implemented by iterating twice  $\alpha$  and  $\alpha^{-1}$ , respectively. Note that these implementation does not take into account security issues related to other side-channel attacks, like SPA/DPA.

```
;; Implementation of talpha() on 8051
;;
;; R0      : input
;; R0      : output

MOV A, R0          ;; A := R0
RLC A             ;; left rotation through carry
MOV R0, A         ;; storing the result
CLR A            ;; A := 0
SUBB A, #0       ;; C set ? A = 0xFF : A = 0x00
ANL A, #F9       ;; C set ? A = 0xF9 : A = 0x00
XRL A, R0        ;; A := A XOR R0
MOV R0, A        ;; R0 := A

;; Implementation of dalpha() on 8051
;;
;; R0      : input
;; R0      : output

MOV A, R0          ;; A := R0
RRC A             ;; left rotation through carry
MOV R0, A         ;; storing the result
CLR A            ;; A := 0
SUBB A, #0       ;; C set ? A = 0xFF : A = 0x00
ANL A, #FC       ;; C set ? A = 0xFC : A = 0x00
XRL A, R0        ;; A := A XOR R0
MOV R0, A        ;; R0 := A
```

### 4.3 Hardware Implementations

In a hardware implementation, one spare table-lookups for `sbox` by taking into account its structure, as described in §3.2. For instance, most FPGAs may implement directly any 4-bit to 4-bit boolean mappings, which allows to wire directly `sbox`.

# Bibliography

- [Fei73] H. Feistel. Cryptography and data security. *Scientific American*, 228(5):15–23, 1973.
- [HLL<sup>+</sup>01] S. Hong, S. Lee, J. Lim, J. Sung, D. Cheon, and I. Cho. Provable security against differential and linear cryptanalysis for the SPN structure. In FSE '00, volume 1978 of *LNCS*, pages 273–283. Springer-Verlag, 2001.
- [LR88] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.
- [SV95] C. P. Schnorr and S. Vaudenay. Black box cryptanalysis of hash networks based on multipermutations. In *Advances in Cryptology - EUROCRYPT '94*, volume 950 of *LNCS*, pages 47–57. Springer-Verlag, 1995.
- [Vau00a] S. Vaudenay. Adaptive-attack norm for decorrelation and super-pseudorandomness. In *Proceedings of SAC '00*, volume 1758 of *LNCS*, pages 49–61. Springer-Verlag, 2000.
- [Vau00b] S. Vaudenay. On the Lai-Massey scheme. In *Advances in Cryptology - ASIA-CRYPT '99*, volume 1716 of *LNCS*, pages 9–19. Springer-Verlag, 2000.
- [Vau03] S. Vaudenay. Decorrelation: a theory for block cipher security. *Journal of Cryptology*, 16(4):249–286, 2003.
- [Wu02] H. Wu. Related-cipher attacks. In *ICICS '02*, volume 2513 of *LNCS*, pages 447–455. Springer-Verlag, 2002.

# Appendix A

## Test Vectors

```
1  FOX test vectors generator
2  -----
3
4
5
6  FOX64/16/128 key       : 00112233 44556677 8899AABB CCDDEEFF
7  FOX64/16/128 message  : 01234567 89ABCDEF
8  FOX64/16/128 ciphertext : FAC289D2 9CB9BAFF
9  FOX64/16/128 message  : 01234567 89ABCDEF
10
11
12
13 FOX64/16/192 key       : 00112233 44556677 8899AABB CCDDEEFF FFEEDDCC BBAA9988
14 FOX64/16/192 message  : 01234567 89ABCDEF
15 FOX64/16/192 ciphertext : D53B1866 7ECB7070
16 FOX64/16/192 message  : 01234567 89ABCDEF
17
18
19
20 FOX64/16/256 key       : 00112233 44556677 8899AABB CCDDEEFF FFEEDDCC BBAA9988 77665544 33221100
21 FOX64/16/256 message  : 01234567 89ABCDEF
22 FOX64/16/256 ciphertext : 87B551DF 4364C5D9
23 FOX64/16/256 message  : 01234567 89ABCDEF
24
25
26
27 FOX128/16/128 key      : 00112233 44556677 8899AABB CCDDEEFF
28 FOX128/16/128 message : 01234567 89ABCDEF FEDCBA98 76543210
29 FOX128/16/128 ciphertext : FA8E215A B8471AB2 71ABFFA1 34591451
30 FOX128/16/128 message : 01234567 89ABCDEF FEDCBA98 76543210
31
32
33
34 FOX128/16/192 key      : 00112233 44556677 8899AABB CCDDEEFF FFEEDDCC BBAA9988
35 FOX128/16/192 message : 01234567 89ABCDEF FEDCBA98 76543210
36 FOX128/16/192 ciphertext : 8E5B723A 5996745E FA898BE6 A1034361
37 FOX128/16/192 message : 01234567 89ABCDEF FEDCBA98 76543210
38
39
40
```

```
41 FOX128/16/256 key      : 00112233 44556677 8899AABB CCDDEEFF FFEEDDCC BBAA9988 77665544 33221100
42 FOX128/16/256 message : 01234567 89ABCDEF FEDCBA98 76543210
43 FOX128/16/256 ciphertext : FF852954 AA5954DA 7A936AFE 1F4DEDB3
44 FOX128/16/256 message : 01234567 89ABCDEF FEDCBA98 76543210
```



# Appendix B

## Reference Implementations

### B.1 File README

```

1  FOX / Reference implementation
2  Pascal Junod <pascal.junod@epfl.ch>
3  $Id: README,v 1.2 2003/09/24 11:08:02 pjunod Exp $
4  -----
5
6  The sole purpose of this reference code is to output
7  a set of test vectors and to help understanding the
8  structure of FOX. It is not fast, not portable, not
9  elegant and not secure. It implements a full
10 precomputed table-lookup strategy.
11
12 The code has been written for the IA32 architecture,
13 which is a little-endian architecture. It won't work
14 on a big-endian architecture.
```

### B.2 File Makefile

```

1  #####
2  ## FOX project / Reference implementation      ##
3  ## Pascal Junod <pascal.junod@epfl.ch>        ##
4  ##                                           ##
5  ## $Id: Makefile,v 1.5 2003/09/24 11:13:27 pjunod Exp $ ##
6  #####
7
8  EXEC_NAME =          fox_util
9
10 CFLAGS =             -W -Wall -pedantic -g
11
12 objects =           fox128.o fox64.o fox_ctx.o fox_cst.o fox_util.o
13
14 all:                $(objects)
15                    $(CC) -o $(EXEC_NAME) $(objects)
16
17 $(objects):          %.o: %.c %.h
18                    $(CC) -c $(CFLAGS) $< -o $@
```

```

19
20 .PHONY:          clean debug
21
22 clean:
23         -rm -f $(objects) *~ $(EXEC_NAME)
24

```

### B.3 File fox\_portable.h

```

1  /*****
2  /* FOX project / Reference implementation          */
3  /* Pascal Junod <pascal.junod@epfl.ch>           */
4  /*                                               */
5  /* Base file is "nessie.h"                       */
6  /* $Id: fox_portable.h,v 1.3 2003/09/24 11:16:43 pjunod Exp $ */
7  *****/
8
9  #ifndef _FOX_PORTABLE_H_
10 #define _FOX_PORTABLE_H_
11
12 #include <limits.h>
13
14 typedef signed char sint8;
15 typedef unsigned char uint8;
16
17 #if UINT_MAX >= 4294967295UL
18
19 typedef signed short sint16;
20 typedef signed int sint32;
21 typedef unsigned short uint16;
22 typedef unsigned int uint32;
23
24 #define ONE32  0xffffffffU
25
26 #else
27
28 typedef signed int sint16;
29 typedef signed long sint32;
30 typedef unsigned int uint16;
31 typedef unsigned long uint32;
32
33 #define ONE32  0xffffffffUL
34
35 #endif
36
37 #define ONE8   0xffU
38 #define ONE16  0xffffU
39
40 #define T08(x)  ((x) & ONE8)
41 #define T016(x) ((x) & ONE16)
42 #define T032(x) ((x) & ONE32)
43
44 #define EXTRACT8_BIT(d, b)  (((uint8)(d) & ((uint8)0x1 << (b))) >> (b))
45 #define EXTRACT16_BIT(d, b) (((uint16)(d) & ((uint16)0x1 << (b))) >> (b))
46 #define EXTRACT32_BIT(d, b) (((uint32)(d) & ((uint32)0x1 << (b))) >> (b))
47
48 #define ROTL8(v, n)  ((uint8)((v) << (n)) | ((uint8)(v) >> (8 - (n))))

```

```

49 #define ROTL16(v, n)  ((uint16)((v) << (n)) | ((uint16)(v) >> (16 - (n))))
50 #define ROTL32(v, n)  ((uint32)((v) << (n)) | ((uint32)(v) >> (32 - (n))))
51
52 /*
53  * U8T032_BIG(c) returns the 32-bit value stored in big-endian convention
54  * in the unsigned char array pointed to by c.
55  */
56
57 #define U8T032_BIG(c)  (((uint32)T08(*(c)) << 24) | ((uint32)T08(*(c) + 1)) << 16) | \
58                       ((uint32)T08(*(c) + 2)) << 8) | \
59                       ((uint32)T08(*(c) + 3)))
60
61 /*
62  * U8T032_LITTLE(c) returns the 32-bit value stored in little-endian convention
63  * in the unsigned char array pointed to by c.
64  */
65
66 #define U8T032_LITTLE(c)  (((uint32)T08(*(c))) | ((uint32)T08(*(c) + 1)) << 8) | \
67                          ((uint32)T08(*(c) + 2)) << 16) | ((uint32)T08(*(c) + 3)) << 24))
68
69 /*
70  * U32T08_BIG(c, v) stores the 32-bit-value v in big-endian convention
71  * into the unsigned char array pointed to by c.
72  */
73
74 #define U32T08_BIG(c, v)  do { \
75     uint32 x = (v); \
76     uint8 *d = (c); \
77     d[0] = T08(x >> 24); \
78     d[1] = T08(x >> 16); \
79     d[2] = T08(x >> 8); \
80     d[3] = T08(x); \
81   } while (0)
82
83 /*
84  * U32T08_LITTLE(c, v) stores the 32-bit-value v in little-endian convention
85  * into the unsigned char array pointed to by c.
86  */
87
88 #define U32T08_LITTLE(c, v)  do { \
89     uint32 x = (v); \
90     uint8 *d = (c); \
91     d[0] = T08(x); \
92     d[1] = T08(x >> 8); \
93     d[2] = T08(x >> 16); \
94     d[3] = T08(x >> 24); \
95   } while (0)
96
97 #endif /* _FOX_PORTABLE_H_ */

