FOX Specifications
Version 1.2

Pascal Junod and Serge Vaudenay
http://crypto.junod.info, http://lasecwww.epfl.ch
pascal@junod.info, serge.vaudenay@epfl.ch

April 15, 2005

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 (`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`) for more information about them. This document[1] is organized as follows: in §1, the conventions and mathematical 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. Finally, 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.

[1]This document is the extended version of [JV04a]; it superseeds EPFL technical reports IC/2003/82 and IC/2004/75 entitled respectively "FOX Specifications Version 1.0" and "FOX Specifications Version 1.1".

Contents

Name	Block size (in bits)	Key size (in bits)	Rounds number
FOX64	64	128	16
FOX128	128	256	16
FOX64/k/r	64	k	r
FOX128/k/r	128	k	r

Figure 1: Members of the FOX family

1 Notations

The purpose of this section is to define the mathematical notations, conventions and symbols used throughout this document.

1.1 The FOX Family

The family consists in two main block cipher designs, the first one having a 64-bit blocksize and the other one a 128-bit blocksize. Each design allows a *variable number of rounds* and a *variable key size* up to 256 bits. The different members of the FOX family are listed in Fig. 1. The following conditions *must* hold in the case of FOX64/k/r and FOX128/k/r: the number of rounds r must satisfy $12 \leq r \leq 255$, while the key length k must satisfy $0 \leq k \leq 256$, with k multiple of 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

Fig. 2 is a list of the mathematical operations used throughout this document together with their meanings. Note that the GF $\left(2^8\right)$ representation is defined in §1.6.

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 $_{[a...n]}$ (i.e. $x_{[a...b]}$) indicates the bit subset of the variable x beginning at position a (inclusive) and ending at position b (inclusive).

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	$\texttt{0xABCD} \oplus \texttt{0x1234} = \texttt{0xB9F9}$
$a \wedge b$	Bitwise AND	$\texttt{0xABCD} \wedge \texttt{0x1234} = \texttt{0x0204}$
$a \vee b$	Bitwise OR	$\texttt{0xABCD} \vee \texttt{0x1234} = \texttt{0xBBFD}$
$a \ll n$	Logical left shift of n positions	$\texttt{0x03} \ll 1 = \texttt{0x06}$
$a \gg n$	Logical right shift of n positions	$\texttt{0x03} \gg 1 = \texttt{0x01}$
$\overline{a}$	Logical negation	$\overline{\texttt{0xA}} = \texttt{0x5}$
$a\|\|b$	Concatenation	$\texttt{0xABCD}\|\|\texttt{0x1234} = \texttt{0xABCD1234}$
$a \oplus b$	Addition in GF(2^8)	$\texttt{0x02} \oplus \texttt{0x06} = \texttt{0x04}$
$a \cdot b$	Multiplication in GF(2^8)	$\texttt{0x02} \cdot \texttt{0x60} = \texttt{0xC0}$

Figure 2: Mathematical Operations

- 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 ℓ 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, $x = x_{\mathsf{ll}}||x_{\mathsf{lr}}||x_{\mathsf{rl}}||x_{\mathsf{rr}}$.
- 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

$$\begin{aligned} q_{(128)} &= r_{0(64)}||r_{1(64)} \\ &= s_{0(32)}||s_{1(32)}||s_{2(32)}||s_{3(32)} \\ &= t_{0(8)}||t_{1(8)}||\ldots||t_{14(8)}||t_{15(8)} \\ &= u_{0(1)}||u_{1(1)}||\ldots||u_{126(1)}||u_{127(1)} \end{aligned}$$

1.6 Finite Field 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 GF (2^8). We describe now the *representation* of GF (2^8) used in the FOX definition. Let be the following irreducible polynomial $P(\alpha)$ over GF $(2) = \{0, 1\}$:

$$P(\alpha) = \alpha^8 + \alpha^7 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + 1 \tag{1}$$

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

$$s = s_{0(1)}||s_{1(1)}||s_{2(1)}||s_{3(1)}||s_{4(1)}||s_{5(1)}||s_{6(1)}||s_{7(1)}$$

The corresponding field element is

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

1.6.1 Addition in $\mathrm{GF}\left(2^8\right)$

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

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

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

1.6.2 Multiplication in $\mathrm{GF}\left(2^8\right)$

The multiplication in $\mathrm{GF}\left(2^8\right)$, denoted "$\cdot$", is the usual multiplication of polynomials where the result is taken modulo the polynomial defined in Eq. (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,

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

2 Description

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

This chapter is organized as follows: in §2.1.1, the high-level structure of $\mathsf{FOX64}/k/r$, which is a *Lai-Massey scheme*, is formally described, together with the encryption and decryption operations. In §2.1.2, the same is done for $\mathsf{FOX128}/k/r$, which is built on an *Extended Lai-Massey scheme.* In §2.2, the *internal functions* f32 and f64 used in both algorithms are formally defined, together with their building blocks. Finally, in §2.3, the key-schedule algorithm is described.