```

## B.4 File fox\_error.h

```

1  /*****
2  /* FOX project / Reference implementation */
3  /* Pascal Junod <pascal.junod@epfl.ch> */
4  /* */
5  /* $Id: fox_error.h,v 1.2 2002/11/19 15:30:03 pjunod Exp $ */

```

```

6  /*****
7
8  #ifndef _FOX_ERROR_H_
9  #define _FOX_ERROR_H_
10
11 #define FOX_ERROR_MEMORY_ALLOCATION      "\nError: impossible to allocate enough memory"
12 #define FOX_ERROR_CONTEXT_INITIALIZATION "\nError: could not initialize a context"
13 #define FOX_ERROR_TABLE_INITIALIZATION "\nError: could not initialize a table"
14 #define FOX_ERROR_KEY_INITIALIZATION    "\nError: could not initialize a key"
15 #define FOX_ERROR_UNKNOWN_TABLE_ID     "\nError: unknown table ID"
16 #define FOX_ERROR_UNKNOWN_MODE         "\nError: unknown mode"
17 #define FOX_BUG                         "\nError: probably a bug"
18
19 #endif /* _FOX_ERROR_H_ */

```

## B.5 File fox\_cst.h

```

1  /*****
2  /* FOX project / Reference implementation */
3  /* Pascal Junod <pascal.junod@epfl.ch> */
4  /* */
5  /* $Id: fox_cst.h,v 1.1 2003/09/24 11:32:51 pjunod Exp $ */
6  /*****
7
8  #ifndef _FOX_CST_H_
9  #define _FOX_CST_H_
10
11 #include "fox_portable.h"
12
13 /* Constants */
14
15 #define FOX_NUMBER_ROUNDS      FOX_NUMBER_ROUNDS_MIN
16
17 #define FOX64_TABLE_SIGMA4_MU4_ID0      0x00
18 #define FOX64_TABLE_SIGMA4_MU4_ID1      0x01
19 #define FOX64_TABLE_SIGMA4_MU4_ID2      0x02
20 #define FOX64_TABLE_SIGMA4_MU4_ID3      0x03
21
22 #define FOX64_TABLE_SIGMA4_ID0           0x04
23 #define FOX64_TABLE_SIGMA4_ID1           0x05
24 #define FOX64_TABLE_SIGMA4_ID2           0x06
25 #define FOX64_TABLE_SIGMA4_ID3           0x07
26
27 #define FOX128_TABLE_SIGMA8_ID0          0x08
28 #define FOX128_TABLE_SIGMA8_ID1          0x09
29 #define FOX128_TABLE_SIGMA8_ID2          0x0A
30 #define FOX128_TABLE_SIGMA8_ID3          0x0B
31
32 #define FOX128_TABLE_SIGMA8_MU8_ID0      0x10
33 #define FOX128_TABLE_SIGMA8_MU8_ID1      0x11
34 #define FOX128_TABLE_SIGMA8_MU8_ID2      0x12
35 #define FOX128_TABLE_SIGMA8_MU8_ID3      0x13
36 #define FOX128_TABLE_SIGMA8_MU8_ID4      0x14
37 #define FOX128_TABLE_SIGMA8_MU8_ID5      0x15
38 #define FOX128_TABLE_SIGMA8_MU8_ID6      0x16
39 #define FOX128_TABLE_SIGMA8_MU8_ID7      0x17
40

```

```

41  /* FOX_IRRPOLY = x^8+x^7+x^6+x^5+x^4+x^3+1 */
42
43  #define FOX_IRRPOLY 0x1F9
44
45  /* These are the first decimal of e-2 */
46
47  extern const uint8 FOX_KEY_PAD[32];
48
49  /* The three "small" S-boxes */
50
51  extern const uint8 FOX_S1[16];
52  extern const uint8 FOX_S2[16];
53  extern const uint8 FOX_S3[16];
54
55  /* Constants used in the key-schedule algorithm */
56
57  extern const uint8 FOX_MKEYM2;
58  extern const uint8 FOX_MKEYM1;
59
60  extern const uint32 FOX_LFSR_C;
61  extern const uint32 FOX_LFSR_FP;
62
63  #endif /* _FOX_CST_H_ */

```

## B.6 File fox\_cst.c

```

1  /*****
2  /* FOX project / Reference implementation */
3  /* Pascal Junod <pascal.junod@epfl.ch> */
4  /*
5  /* $Id: fox_cst.c,v 1.1 2003/09/24 11:33:04 pjunod Exp $
6  *****/
7
8  #include "fox_portable.h"
9  #include "fox_cst.h"
10
11  /* These are the first decimal of e-2 */
12
13  const uint8 FOX_KEY_PAD[32] = { 0xB7, 0xE1, 0x51, 0x62, 0x8A, 0xED, 0x2A, 0x6A,
14                                0xBF, 0x71, 0x58, 0x80, 0x9C, 0xF4, 0xF3, 0xC7,
15                                0x62, 0xE7, 0x16, 0x0F, 0x38, 0xB4, 0xDA, 0x56,
16                                0xA7, 0x84, 0xD9, 0x04, 0x51, 0x90, 0xCF, 0xEF };
17
18  /* The three "small" S-boxes */
19
20  const uint8 FOX_S1[16] = { 0x2, 0x5, 0x1, 0x9,
21                            0xE, 0xA, 0xC, 0x8,
22                            0x6, 0x4, 0x7, 0xF,
23                            0xD, 0xB, 0x0, 0x3 };
24
25  const uint8 FOX_S2[16] = { 0xB, 0x4, 0x1, 0xF,
26                            0x0, 0x3, 0xE, 0xD,
27                            0xA, 0x8, 0x7, 0x5,
28                            0xC, 0x2, 0x9, 0x6 };
29
30  const uint8 FOX_S3[16] = { 0xD, 0xA, 0xB, 0x1,
31                            0x4, 0x3, 0x8, 0x9,

```

```

32             0x5, 0x7, 0x2, 0xC,
33             0xF, 0x0, 0x6, 0xE };
34
35 /* Constants used in the key-schedule algorithm */
36
37 const uint8 FOX_MKEYM2 = 0x6A;
38 const uint8 FOX_MKEYM1 = 0x76;
39
40 const uint32 FOX_LFSR_C = 0x006A0000UL;
41 const uint32 FOX_LFSR_FP = 0x0100001BUL;
42

```

## B.7 File fox\_ctx.h

```

1  /*****
2  /* FOX project / Reference implementation */
3  /* Pascal Junod <pascal.junod@epfl.ch> */
4  /* */
5  /* $Id: fox_ctx.h,v 1.3 2002/12/31 13:56:55 pjunod Exp $ */
6  *****/
7
8  #ifndef _FOX_CTX_H_
9  #define _FOX_CTX_H_
10
11 #include "fox_portable.h"
12 #include "fox_cst.h"
13
14 /* Types */
15
16 typedef uint8 FOX_mode;
17
18
19 typedef struct {
20     uint32 *exp_key;
21     uint8 raw_key[32];
22     uint8 key_length;
23     uint8 rounds;
24 } FOX_key_;
25
26 typedef FOX_key_ *FOX_key;
27
28
29 typedef struct {
30     uint32 *val;
31     uint32 size_bytes;
32     uint8 id;
33 } FOX_table_;
34
35 typedef FOX_table_ *FOX_table;
36
37
38 typedef struct {
39     FOX_table sigma4_mu4_0;
40     FOX_table sigma4_mu4_1;
41     FOX_table sigma4_mu4_2;
42     FOX_table sigma4_mu4_3;
43

```

```
44     FOX_table sigma4_0;
45     FOX_table sigma4_1;
46     FOX_table sigma4_2;
47     FOX_table sigma4_3;
48
49 } FOX64_ctx_;
50
51 typedef FOX64_ctx_ *FOX64_ctx;
52
53 typedef struct {
54     FOX_table sigma8_mu8_0;
55     FOX_table sigma8_mu8_1;
56     FOX_table sigma8_mu8_2;
57     FOX_table sigma8_mu8_3;
58     FOX_table sigma8_mu8_4;
59     FOX_table sigma8_mu8_5;
60     FOX_table sigma8_mu8_6;
61     FOX_table sigma8_mu8_7;
62
63     FOX_table sigma8_0;
64     FOX_table sigma8_1;
65     FOX_table sigma8_2;
66     FOX_table sigma8_3;
67
68 } FOX128_ctx_;
69
70 typedef FOX128_ctx_ *FOX128_ctx;
71
72
73 /* Exportable routines */
74
75 extern int FOX64_init_ctx (FOX64_ctx *);
76 extern void FOX64_clean_ctx (FOX64_ctx);
77
78 extern int FOX128_init_ctx (FOX128_ctx *);
79 extern void FOX128_clean_ctx (FOX128_ctx);
80
81
82 extern int FOX64_init_key (FOX_key *, const FOX64_ctx,
83                          const uint32 *, const uint32,
84                          const uint8);
85 extern void FOX64_clean_key (FOX_key);
86
87 extern int FOX128_init_key (FOX_key *, const FOX128_ctx,
88                           const uint32 *, const uint32,
89                           const uint8);
90 extern void FOX128_clean_key (FOX_key);
91
92 extern void FOX_io (uint32 *);
93 extern void FOX_or (uint32 *);
94
95
96 /* Internal routines */
97
98 int FOX_init_table (FOX_table *, const uint8);
99 void FOX_clean_table (FOX_table);
100
101 uint32 FOX_times_alpha (const uint32);
```

```
102 uint32 FOX_div_alpha (const uint32);
103
104 uint32 FOX_eval_sbox (const uint32 x, const uint8 *s1,
105                      const uint8 *s2, const uint8 *s3);
106
107
108 #endif /* _FOX_CTX_H_ */
```

## B.8 File fox\_ctx.c

```
1  /*****
2  /* FOX project / Reference implementation */
3  /* Pascal Junod <pascal.junod@epfl.ch> */
4  /* */
5  /* $Id: fox_ctx.c,v 1.3 2002/12/31 13:56:27 pjunod Exp $ */
6  *****/
7
8  #include <stdlib.h>
9  #include <stdio.h>
10 #include <assert.h>
11 #include <string.h>
12
13 #include "fox_portable.h"
14 #include "fox_error.h"
15 #include "fox_ctx.h"
16 #include "fox64.h"
17 #include "fox128.h"
18
19
20 void FOX_or (uint32 *data)
21 {
22     uint32 l, r;
23
24     assert (data != NULL);
25
26     l = *data >> 16;
27     r = *data & 0xFFFF;
28
29     *data = (r << 16) | (l ^ r);
30 }
31
32 void FOX_io (uint32 *data)
33 {
34     uint32 l, r;
35
36     assert (data != NULL);
37
38     l = *data >> 16;
39     r = *data & 0xFFFF;
40
41     *data = ((l ^ r) << 16) | l;
42 }
43
44 uint32 FOX_times_alpha (const uint32 input)
45 {
46
47     if (input) {
```



```
48         return (input & 0x80) ? (input << 1) ^ FOX_IRRPOLY : input << 1;
49     } else {
50         return 0x00;
51     }
52 }
53
54 uint32 FOX_div_alpha (const uint32 input)
55 {
56
57     if (input) {
58         return (input & 0x01) ? (input ^ FOX_IRRPOLY) >> 1 : input >> 1;
59     } else {
60         return 0x00;
61     }
62 }
63
64 int FOX64_init_ctx (FOX64_ctx *ptr)
65 {
66     FOX64_ctx ctx;
67
68     if ( (ctx = malloc (sizeof (FOX64_ctx))) == NULL) {
69         fprintf (stderr, FOX_ERROR_MEMORY_ALLOCATION);
70         goto error_label;
71     }
72     if (FOX_init_table (&ctx->sigma4_mu4_0, FOX64_TABLE_SIGMA4_MU4_ID0)) {
73         goto error_label;
74     }
75     if (FOX_init_table (&ctx->sigma4_mu4_1, FOX64_TABLE_SIGMA4_MU4_ID1)) {
76         goto error_label;
77     }
78     if (FOX_init_table (&ctx->sigma4_mu4_2, FOX64_TABLE_SIGMA4_MU4_ID2)) {
79         goto error_label;
80     }
81     if (FOX_init_table (&ctx->sigma4_mu4_3, FOX64_TABLE_SIGMA4_MU4_ID3)) {
82         goto error_label;
83     }
84     if (FOX_init_table (&ctx->sigma4_0, FOX64_TABLE_SIGMA4_ID0)) {
85         goto error_label;
86     }
87     if (FOX_init_table (&ctx->sigma4_1, FOX64_TABLE_SIGMA4_ID1)) {
88         goto error_label;
89     }
90     if (FOX_init_table (&ctx->sigma4_2, FOX64_TABLE_SIGMA4_ID2)) {
91         goto error_label;
92     }
93     if (FOX_init_table (&ctx->sigma4_3, FOX64_TABLE_SIGMA4_ID3)) {
94         goto error_label;
95     }
96
97     *ptr = ctx;
98
99     return 0;
100
101 error_label:
102     fprintf (stderr, FOX_ERROR_CONTEXT_INITIALIZATION);
103     FOX64_clean_ctx (ctx);
104
105     return -1;
```

```
106 }
107
108
109 void FOX64_clean_ctx (FOX64_ctx ctx)
110 {
111     if (ctx != NULL) {
112         FOX_clean_table (ctx->sigma4_mu4_0);
113         FOX_clean_table (ctx->sigma4_mu4_1);
114         FOX_clean_table (ctx->sigma4_mu4_2);
115         FOX_clean_table (ctx->sigma4_mu4_3);
116
117         FOX_clean_table (ctx->sigma4_0);
118         FOX_clean_table (ctx->sigma4_1);
119         FOX_clean_table (ctx->sigma4_2);
120         FOX_clean_table (ctx->sigma4_3);
121
122         free (memset (ctx, 0x00, sizeof (FOX64_ctx_)));
123     }
124 }
125
126 int FOX128_init_ctx (FOX128_ctx *ptr)
127 {
128     FOX128_ctx ctx;
129
130     if ( (ctx = malloc (sizeof (FOX128_ctx_))) == NULL) {
131         fprintf (stderr, FOX_ERROR_MEMORY_ALLOCATION);
132         goto error_label;
133     }
134     if (FOX_init_table (&ctx->sigma8_mu8_0, FOX128_TABLE_SIGMA8_MU8_ID0)) {
135         goto error_label;
136     }
137     if (FOX_init_table (&ctx->sigma8_mu8_1, FOX128_TABLE_SIGMA8_MU8_ID1)) {
138         goto error_label;
139     }
140     if (FOX_init_table (&ctx->sigma8_mu8_2, FOX128_TABLE_SIGMA8_MU8_ID2)) {
141         goto error_label;
142     }
143     if (FOX_init_table (&ctx->sigma8_mu8_3, FOX128_TABLE_SIGMA8_MU8_ID3)) {
144         goto error_label;
145     }
146     if (FOX_init_table (&ctx->sigma8_mu8_4, FOX128_TABLE_SIGMA8_MU8_ID4)) {
147         goto error_label;
148     }
149     if (FOX_init_table (&ctx->sigma8_mu8_5, FOX128_TABLE_SIGMA8_MU8_ID5)) {
150         goto error_label;
151     }
152     if (FOX_init_table (&ctx->sigma8_mu8_6, FOX128_TABLE_SIGMA8_MU8_ID6)) {
153         goto error_label;
154     }
155     if (FOX_init_table (&ctx->sigma8_mu8_7, FOX128_TABLE_SIGMA8_MU8_ID7)) {
156         goto error_label;
157     }
158     if (FOX_init_table (&ctx->sigma8_0, FOX128_TABLE_SIGMA8_ID0)) {
159         goto error_label;
160     }
161     if (FOX_init_table (&ctx->sigma8_1, FOX128_TABLE_SIGMA8_ID1)) {
162         goto error_label;
163     }
164 }
```

```
164     if (FOX_init_table (&ctx->sigma8_2, FOX128_TABLE_SIGMA8_ID2)) {
165         goto error_label;
166     }
167     if (FOX_init_table (&ctx->sigma8_3, FOX128_TABLE_SIGMA8_ID3)) {
168         goto error_label;
169     }
170
171     *ptr = ctx;
172
173     return 0;
174
175 error_label:
176     fprintf (stderr, FOX_ERROR_CONTEXT_INITIALIZATION);
177     FOX128_clean_ctx (ctx);
178
179     return -1;
180 }
181
182 void FOX128_clean_ctx (FOX128_ctx ctx)
183 {
184     if (ctx != NULL) {
185         FOX_clean_table (ctx->sigma8_mu8_0);
186         FOX_clean_table (ctx->sigma8_mu8_1);
187         FOX_clean_table (ctx->sigma8_mu8_2);
188         FOX_clean_table (ctx->sigma8_mu8_3);
189         FOX_clean_table (ctx->sigma8_mu8_4);
190         FOX_clean_table (ctx->sigma8_mu8_5);
191         FOX_clean_table (ctx->sigma8_mu8_6);
192         FOX_clean_table (ctx->sigma8_mu8_7);
193
194         FOX_clean_table (ctx->sigma8_0);
195         FOX_clean_table (ctx->sigma8_1);
196         FOX_clean_table (ctx->sigma8_2);
197         FOX_clean_table (ctx->sigma8_3);
198
199         free (memset (ctx, 0x00, sizeof (FOX128_ctx)));
200     }
201 }
202
203 int FOX64_init_key (FOX_key *ptr,
204                   const FOX64_ctx ctx,
205                   const uint32 *words,
206                   const uint32 length,
207                   const uint8 rounds)
208 {
209     FOX_key key;
210     uint32 i, j;
211     uint32 k, o;
212     uint8 temp8[32];
213     uint8 dkey[32];
214     uint32 dkey32[8], temp32[8];
215     uint32 b, ek;
216     uint32 lfsr_state;
217     uint8 lfsr[4];
218
219     assert (length <= 256);
220     assert (length % 8 == 0);
221 }
```

```
222     if ( (key = malloc (sizeof (FOX_key_))) == NULL) {
223         fprintf (stderr, FOX_ERROR_MEMORY_ALLOCATION);
224         goto error_label;
225     }
226     if ( (key->exp_key = malloc (sizeof (uint32) * 2 * rounds )) == NULL) {
227         fprintf (stderr, FOX_ERROR_MEMORY_ALLOCATION);
228         goto error_label;
229     }
230
231     memcpy (key->raw_key, words, (length >> 3));
232
233     /* Endianness issue here ! */
234     for (k = 0; k < (length >> 5); k++) {
235         U32T08_BIG (temp8 + (k * 4), words[k]);
236     }
237
238     /* Size in 32-bit words */
239     key->key_length = length >> 5;
240
241     key->rounds = rounds;
242
243     /* Computation of the state bit b and of ek */
244
245     if ( (length == 128) || (length == 256) ) {
246         b = 1;
247     } else {
248         b = 0;
249     }
250
251     if (length <= 128) {
252         ek = 128;
253     } else {
254         ek = 256;
255     }
256
257     /* P-part */
258
259     if (length < ek) {
260         for (i = (length >> 3), j = 0; i < 32; i++, j++) {
261             temp8[i] = FOX_KEY_PAD[j];
262         }
263     }
264
265     /* M-part */
266
267     if (length < ek) {
268         temp8[0] ^= (FOX_MKEYM2 + FOX_MKEYM1);
269         temp8[1] ^= (FOX_MKEYM1 + temp8[0]);
270         for (i = 2; i < (ek >> 3); i++) {
271             temp8[i] ^= (temp8[i - 2] + temp8[i - 1]);
272         }
273     }
274
275     /* D-Part */
276
277     /* Initialization of the LFSR */
278     lfsr_state = FOX_LFSR_C | ((uint32)rounds << 8) | (~rounds & 0xFF);
279
```

```

280     /* We back-clock the LFSR once */
281     if (lfsr_state & 0x1) {
282         lfsr_state ^= FOX_LFSR_FP;
283     }
284     lfsr_state >>= 1;
285
286     for (i = 0; i < rounds; i++) {
287         j = 0;
288         while (j < (ek >> 3)) {
289             if ( (j % 3) == 0 ) {
290                 /* We have to clock the LFSR */
291                 lfsr_state <<= 1;
292                 if (lfsr_state & 0x01000000) {
293                     lfsr_state ^= FOX_LFSR_FP;
294                 }
295                 /* Endianness issue here ! */
296                 U32T08_BIG (lfsr, lfsr_state);
297             }
298             dkey[j] = temp8[j] ^ lfsr[(j % 3) + 1];
299             j++;
300         }
301
302         for (j = 0; j < (ek >> 5); j++) {
303             dkey32[j] = U8T032_BIG (dkey + (j << 2));
304         }
305
306         /* NL-part : we feed the current DKEY to the NLx part */
307         /* sigma4 - mu4 operation */
308         for (j = 0; j < (ek >> 5); j++) {
309             o = ctx->sigma4_mu4_0->val[(dkey32[j] & 0xFF000000) >> 24];
310             o ^= ctx->sigma4_mu4_1->val[(dkey32[j] & 0x00FF0000) >> 16];
311             o ^= ctx->sigma4_mu4_2->val[(dkey32[j] & 0x0000FF00) >> 8];
312             o ^= ctx->sigma4_mu4_3->val[(dkey32[j] & 0x000000FF)];
313             dkey32[j] = o;
314         }
315
316         if (ek == 128) {
317             /* mix64 operation */
318             temp32[0] = dkey32[1] ^ dkey32[2] ^ dkey32[3];
319             temp32[1] = dkey32[0] ^ dkey32[2] ^ dkey32[3];
320             temp32[2] = dkey32[0] ^ dkey32[1] ^ dkey32[3];
321             temp32[3] = dkey32[0] ^ dkey32[1] ^ dkey32[2];
322         } else {
323             /* mix64h operation */
324             temp32[0] = dkey32[2] ^ dkey32[4] ^ dkey32[6];
325             temp32[1] = dkey32[3] ^ dkey32[5] ^ dkey32[7];
326             temp32[2] = dkey32[0] ^ dkey32[4] ^ dkey32[6];
327             temp32[3] = dkey32[1] ^ dkey32[5] ^ dkey32[7];
328             temp32[4] = dkey32[0] ^ dkey32[2] ^ dkey32[6];
329             temp32[5] = dkey32[1] ^ dkey32[3] ^ dkey32[7];
330             temp32[6] = dkey32[0] ^ dkey32[2] ^ dkey32[4];
331             temp32[7] = dkey32[1] ^ dkey32[3] ^ dkey32[5];
332         }
333         /* Constant addition */
334         for (j = 0; j < (ek >> 5); j++) {
335             temp32[j] ^= U8T032_BIG (FOX_KEY_PAD + (j << 2));
336         }
337         /* Conditional flip */