2.1 High-Level Structure

In this part, we describe the skeleton and the encryption/decryption processes for FOX64 and FOX128. For this purpose, we will follow a top-down approach.

2.1.1 $\mathsf{FOX64}/k/r$ Skeleton

The 64-bit version of FOX is the $(r-1)$-times iteration of a round function denoted lmor64, followed by the application of a slightly modified version of lmor64, named lmid64. lmio64 is a function used during the decryption operation. Formally, lmor64, lmio64 and lmid64 take all a 64-bit input $x_{(64)}$, a 64-bit round key $rk_{(64)}$ and return a 64-bit output $y_{(64)}$:

$$\mathsf{lmor64}, \mathsf{lmio64}, \mathsf{lmid64} : \left\{ \begin{array}{ccc} \{0,1\}^{64} \times \{0,1\}^{64} & \rightarrow & \{0,1\}^{64} \\ (x_{(64)}, rk_{(64)}) & \mapsto & y_{(64)} \end{array} \right.$$

FOX64 Encryption The encryption $c_{(64)}$ by $\mathsf{FOX64}/k/r$ of a 64-bit plaintext $p_{(64)}$ is defined as

$$c_{(64)} = \mathsf{lmid64}(\mathsf{lmor64}(\dots(\mathsf{lmor64}(p_{(64)}, rk_{0(64)}), \dots, rk_{r-2(64)}), rk_{r-1(64)})$$

where

$$rk_{(r\cdot 64)} = rk_{0(64)}||rk_{1(64)}||\dots||rk_{r-1(64)}$$

is the subkey stream produced by the key schedule algorithm from the key $k_{(\ell)}$.

FOX64 Decryption The decryption $p_{(64)}$ by $\mathsf{FOX64}/k/r$ of a 64-bit ciphertext $c_{(64)}$ is defined as

$$p_{(64)} = \mathsf{lmid64}(\mathsf{lmio64}(\dots(\mathsf{lmio64}(c_{(64)}, rk_{r-1(64)}), \dots, rk_{1(64)}), rk_{0(64)})$$

where

$$rk_{(r\cdot 64)} = rk_{0(64)}||rk_{1(64)}||\dots||rk_{r-1(64)}$$

is the subkey stream produced by the key schedule algorithm from the key $k_{(\ell)}$, as for the encryption.

2.1.2 FOX128/k/r Skeleton

Similarly to the definition of FOX64, the 128-bit version of FOX is the $(r-1)$-times iteration of a round function denoted elmor128, followed by the application of a modified version of elmor128 named 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 $y_{(128)}$:

$$\mathsf{elmor128}, \mathsf{elmio128}, \mathsf{elmid128} : \begin{cases} \{0,1\}^{128} \times \{0,1\}^{128} & \to & \{0,1\}^{128} \\ (x_{(128)}, rk_{(128)}) & \mapsto & y_{(128)} \end{cases}$$

FOX128 Encryption The encryption $c_{(128)}$ by $\mathsf{FOX128}/k/r$ of a 128-bit plaintext $p_{(128)}$ is defined as

$$c_{(128)} = \mathsf{elmid128}(\mathsf{elmor128}(\dots\mathsf{elmor128}(p_{(128)}, rk_{0(128)}), \dots, rk_{r-2(128)}), rk_{r-1(128)})$$

where

$$rk_{(r\cdot 128)} = rk_{0(128)}||rk_{1(128)}||\dots||rk_{r-1(128)}$$

is the subkey stream produced by the key schedule algorithm from the key $k_{(\ell)}$.

FOX128 Decryption The decryption $p_{(128)}$ by $\mathsf{FOX128}/k/r$ of a 128-bit ciphertext $c_{(128)}$ is defined as

$$p_{(128)} = \mathsf{elmid128}(\mathsf{elmio128}(\dots\mathsf{elmio128}(C_{(128)}, rk_{r-1(128)}), \dots, rk_{1(128)}), rk_{0(128)})$$

where

$$rk_{(r\cdot 128)} = rk_{0(128)}||rk_{1(128)}||\dots||rk_{r-1(128)}$$

is the subkey stream produced by the key schedule algorithm from the key $k_{(\ell)}$, as for the encryption operation.