```

```
338     if (b) {
339         for (j = 0; j < (ek >> 5); j++) {
340             temp32[j] = ~temp32[j];
341         }
342     }
343
344     /* sigma4 operation */
345     for (j = 0; j < (ek >> 5); j++) {
346         o = ctx->sigma4_0->val[(temp32[j] & 0xFF000000) >> 24];
347         o ^= ctx->sigma4_1->val[(temp32[j] & 0x00FF0000) >> 16];
348         o ^= ctx->sigma4_2->val[(temp32[j] & 0x0000FF00) >> 8];
349         o ^= ctx->sigma4_3->val[(temp32[j] & 0x000000FF)];
350         temp32[j] = o;
351     }
352
353     if (ek == 128) {
354         /* Hashing */
355         dkey32[0] = temp32[0] ^ temp32[2];
356         dkey32[1] = temp32[1] ^ temp32[3];
357
358         /* Encryption phase */
359         FOX_lmor64 (dkey32, words, ctx);
360         FOX_lmld64 (dkey32, words + 2, ctx);
361         *(key->exp_key + 2*i) = dkey32[0];
362         *(key->exp_key + 2*i) = dkey32[1];
363     } else {
364         /* Hashing */
365         dkey32[0] = temp32[0] ^ temp32[1];
366         dkey32[1] = temp32[2] ^ temp32[3];
367         dkey32[2] = temp32[4] ^ temp32[5];
368         dkey32[3] = temp32[6] ^ temp32[7];
369
370         temp32[0] = dkey32[0] ^ dkey[1];
371         temp32[1] = dkey32[2] ^ dkey[3];
372         /* Encryption phase */
373         FOX_lmor64 (temp32, words, ctx);
374         FOX_lmor64 (temp32, words + 2, ctx);
375         FOX_lmor64 (temp32, words + 4, ctx);
376         FOX_lmld64 (temp32, words + 6, ctx);
377         *(key->exp_key + 2*i) = temp32[0];
378         *(key->exp_key + 2*i + 1) = temp32[1];
379     }
380 }
381
382 *ptr = key;
383
384 return 0;
385
386 error_label:
387     FOX64_clean_key (key);
388
389     return -1;
390 }
391
392 void FOX64_clean_key (FOX_key k)
393 {
394     if (k != NULL) {
395         if (k->exp_key != NULL) {
```

```
396         free (memset (k->exp_key, 0x00, k->rounds * 2 * sizeof (uint32)));
397     }
398     free (memset (k, 0x00, sizeof (FOX_key_)));
399 }
400 }
401
402 int FOX128_init_key (FOX_key *ptr,
403                    const FOX128_ctx ctx,
404                    const uint32 *words,
405                    const uint32 length,
406                    const uint8 rounds)
407 {
408     FOX_key key;
409     uint32 i, j;
410     uint32 k;
411     uint32 o[2];
412     uint8 temp8[32];
413     uint8 dkey8[32];
414     uint32 dkey32[8], temp32[8];
415     uint32 b, ek;
416     uint32 lfsr_state;
417     uint8 lfsr[4];
418
419     assert (length <= 256);
420     assert (length % 8 == 0);
421
422     if ( (key = malloc (sizeof (FOX_key_))) == NULL) {
423         fprintf (stderr, FOX_ERROR_MEMORY_ALLOCATION);
424         goto error_label;
425     }
426     if ( (key->exp_key = malloc (sizeof (uint32) * 4 * rounds )) == NULL) {
427         fprintf (stderr, FOX_ERROR_MEMORY_ALLOCATION);
428         goto error_label;
429     }
430
431     memcpy (key->raw_key, words, (length >> 3));
432
433     /* Endianness issue here ! */
434     for (k = 0; k < (length >> 5); k++) {
435         U32T08_BIG (temp8 + (k * 4), words[k]);
436     }
437
438     /* Size in 32-bit words */
439     key->key_length = length >> 5;
440
441     key->rounds = rounds;
442
443     /* Computation of the state bit b and of ek */
444
445     if ( length == 256 ) {
446         b = 1;
447     } else {
448         b = 0;
449     }
450     ek = 256;
451
452
453     /* P-part */
```

```

454
455     if (length < ek) {
456         for (i = (length >> 3), j = 0; i < 32; i++, j++) {
457             temp8[i] = FOX_KEY_PAD[j];
458         }
459     }
460
461     /* M-part */
462
463     if (length < ek) {
464         temp8[0] ^= (FOX_MKEYM2 + FOX_MKEYM1);
465         temp8[1] ^= (FOX_MKEYM1 + temp8[0]);
466         for (i = 2; i < (ek >> 3); i++) {
467             temp8[i] ^= (temp8[i - 2] + temp8[i - 1]);
468         }
469     }
470
471     /* D-Part */
472
473     /* Initialization of the LFSR */
474     lfsr_state = FOX_LFSR_C | ((uint32)rounds << 8) | (~rounds & 0xFF);
475
476     /* We back-clock the LFSR once */
477     if (lfsr_state & 0x1) {
478         lfsr_state ^= FOX_LFSR_FP;
479     }
480     lfsr_state >>= 1;
481
482     for (i = 0; i < rounds; i++) {
483         j = 0;
484         while (j < (ek >> 3)) {
485             if ((j % 3) == 0) {
486                 /* We have to clock the LFSR */
487                 lfsr_state <<= 1;
488                 if (lfsr_state & 0x01000000) {
489                     lfsr_state ^= FOX_LFSR_FP;
490                 }
491                 /* Endianness issue here ! */
492                 U32T08_BIG (lfsr, lfsr_state);
493             }
494             dkey8[j] = temp8[j] ^ lfsr[(j % 3) + 1];
495             j++;
496         }
497
498         for (j = 0; j < 8; j++) {
499             dkey32[j] = U8T032_BIG (dkey8 + (j << 2));
500         }
501
502         /* NL Part */
503
504         /* sigma8 - mu8 operation */
505         for (j = 0; j < 4; j++) {
506             o[0] = ctx->sigma8_mu8_0->val[(dkey32[2*j] & 0xFF000000) >> 23];
507             o[1] = ctx->sigma8_mu8_0->val[((dkey32[2*j] & 0xFF000000) >> 23) + 1];
508             o[0] ^= ctx->sigma8_mu8_1->val[(dkey32[2*j] & 0x00FF0000) >> 15];
509             o[1] ^= ctx->sigma8_mu8_1->val[((dkey32[2*j] & 0x00FF0000) >> 15) + 1];
510             o[0] ^= ctx->sigma8_mu8_2->val[(dkey32[2*j] & 0x0000FF00) >> 7];
511             o[1] ^= ctx->sigma8_mu8_2->val[((dkey32[2*j] & 0x0000FF00) >> 7) + 1];

```



```

512         o[0] ^= ctx->sigma8_mu8_3->val[(dkey32[2*j] & 0x000000FF) << 1];
513         o[1] ^= ctx->sigma8_mu8_3->val[((dkey32[2*j] & 0x000000FF) << 1) + 1];
514
515         o[0] ^= ctx->sigma8_mu8_4->val[(dkey32[2*j+1] & 0xFF000000) >> 23];
516         o[1] ^= ctx->sigma8_mu8_4->val[((dkey32[2*j+1] & 0xFF000000) >> 23) + 1];
517         o[0] ^= ctx->sigma8_mu8_5->val[(dkey32[2*j+1] & 0x00FF0000) >> 15];
518         o[1] ^= ctx->sigma8_mu8_5->val[((dkey32[2*j+1] & 0x00FF0000) >> 15) + 1];
519         o[0] ^= ctx->sigma8_mu8_6->val[(dkey32[2*j+1] & 0x0000FF00) >> 7];
520         o[1] ^= ctx->sigma8_mu8_6->val[((dkey32[2*j+1] & 0x0000FF00) >> 7) + 1];
521         o[0] ^= ctx->sigma8_mu8_7->val[(dkey32[2*j+1] & 0x000000FF) << 1];
522         o[1] ^= ctx->sigma8_mu8_7->val[((dkey32[2*j+1] & 0x000000FF) << 1) + 1];
523
524         dkey32[2*j] = o[0];
525         dkey32[2*j + 1] = o[1];
526     }
527
528     /* mix128 operation */
529
530     temp32[0] = dkey32[2] ^ dkey32[4] ^ dkey32[6];
531     temp32[1] = dkey32[3] ^ dkey32[5] ^ dkey32[7];
532     temp32[2] = dkey32[0] ^ dkey32[4] ^ dkey32[6];
533     temp32[3] = dkey32[1] ^ dkey32[5] ^ dkey32[7];
534     temp32[4] = dkey32[0] ^ dkey32[2] ^ dkey32[6];
535     temp32[5] = dkey32[1] ^ dkey32[3] ^ dkey32[7];
536     temp32[6] = dkey32[0] ^ dkey32[2] ^ dkey32[4];
537     temp32[7] = dkey32[1] ^ dkey32[3] ^ dkey32[5];
538
539     /* Constant addition */
540     for (j = 0; j < 8; j++) {
541         temp32[j] ^= U8T032_BIG (FOX_KEY_PAD + (j << 2));
542     }
543     /* Conditional flip */
544     if (b) {
545         for (j = 0; j < 8; j++) {
546             temp32[j] = ~temp32[j];
547         }
548     }
549
550     /* sigma8 operation */
551     for (j = 0; j < 4; j++) {
552         o[0] = ctx->sigma8_0->val[(dkey32[2*j] & 0xFF000000) >> 24];
553         o[1] ^= ctx->sigma8_1->val[(dkey32[2*j] & 0x00FF0000) >> 16];
554         o[0] ^= ctx->sigma8_2->val[(dkey32[2*j] & 0x0000FF00) >> 8];
555         o[0] ^= ctx->sigma8_3->val[(dkey32[2*j] & 0x000000FF)];
556
557         o[1] = ctx->sigma8_0->val[(dkey32[2*j+1] & 0xFF000000) >> 24];
558         o[1] ^= ctx->sigma8_1->val[(dkey32[2*j+1] & 0x00FF0000) >> 16];
559         o[1] ^= ctx->sigma8_2->val[(dkey32[2*j+1] & 0x0000FF00) >> 8];
560         o[1] ^= ctx->sigma8_3->val[(dkey32[2*j+1] & 0x000000FF)];
561
562         dkey32[2*j] = o[0];
563         dkey32[2*j + 1] = o[1];
564     }
565
566     dkey32[0] = temp32[0] ^ temp32[4];
567     dkey32[1] = temp32[1] ^ temp32[5];
568     dkey32[2] = temp32[2] ^ temp32[6];
569     dkey32[3] = temp32[3] ^ temp32[7];

```

```
570
571     /* Encryption phase */
572     FOX_elmor128 (dkey32, words, ctx);
573     FOX_elmid128 (dkey32, words + 4, ctx);
574
575     *(key->exp_key + 4*i)    = temp32[0];
576     *(key->exp_key + 4*i + 1) = temp32[1];
577     *(key->exp_key + 4*i + 2) = temp32[2];
578     *(key->exp_key + 4*i + 3) = temp32[3];
579 }
580
581 *ptr = key;
582
583 return 0;
584
585 error_label:
586     FOX128_clean_key (key);
587
588     return -1;
589 }
590
591 void FOX128_clean_key (FOX_key k)
592 {
593     if (k != NULL) {
594         if (k->exp_key != NULL) {
595             free (memset (k->exp_key, 0x00, k->rounds * 4 * sizeof (uint32)));
596         }
597         free (memset (k, 0x00, sizeof (FOX_key_)));
598     }
599 }
600
601 int FOX_init_table (FOX_table *ptr, const uint8 id)
602 {
603     uint32 i, size, tmp;
604     FOX_table table;
605
606     if ( (table = malloc (sizeof (FOX_table_))) == NULL) {
607         fprintf (stderr, FOX_ERROR_MEMORY_ALLOCATION);
608         goto error_label;
609     }
610
611     if (id & 0x10) {
612         size = 2;
613     } else {
614         size = 1;
615     }
616
617     if ( (table->val = malloc (256 * sizeof(uint32) * size)) == NULL) {
618         fprintf (stderr, FOX_ERROR_MEMORY_ALLOCATION);
619         goto error_label;
620     }
621     table->id = id;
622     table->size_bytes = 256 * sizeof(uint32) * size;
623
624     switch (id) {
625
626         case FOX64_TABLE_SIGMA4_MU4_ID0:
627             for (i = 0; i < 256; i++) {
```

```

628         tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
629         table->val[i] = tmp << 24;
630         table->val[i] |= tmp << 16;
631         table->val[i] |= (FOX_div_alpha(tmp) ^ tmp) << 8;
632         table->val[i] |= FOX_times_alpha (tmp);
633     }
634     break;
635
636     case FOX64_TABLE_SIGMA4_MU4_ID1:
637         for (i = 0; i < 256; i++) {
638             tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
639             table->val[i] = tmp << 24;
640             table->val[i] |= (FOX_div_alpha(tmp) ^ tmp) << 16;
641             table->val[i] |= FOX_times_alpha (tmp) << 8;
642             table->val[i] |= tmp;
643         }
644         break;
645
646     case FOX64_TABLE_SIGMA4_MU4_ID2:
647         for (i = 0; i < 256; i++) {
648             tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
649             table->val[i] = tmp << 24;
650             table->val[i] |= FOX_times_alpha (tmp) << 16;
651             table->val[i] |= tmp << 8;
652             table->val[i] |= (FOX_div_alpha(tmp) ^ tmp);
653         }
654         break;
655
656     case FOX64_TABLE_SIGMA4_MU4_ID3:
657         for (i = 0; i < 256; i++) {
658             tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
659             table->val[i] = FOX_times_alpha (tmp) << 24;
660             table->val[i] |= tmp << 16;
661             table->val[i] |= tmp << 8;
662             table->val[i] |= tmp;
663         }
664         break;
665
666     case FOX128_TABLE_SIGMA8_MU8_ID0:
667         for (i = 0; i < 256; i++) {
668             tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
669             table->val[2*i] = tmp << 24;
670             table->val[2*i] |= tmp << 16;
671             table->val[2*i] |= (FOX_times_alpha (tmp) ^ tmp) << 8;
672             table->val[2*i] |= (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp)));
673             table->val[2*i+1] = FOX_times_alpha (tmp) << 24;
674             table->val[2*i+1] |= FOX_times_alpha (FOX_times_alpha (tmp)) << 16;
675             table->val[2*i+1] |= FOX_div_alpha (tmp) << 8;
676             table->val[2*i+1] |= FOX_div_alpha (FOX_div_alpha (tmp));
677         }
678         break;
679
680     case FOX128_TABLE_SIGMA8_MU8_ID1:
681         for (i = 0; i < 256; i++) {
682             tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
683             table->val[2*i] = tmp << 24;
684             table->val[2*i] |= (FOX_times_alpha (tmp) ^ tmp) << 16;
685             table->val[2*i] |= (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp))) << 8;

```

```
686         table->val[2*i] |= FOX_times_alpha (tmp);
687         table->val[2*i+1] = FOX_times_alpha (FOX_times_alpha (tmp)) << 24;
688         table->val[2*i+1] |= FOX_div_alpha (tmp) << 16;
689         table->val[2*i+1] |= FOX_div_alpha (FOX_div_alpha (tmp)) << 8;
690         table->val[2*i+1] |= tmp;
691     }
692     break;
693
694 case FOX128_TABLE_SIGMA8_MU8_ID2:
695     for (i = 0; i < 256; i++) {
696         tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
697         table->val[2*i] = tmp << 24;
698         table->val[2*i] |= (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp))) << 16;
699         table->val[2*i] |= FOX_times_alpha (tmp) << 8;
700         table->val[2*i] |= FOX_times_alpha (FOX_times_alpha (tmp));
701         table->val[2*i+1] = FOX_div_alpha (tmp) << 24;
702         table->val[2*i+1] |= FOX_div_alpha (FOX_div_alpha (tmp)) << 16;
703         table->val[2*i+1] |= tmp << 8;
704         table->val[2*i+1] |= (FOX_times_alpha (tmp) ^ tmp);
705     }
706     break;
707
708 case FOX128_TABLE_SIGMA8_MU8_ID3:
709     for (i = 0; i < 256; i++) {
710         tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
711         table->val[2*i] = tmp << 24;
712         table->val[2*i] |= FOX_times_alpha (tmp) << 16;
713         table->val[2*i] |= FOX_times_alpha (FOX_times_alpha (tmp)) << 8;
714         table->val[2*i] |= FOX_div_alpha (tmp);
715         table->val[2*i+1] = FOX_div_alpha (FOX_div_alpha (tmp)) << 24;
716         table->val[2*i+1] |= tmp << 16;
717         table->val[2*i+1] |= (FOX_times_alpha (tmp) ^ tmp) << 8;
718         table->val[2*i+1] |= (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp)));
719     }
720     break;
721
722 case FOX128_TABLE_SIGMA8_MU8_ID4:
723     for (i = 0; i < 256; i++) {
724         tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
725         table->val[2*i] = tmp << 24;
726         table->val[2*i] |= FOX_times_alpha (FOX_times_alpha (tmp)) << 16;
727         table->val[2*i] |= FOX_div_alpha (tmp) << 8;
728         table->val[2*i] |= FOX_div_alpha (FOX_div_alpha (tmp));
729         table->val[2*i+1] = tmp << 24;
730         table->val[2*i+1] |= (FOX_times_alpha (tmp) ^ tmp) << 16;
731         table->val[2*i+1] |= (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp))) << 8;
732         table->val[2*i+1] |= FOX_times_alpha (tmp);
733     }
734     break;
735
736 case FOX128_TABLE_SIGMA8_MU8_ID5:
737     for (i = 0; i < 256; i++) {
738         tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
739         table->val[2*i] = tmp << 24;
740         table->val[2*i] |= FOX_div_alpha (tmp) << 16;
741         table->val[2*i] |= FOX_div_alpha (FOX_div_alpha (tmp)) << 8;
742         table->val[2*i] |= tmp;
743         table->val[2*i+1] = (FOX_times_alpha (tmp) ^ tmp) << 24;
```

```

744         table->val[2*i+1] |= (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp))) << 16;
745         table->val[2*i+1] |= FOX_times_alpha (tmp) << 8;
746         table->val[2*i+1] |= FOX_times_alpha (FOX_times_alpha (tmp));
747     }
748     break;
749
750     case FOX128_TABLE_SIGMA8_MU8_ID6:
751     for (i = 0; i < 256; i++) {
752         tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
753         table->val[2*i] = tmp << 24;
754         table->val[2*i] |= FOX_div_alpha (FOX_div_alpha (tmp)) << 16;
755         table->val[2*i] |= tmp << 8;
756         table->val[2*i] |= (FOX_times_alpha (tmp) ^ tmp);
757         table->val[2*i+1] = (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp))) << 24;
758         table->val[2*i+1] |= FOX_times_alpha (tmp) << 16;
759         table->val[2*i+1] |= FOX_times_alpha (FOX_times_alpha (tmp)) << 8;
760         table->val[2*i+1] |= FOX_div_alpha (tmp);
761     }
762     break;
763
764     case FOX128_TABLE_SIGMA8_MU8_ID7:
765     for (i = 0; i < 256; i++) {
766         tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
767         table->val[2*i] = (FOX_times_alpha (tmp) ^ tmp) << 24;
768         table->val[2*i] |= tmp << 16;
769         table->val[2*i] |= tmp << 8;
770         table->val[2*i] |= tmp;
771         table->val[2*i+1] = tmp << 24;
772         table->val[2*i+1] |= tmp << 16;
773         table->val[2*i+1] |= tmp << 8;
774         table->val[2*i+1] |= tmp;
775     }
776     break;
777
778     case FOX64_TABLE_SIGMA4_ID0:
779     case FOX128_TABLE_SIGMA8_ID0:
780     for (i = 0; i < 256; i++) {
781         table->val[i] = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3) << 24;
782     }
783     break;
784
785     case FOX64_TABLE_SIGMA4_ID1:
786     case FOX128_TABLE_SIGMA8_ID1:
787     for (i = 0; i < 256; i++) {
788         table->val[i] = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3) << 16;
789     }
790     break;
791
792     case FOX64_TABLE_SIGMA4_ID2:
793     case FOX128_TABLE_SIGMA8_ID2:
794     for (i = 0; i < 256; i++) {
795         table->val[i] = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3) << 8;
796     }
797     break;
798
799     case FOX64_TABLE_SIGMA4_ID3:
800     case FOX128_TABLE_SIGMA8_ID3:
801     for (i = 0; i < 256; i++) {

```

```
802         table->val[i] = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
803     }
804     break;
805
806     default:
807         fprintf (stderr, FOX_ERROR_UNKNOWN_TABLE_ID);
808         goto error_label;
809 }
810
811 *ptr = table;
812
813 return 0;
814
815 error_label:
816     fprintf (stderr, FOX_ERROR_TABLE_INITIALIZATION);
817     FOX_clean_table (table);
818
819     return -1;
820 }
821
822 void FOX_clean_table (FOX_table table)
823 {
824     if (table != NULL) {
825         if (table->val != NULL) {
826             free (memset (table->val, 0x00, table->size_bytes));
827         }
828         free (memset (table, 0x00, sizeof (FOX_table)));
829     }
830 }
831
832 uint32 FOX_eval_sbox (const uint32 x, const uint8 *s1,
833                     const uint8 *s2, const uint8 *s3)
834 {
835     uint8 l, r, ll, lr, state;
836
837     assert ( (x <= 0xFF) && (s1 != NULL) && (s2 != NULL) && (s3 != NULL) );
838
839     l = (x & 0xF0) >> 4;
840     r = (x & 0x0F);
841
842     /* Stage 1 */
843
844     state = s1[l ^ r];
845     l ^= state;
846     r ^= state;
847
848     ll = (l & 0xC) >> 2;
849     lr = (l & 0x3);
850
851     l = (lr << 2) | (ll ^ lr);
852
853     /* Stage 2 */
854
855     state = s2[l ^ r];
856     l ^= state;
857     r ^= state;
858
859     ll = (l & 0xC) >> 2;
```

```

860     lr = (l & 0x3);
861
862     l = (lr << 2) | (ll ^ lr);
863
864     /* Stage 3 */
865
866     state = s3[l ^ r];
867     l ^= state;
868     r ^= state;
869
870     ll = (l & 0xC) >> 2;
871     lr = (l & 0x3);
872
873     l = (lr << 2) | (ll ^ lr);
874
875     /* Saving of the value */
876
877     return (uint32)((l << 4) | r);
878 }

```

## B.9 File fox64.h

```

1  /*****
2  /* FOX project / Reference implementation
3  /* Pascal Junod <pascal@junod.info>
4  /*
5  /* $Id: fox64.h,v 1.4 2003/09/24 11:19:42 pjunod Exp $
6  *****/
7
8  #ifndef _FOX64_H_
9  #define _FOX64_H_
10
11 #include "fox_portable.h"
12 #include "fox_ctx.h"
13
14 #define FOX64_MODE_ENCRYPT      0x0
15 #define FOX64_MODE_DECRYPT     0x1
16
17 #define FOX64_NUMBER_ROUNDS_MIN      12
18 #define FOX64_NUMBER_ROUNDS_SECURE  16
19
20 #define FOX64_encrypt(p, k, ctx) FOX64_process((p), (k), (ctx), FOX64_MODE_ENCRYPT)
21 #define FOX64_decrypt(c, k, ctx) FOX64_process((c), (k), (ctx), FOX64_MODE_DECRYPT)
22
23 extern int FOX64_process (uint32 *, const FOX_key, const FOX64_ctx, const FOX_mode);
24
25 void FOX_lmor64 (uint32 *, const uint32 *, const FOX64_ctx);
26 void FOX_lmids64 (uint32 *, const uint32 *, const FOX64_ctx);
27 void FOX_lmio64 (uint32 *, const uint32 *, const FOX64_ctx);
28 void FOX_f32 (uint32 *, const uint32 *, const FOX64_ctx);
29
30 #endif /* _FOX64_H_ */

```

## B.10 File fox64.c

```
1  /*****
2  /* FOX project / Reference implementation */
3  /* Pascal Junod <pascal@junod.info> */
4  /*
5  /* $Id: fox64.c,v 1.4 2003/09/24 11:17:48 pjunod Exp $
6  /*****
7
8  #include <assert.h>
9  #include <stdlib.h>
10 #include <stdio.h>
11
12 #include "fox_portable.h"
13 #include "fox_error.h"
14 #include "fox_ctx.h"
15 #include "fox64.h"
16
17 int FOX64_process (uint32 *data, const FOX_key k,
18                  const FOX64_ctx ctx, const FOX_mode mode)
19 {
20     int r;
21     uint32 input[2];
22
23     assert (data != NULL);
24     assert (k != NULL);
25     assert (ctx != NULL);
26
27     assert (k->rounds >= FOX64_NUMBER_ROUNDS_MIN);
28
29     input[0] = data[0];
30     input[1] = data[1];
31
32     switch (mode) {
33
34         case FOX64_MODE_ENCRYPT:
35             for (r = 0; r < k->rounds - 1; r++) {
36                 FOX_lmor64 (input, k->exp_key + (r * 2), ctx);
37             }
38             FOX_lmld64 (input, k->exp_key + (k->rounds-1) * 2, ctx);
39             break;
40
41         case FOX64_MODE_DECRYPT:
42             for (r = k->rounds - 1; r > 0; r--) {
43                 FOX_lmio64 (input, k->exp_key + (r * 2), ctx);
44             }
45             FOX_lmld64 (input, k->exp_key, ctx);
46             break;
47
48         default:
49             fprintf (stderr, FOX_ERROR_UNKNOWN_MODE);
50             return -1;
51     }
52
53     data[0] = input[0];
54     data[1] = input[1];
55
56     return 0;
```



```
57 }
58
59 void FOX_lmor64 (uint32 *data, const uint32 *key,
60                const FOX64_ctx ctx)
61 {
62     uint32 tmp[2], f;
63
64     tmp[0] = data[0];
65     tmp[1] = data[1];
66
67     f = tmp[0] ^ tmp[1];
68     FOX_f32 (&f, key, ctx);
69     tmp[0] ^= f;
70     tmp[1] ^= f;
71     FOX_or (tmp);
72
73     data[0] = tmp[0];
74     data[1] = tmp[1];
75 }
76
77 void FOX_lmid64 (uint32 *data, const uint32 *key,
78                const FOX64_ctx ctx)
79 {
80     uint32 tmp[2], f;
81
82     tmp[0] = data[0];
83     tmp[1] = data[1];
84
85     f = tmp[0] ^ tmp[1];
86     FOX_f32 (&f, key, ctx);
87     tmp[0] ^= f;
88     tmp[1] ^= f;
89
90     data[0] = tmp[0];
91     data[1] = tmp[1];
92 }
93
94 void FOX_lmio64 (uint32 *data, const uint32 *key,
95                const FOX64_ctx ctx)
96 {
97     uint32 tmp[2], f;
98
99     tmp[0] = data[0];
100    tmp[1] = data[1];
101
102    f = tmp[0] ^ tmp[1];
103    FOX_f32 (&f, key, ctx);
104    tmp[0] ^= f;
105    tmp[1] ^= f;
106    FOX_io (tmp);
107
108    data[0] = tmp[0];
109    data[1] = tmp[1];
110 }
111
112 void FOX_f32 (uint32 *data, const uint32 *key,
113              const FOX64_ctx ctx)
114 {
```

```

115     uint32 i, o;
116
117     i = *data;
118
119     i ^= key[0];
120
121     o = ctx->sigma4_mu4_0->val[(i & 0xFF000000) >> 24];
122     o ^= ctx->sigma4_mu4_1->val[(i & 0x00FF0000) >> 16];
123     o ^= ctx->sigma4_mu4_2->val[(i & 0x0000FF00) >> 8];
124     o ^= ctx->sigma4_mu4_3->val[(i & 0x000000FF)];
125
126     o ^= key[1];
127
128     i = ctx->sigma4_0->val[(o & 0xFF000000) >> 24];
129     i ^= ctx->sigma4_1->val[(o & 0x00FF0000) >> 16];
130     i ^= ctx->sigma4_2->val[(o & 0x0000FF00) >> 8];
131     i ^= ctx->sigma4_3->val[(o & 0x000000FF)];
132
133     *data = i ^ key[0];
134 }

```

## B.11 File fox128.h

```

1  /*****
2  /* FOX project / Reference implementation
3  /* Pascal Junod <pascal@junod.info>
4  /*
5  /* $Id: fox128.h,v 1.4 2003/09/24 11:19:20 pjunod Exp $
6  /*****
7
8  #ifndef _FOX128_H_
9  #define _FOX128_H_
10
11 #include "fox_portable.h"
12 #include "fox_ctx.h"
13
14 #define FOX128_MODE_ENCRYPT      0x0
15 #define FOX128_MODE_DECRYPT     0x1
16
17 #define FOX128_NUMBER_ROUNDS_MIN      12
18 #define FOX128_NUMBER_ROUNDS_SECURE  16
19
20 #define FOX128_encrypt(p, k, ctx) FOX128_process((p), (k), (ctx), FOX128_MODE_ENCRYPT)
21 #define FOX128_decrypt(c, k, ctx) FOX128_process((c), (k), (ctx), FOX128_MODE_DECRYPT)
22
23 extern int FOX128_process (uint32 *, const FOX_key, const FOX128_ctx, const FOX_mode);
24
25 void FOX_elmor128 (uint32 *, const uint32 *, const FOX128_ctx);
26 void FOX_elmid128 (uint32 *, const uint32 *, const FOX128_ctx);
27 void FOX_elmio128 (uint32 *, const uint32 *, const FOX128_ctx);
28
29 void FOX_f64 (uint32 *, const uint32 *, const FOX128_ctx);
30
31 #endif /* _FOX128_H_ */

```

## B.12 File fox128.c

```
1  /*****  
2  /* FOX project / Reference implementation */  
3  /* Pascal Junod <pascal@junod.info> */  
4  /* */  
5  /* $Id: fox128.c,v 1.5 2003/09/24 11:17:30 pjunod Exp $ */  
6  *****/  
7  
8  #include <assert.h>  
9  #include <stdlib.h>  
10 #include <stdio.h>  
11  
12 #include "fox_portable.h"  
13 #include "fox_error.h"  
14 #include "fox_ctx.h"  
15 #include "fox128.h"  
16  
17 int FOX128_process (uint32 *data, const FOX_key k,  
18                   const FOX128_ctx ctx, const FOX_mode mode)  
19 {  
20     int r;  
21     uint32 input[4];  
22  
23     assert (data != NULL);  
24     assert (k != NULL);  
25     assert (ctx != NULL);  
26  
27     assert (k->rounds >= FOX128_NUMBER_ROUNDS_MIN);  
28  
29     input[0] = data[0];  
30     input[1] = data[1];  
31     input[2] = data[2];  
32     input[3] = data[3];  
33  
34     switch (mode) {  
35  
36         case FOX128_MODE_ENCRYPT:  
37             for (r = 0; r < k->rounds - 1; r++) {  
38                 FOX_elmor128 (input, k->exp_key + (r * 4), ctx);  
39             }  
40             FOX_elmid128 (input, k->exp_key + (k->rounds - 1) * 4, ctx);  
41             break;  
42  
43         case FOX128_MODE_DECRYPT:  
44             for (r = k->rounds - 1; r > 0; r--) {  
45                 FOX_elmio128 (input, k->exp_key + (r * 4), ctx);  
46             }  
47             FOX_elmid128 (input, k->exp_key, ctx);  
48             break;  
49  
50         default:  
51             fprintf (stderr, FOX_ERROR_UNKNOWN_MODE);  
52             return -1;  
53     }  
54  
55     data[0] = input[0];  
56     data[1] = input[1];
```

```
57     data[2] = input[2];
58     data[3] = input[3];
59
60     return 0;
61 }
62
63 void FOX_elmor128 (uint32 *data, const uint32 *key,
64                  const FOX128_ctx ctx)
65 {
66     uint32 tmp[4], f[2];
67
68     tmp[0] = data[0];
69     tmp[1] = data[1];
70     tmp[2] = data[2];
71     tmp[3] = data[3];
72
73     f[0] = tmp[0] ^ tmp[1];
74     f[1] = tmp[2] ^ tmp[3];
75
76     FOX_f64 (f, key, ctx);
77
78     tmp[0] ^= f[0];
79     tmp[1] ^= f[0];
80     tmp[2] ^= f[1];
81     tmp[3] ^= f[1];
82
83     FOX_or (tmp);
84     FOX_or (tmp + 2);
85
86     data[0] = tmp[0];
87     data[1] = tmp[1];
88     data[2] = tmp[2];
89     data[3] = tmp[3];
90 }
91
92 void FOX_elmid128 (uint32 *data, const uint32 *key,
93                  const FOX128_ctx ctx)
94 {
95     uint32 tmp[4], f[2];
96
97     tmp[0] = data[0];
98     tmp[1] = data[1];
99     tmp[2] = data[2];
100    tmp[3] = data[3];
101
102    f[0] = tmp[0] ^ tmp[1];
103    f[1] = tmp[2] ^ tmp[3];
104
105    FOX_f64 (f, key, ctx);
106
107    tmp[0] ^= f[0];
108    tmp[1] ^= f[0];
109    tmp[2] ^= f[1];
110    tmp[3] ^= f[1];
111
112    data[0] = tmp[0];
113    data[1] = tmp[1];
114    data[2] = tmp[2];
```

```

115     data[3] = tmp[3];
116 }
117
118 void FOX_elmio128 (uint32 *data, const uint32 *key,
119                  const FOX128_ctx ctx)
120 {
121     uint32 tmp[4], f[2];
122
123     tmp[0] = data[0];
124     tmp[1] = data[1];
125     tmp[2] = data[2];
126     tmp[3] = data[3];
127
128     f[0] = tmp[0] ^ tmp[1];
129     f[1] = tmp[2] ^ tmp[3];
130
131     FOX_f64 (f, key, ctx);
132
133     tmp[0] ^= f[0];
134     tmp[1] ^= f[0];
135     tmp[2] ^= f[1];
136     tmp[3] ^= f[1];
137
138     FOX_io (tmp);
139     FOX_io (tmp + 2);
140
141     data[0] = tmp[0];
142     data[1] = tmp[1];
143     data[2] = tmp[2];
144     data[3] = tmp[3];
145 }
146
147 void FOX_f64 (uint32 *data, const uint32 *key,
148              const FOX128_ctx ctx)
149 {
150     uint32 i[2], o[2];
151
152     i[0] = data[0];
153     i[1] = data[1];
154
155     i[0] ^= key[0];
156     i[1] ^= key[1];
157
158     o[0] = ctx->sigma8_mu8_0->val[(i[0] & 0xFF000000) >> 23];
159     o[1] = ctx->sigma8_mu8_0->val[((i[0] & 0xFF000000) >> 23) + 1];
160     o[0] ^= ctx->sigma8_mu8_1->val[(i[0] & 0x00FF0000) >> 15];
161     o[1] ^= ctx->sigma8_mu8_1->val[((i[0] & 0x00FF0000) >> 15) + 1];
162     o[0] ^= ctx->sigma8_mu8_2->val[(i[0] & 0x0000FF00) >> 7];
163     o[1] ^= ctx->sigma8_mu8_2->val[((i[0] & 0x0000FF00) >> 7) + 1];
164     o[0] ^= ctx->sigma8_mu8_3->val[(i[0] & 0x000000FF) << 1];
165     o[1] ^= ctx->sigma8_mu8_3->val[((i[0] & 0x000000FF) << 1) + 1];
166
167     o[0] ^= ctx->sigma8_mu8_4->val[(i[1] & 0xFF000000) >> 23];
168     o[1] ^= ctx->sigma8_mu8_4->val[((i[1] & 0xFF000000) >> 23) + 1];
169     o[0] ^= ctx->sigma8_mu8_5->val[(i[1] & 0x00FF0000) >> 15];
170     o[1] ^= ctx->sigma8_mu8_5->val[((i[1] & 0x00FF0000) >> 15) + 1];
171     o[0] ^= ctx->sigma8_mu8_6->val[(i[1] & 0x0000FF00) >> 7];
172     o[1] ^= ctx->sigma8_mu8_6->val[((i[1] & 0x0000FF00) >> 7) + 1];

```

```

173     o[0] ^= ctx->sigma8_mu8_7->val[(i[1] & 0x000000FF) << 1];
174     o[1] ^= ctx->sigma8_mu8_7->val[((i[1] & 0x000000FF) << 1) + 1];
175
176     o[0] ^= key[2];
177     o[1] ^= key[3];
178
179     i[0] = ctx->sigma8_0->val[(o[0] & 0xFF000000) >> 24];
180     i[0] ^= ctx->sigma8_1->val[(o[0] & 0x00FF0000) >> 16];
181     i[0] ^= ctx->sigma8_2->val[(o[0] & 0x0000FF00) >> 8];
182     i[0] ^= ctx->sigma8_3->val[(o[0] & 0x000000FF)];
183
184     i[1] = ctx->sigma8_0->val[(o[1] & 0xFF000000) >> 24];
185     i[1] ^= ctx->sigma8_1->val[(o[1] & 0x00FF0000) >> 16];
186     i[1] ^= ctx->sigma8_2->val[(o[1] & 0x0000FF00) >> 8];
187     i[1] ^= ctx->sigma8_3->val[(o[1] & 0x000000FF)];
188
189     data[0] = i[0] ^ key[0];
190     data[1] = i[1] ^ key[1];
191 }

```

## B.13 File fox\_util.h

```

1  /*****
2  /* FOX project / Reference implementation
3  /* Pascal Junod <pascal@junod.info>
4  /*
5  /* $Id: fox_util.h,v 1.3 2003/06/11 09:58:58 pjunod Exp $
6  *****/
7
8  #ifndef _FOX_UTIL_H_
9  #define _FOX_UTIL_H_
10
11 #include "fox_portable.h"
12
13 int fox64_128_16_test (const uint32 *p, const uint32 *k);
14 int fox64_192_16_test (const uint32 *p, const uint32 *k);
15 int fox64_256_16_test (const uint32 *p, const uint32 *k);
16
17 int fox128_128_16_test (const uint32 *p, const uint32 *k);
18 int fox128_192_16_test (const uint32 *p, const uint32 *k);
19 int fox128_256_16_test (const uint32 *p, const uint32 *k);
20
21 #endif /* _FOX_UTIL_H_ */

```

## B.14 File fox\_util.c

```

1  /*****
2  /* FOX project / Reference implementation
3  /* Pascal Junod <pascal@junod.info>
4  /*
5  /* $Id: fox_util.c,v 1.3 2003/06/11 09:59:28 pjunod Exp $
6  *****/
7
8  #include <stdio.h>
9  #include <stdlib.h>

```

```
10 #include <string.h>
11
12 #include "fox_portable.h"
13 #include "fox_error.h"
14 #include "fox64.h"
15 #include "fox128.h"
16 #include "fox_ctx.h"
17 #include "fox_util.h"
18
19 const uint32 p64[2] = {0x01234567, 0x89ABCDEF};
20
21 const uint32 p128[4] = {0x01234567, 0x89ABCDEF, 0xFEDCBA98, 0x76543210};
22
23 const uint32 k[8] = {0x00112233, 0x44556677, 0x8899AABB, 0xCCDDEEFF,
24                     0xFFEEDDCC, 0xBBAA9988, 0x77665544, 0x33221100 };
25
26
27 int main ()
28 {
29     int return_value = EXIT_SUCCESS;
30
31     fprintf (stdout, "\n\nFOX test vectors generator");
32     fprintf (stdout, "\n-----\n\n");
33
34     /* FOX64 test vectors */
35     if (fox64_128_16_test (p64, k)) {
36         fprintf (stdout, "\nFatal error_exiting!\n");
37         return_value = EXIT_FAILURE;
38         goto error_label;
39     }
40     if (fox64_192_16_test (p64, k)) {
41         fprintf (stdout, "\nFatal error_exiting!\n");
42         return_value = EXIT_FAILURE;
43         goto error_label;
44     }
45     if (fox64_256_16_test (p64, k)) {
46         fprintf (stdout, "\nFatal error_exiting!\n");
47         return_value = EXIT_FAILURE;
48         goto error_label;
49     }
50
51     /* FOX128 test vectors */
52     if (fox128_128_16_test (p128, k)) {
53         fprintf (stdout, "\nFatal error_exiting!\n");
54         return_value = EXIT_FAILURE;
55         goto error_label;
56     }
57     if (fox128_192_16_test (p128, k)) {
58         fprintf (stdout, "\nFatal error_exiting!\n");
59         return_value = EXIT_FAILURE;
60         goto error_label;
61     }
62     if (fox128_256_16_test (p128, k)) {
63         fprintf (stdout, "\nFatal error_exiting!\n");
64         return_value = EXIT_FAILURE;
65         goto error_label;
66     }
67 }
```

```
68
69     error_label:
70
71     return return_value;
72 }
73
74 int fox64_128_16_test (const uint32 *p, const uint32 *k)
75 {
76     FOX64_ctx ctx;
77     FOX_key key;
78     uint32 c[2];
79     int i, return_value = EXIT_SUCCESS;
80
81     if (FOX64_init_ctx (&ctx)) {
82         fprintf (stderr, "\nFatal error...exiting!\n");
83         return_value = EXIT_FAILURE;
84         goto error_label;
85     }
86     fprintf (stdout, "\n\nFOX64/16/128 key      : ");
87     for (i = 0; i < 4; i++) {
88         fprintf (stdout, "%08X ", k[i]);
89     }
90     fprintf (stdout, "\nFOX64/16/128 message  : ");
91     for (i = 0; i < 2; i++) {
92         fprintf (stdout, "%08X ", p[i]);
93     }
94     if (FOX64_init_key (&key, ctx, k, 128, 16)) {
95         fprintf (stderr, "\nFatal error...exiting!\n");
96         return_value = EXIT_FAILURE;
97         goto error_label;
98     }
99     memcpy (c, p, 8);
100    FOX64_encrypt (c, key, ctx);
101    fprintf (stdout, "\nFOX64/16/128 ciphertext : ");
102    for (i = 0; i < 2; i++) {
103        fprintf (stdout, "%08X ", c[i]);
104    }
105    FOX64_decrypt (c, key, ctx);
106    fprintf (stdout, "\nFOX64/16/128 message  : ");
107    for (i = 0; i < 2; i++) {
108        fprintf (stdout, "%08X ", c[i]);
109    }
110    fprintf (stdout, "\n\n");
111
112    error_label:
113        FOX64_clean_ctx (ctx);
114        FOX64_clean_key (key);
115
116    return return_value;
117 }
118
119 int fox64_192_16_test (const uint32 *p, const uint32 *k)
120 {
121     FOX64_ctx ctx;
122     FOX_key key;
123     uint32 c[2];
124     int i, return_value = EXIT_SUCCESS;
125
```



```
126     if (FOX64_init_ctx (&ctx)) {
127         fprintf (stderr, "\nFatal error...exiting!\n");
128         return_value = EXIT_FAILURE;
129         goto error_label;
130     }
131     fprintf (stdout, "\n\nFOX64/16/192 key      : ");
132     for (i = 0; i < 6; i++) {
133         fprintf (stdout, "%08X ", k[i]);
134     }
135     fprintf (stdout, "\nFOX64/16/192 message  : ");
136     for (i = 0; i < 2; i++) {
137         fprintf (stdout, "%08X ", p[i]);
138     }
139     if (FOX64_init_key (&key, ctx, k, 192, 16)) {
140         fprintf (stderr, "\nFatal error...exiting!\n");
141         return_value = EXIT_FAILURE;
142         goto error_label;
143     }
144     memcpy (c, p, 8);
145     FOX64_encrypt (c, key, ctx);
146     fprintf (stdout, "\nFOX64/16/192 ciphertext : ");
147     for (i = 0; i < 2; i++) {
148         fprintf (stdout, "%08X ", c[i]);
149     }
150     FOX64_decrypt (c, key, ctx);
151     fprintf (stdout, "\nFOX64/16/192 message  : ");
152     for (i = 0; i < 2; i++) {
153         fprintf (stdout, "%08X ", c[i]);
154     }
155     fprintf (stdout, "\n\n");
156
157     error_label:
158     FOX64_clean_ctx (ctx);
159     FOX64_clean_key (key);
160
161     return return_value;
162 }
163 int fox64_256_16_test (const uint32 *p, const uint32 *k)
164 {
165     FOX64_ctx ctx;
166     FOX_key key;
167     uint32 c[2];
168     int i, return_value = EXIT_SUCCESS;
169
170     if (FOX64_init_ctx (&ctx)) {
171         fprintf (stderr, "\nFatal error...exiting!\n");
172         return_value = EXIT_FAILURE;
173         goto error_label;
174     }
175     fprintf (stdout, "\n\nFOX64/16/256 key      : ");
176     for (i = 0; i < 8; i++) {
177         fprintf (stdout, "%08X ", k[i]);
178     }
179     fprintf (stdout, "\nFOX64/16/256 message  : ");
180     for (i = 0; i < 2; i++) {
181         fprintf (stdout, "%08X ", p[i]);
182     }
183     if (FOX64_init_key (&key, ctx, k, 256, 16)) {
```

```
184         fprintf (stderr, "\nFatal error...exiting!\n");
185         return_value = EXIT_FAILURE;
186         goto error_label;
187     }
188     memcpy (c, p, 8);
189     FOX64_encrypt (c, key, ctx);
190     fprintf (stdout, "\nFOX64/16/256 ciphertext : ");
191     for (i = 0; i < 2; i++) {
192         fprintf (stdout, "%08X ", c[i]);
193     }
194     FOX64_decrypt (c, key, ctx);
195     fprintf (stdout, "\nFOX64/16/256 message   : ");
196     for (i = 0; i < 2; i++) {
197         fprintf (stdout, "%08X ", c[i]);
198     }
199     fprintf (stdout, "\n\n");
200
201     error_label:
202     FOX64_clean_ctx (ctx);
203     FOX64_clean_key (key);
204
205     return return_value;
206 }
207
208 int fox128_128_16_test (const uint32 *p, const uint32 *k)
209 {
210     FOX128_ctx ctx;
211     FOX_key key;
212     uint32 c[4];
213     int i, return_value = EXIT_SUCCESS;
214
215     if (FOX128_init_ctx (&ctx)) {
216         fprintf (stderr, "\nFatal error...exiting!\n");
217         return_value = EXIT_FAILURE;
218         goto error_label;
219     }
220     fprintf (stdout, "\n\nFOX128/16/128 key       : ");
221     for (i = 0; i < 4; i++) {
222         fprintf (stdout, "%08X ", k[i]);
223     }
224     fprintf (stdout, "\n\nFOX128/16/128 message   : ");
225     for (i = 0; i < 4; i++) {
226         fprintf (stdout, "%08X ", p[i]);
227     }
228     if (FOX128_init_key (&key, ctx, k, 128, 16)) {
229         fprintf (stderr, "\nFatal error...exiting!\n");
230         return_value = EXIT_FAILURE;
231         goto error_label;
232     }
233     memcpy (c, p, 16);
234     FOX128_encrypt (c, key, ctx);
235     fprintf (stdout, "\n\nFOX128/16/128 ciphertext : ");
236     for (i = 0; i < 4; i++) {
237         fprintf (stdout, "%08X ", c[i]);
238     }
239     FOX128_decrypt (c, key, ctx);
240     fprintf (stdout, "\n\nFOX128/16/128 message   : ");
241     for (i = 0; i < 4; i++) {
```

```
242     fprintf (stdout, "%08X ", c[i]);
243     }
244     fprintf (stdout, "\n\n");
245
246     error_label:
247     FOX128_clean_ctx (ctx);
248     FOX128_clean_key (key);
249
250     return return_value;
251 }
252
253 int fox128_192_16_test (const uint32 *p, const uint32 *k)
254 {
255     FOX128_ctx ctx;
256     FOX_key key;
257     uint32 c[4];
258     int i, return_value = EXIT_SUCCESS;
259
260     if (FOX128_init_ctx (&ctx)) {
261         fprintf (stderr, "\nFatal error...exiting!\n");
262         return_value = EXIT_FAILURE;
263         goto error_label;
264     }
265     fprintf (stdout, "\n\nFOX128/16/192 key      : ");
266     for (i = 0; i < 4; i++) {
267         fprintf (stdout, "%08X ", k[i]);
268     }
269     fprintf (stdout, "\n\nFOX128/16/192 message  : ");
270     for (i = 0; i < 4; i++) {
271         fprintf (stdout, "%08X ", p[i]);
272     }
273     if (FOX128_init_key (&key, ctx, k, 192, 16)) {
274         fprintf (stderr, "\nFatal error...exiting!\n");
275         return_value = EXIT_FAILURE;
276         goto error_label;
277     }
278     memcpy (c, p, 16);
279     FOX128_encrypt (c, key, ctx);
280     fprintf (stdout, "\n\nFOX128/16/192 ciphertext : ");
281     for (i = 0; i < 4; i++) {
282         fprintf (stdout, "%08X ", c[i]);
283     }
284     FOX128_decrypt (c, key, ctx);
285     fprintf (stdout, "\n\nFOX128/16/192 message  : ");
286     for (i = 0; i < 4; i++) {
287         fprintf (stdout, "%08X ", c[i]);
288     }
289     fprintf (stdout, "\n\n");
290
291     error_label:
292     FOX128_clean_ctx (ctx);
293     FOX128_clean_key (key);
294
295     return return_value;
296 }
297
298 int fox128_256_16_test (const uint32 *p, const uint32 *k)
299 {
```

```
300     FOX128_ctx ctx;
301     FOX_key key;
302     uint32 c[4];
303     int i, return_value = EXIT_SUCCESS;
304
305     if (FOX128_init_ctx (&ctx)) {
306         fprintf (stderr, "\nFatal error...exiting!\n");
307         return_value = EXIT_FAILURE;
308         goto error_label;
309     }
310     fprintf (stdout, "\n\nFOX128/16/256 key      : ");
311     for (i = 0; i < 8; i++) {
312         fprintf (stdout, "%08X ", k[i]);
313     }
314     fprintf (stdout, "\nFOX128/16/256 message   : ");
315     for (i = 0; i < 4; i++) {
316         fprintf (stdout, "%08X ", p[i]);
317     }
318     if (FOX128_init_key (&key, ctx, k, 256, 16)) {
319         fprintf (stderr, "\nFatal error...exiting!\n");
320         return_value = EXIT_FAILURE;
321         goto error_label;
322     }
323     memcpy (c, p, 16);
324     FOX128_encrypt (c, key, ctx);
325     fprintf (stdout, "\nFOX128/16/256 ciphertext : ");
326     for (i = 0; i < 4; i++) {
327         fprintf (stdout, "%08X ", c[i]);
328     }
329     FOX128_decrypt (c, key, ctx);
330     fprintf (stdout, "\nFOX128/16/256 message   : ");
331     for (i = 0; i < 4; i++) {
332         fprintf (stdout, "%08X ", c[i]);
333     }
334     fprintf (stdout, "\n\n");
335
336     error_label:
337     FOX128_clean_ctx (ctx);
338     FOX128_clean_key (key);
339
340     return return_value;
341 }
```