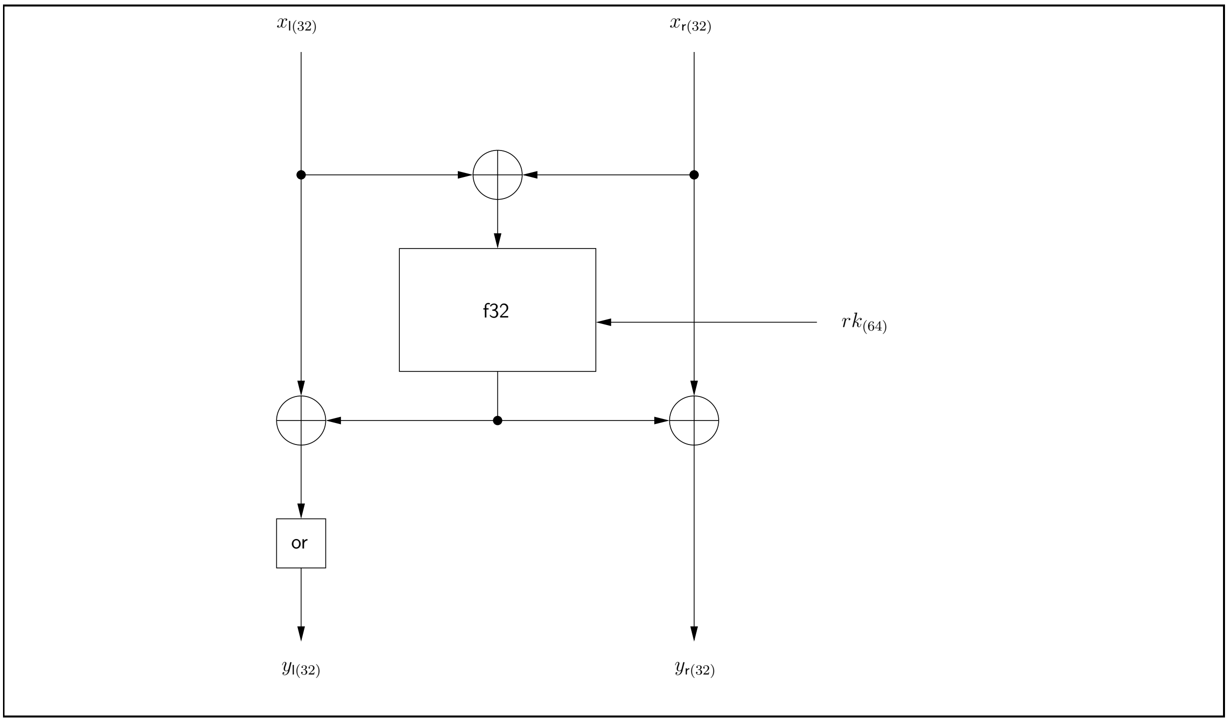

Figure 3: lmor64 Round Function

2.2 Internal Functions

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

2.2.1 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. 3, is built as a Lai-Massey scheme combined with an orthomorphism[2] or. This function transforms a 64-bit input $x_{(64)}$ split in two parts $x_{(64)} = x_{\mathsf{l}(32)}||x_{\mathsf{r}(32)}$ and a 64-bit round key $rk_{(64)}$ in a 64-bit output $y_{(64)} = y_{\mathsf{l}(32)}||y_{\mathsf{r}(32)}$ as follows:

$$
\begin{aligned}
y_{(64)} &= y_{\mathsf{l}(32)}||y_{\mathsf{r}(32)} = \mathsf{lmor64}\left(x_{\mathsf{r}(32)}||x_{\mathsf{r}(32)}\right) \\
&= \mathsf{or}\left(x_{\mathsf{l}(32)} \oplus \mathsf{f32}\left(x_{\mathsf{l}(32)} \oplus x_{\mathsf{r}(32)}, rk_{(64)}\right)\right) || \\
&\quad \left(x_{\mathsf{r}(32)} \oplus \mathsf{f32}\left(x_{\mathsf{l}(32)} \oplus x_{\mathsf{r}(32)}, rk_{(64)}\right)\right)
\end{aligned}
$$

The lmid64 function is a slightly modified version of lmor64, namely it is the same one without the orthomorphism or:

$$
\begin{aligned}
y_{(64)} &= y_{\mathsf{l}(32)}||y_{\mathsf{r}(32)} = \mathsf{lmid64}\left(x_{\mathsf{l}(32)}||x_{\mathsf{r}(32)}\right) \\
&= \left(x_{\mathsf{l}(32)} \oplus \mathsf{f32}\left(x_{\mathsf{l}(32)} \oplus x_{\mathsf{r}(32)}, rk_{(64)}\right)\right) || \\
&\quad \left(x_{\mathsf{r}(32)} \oplus \mathsf{f32}\left(x_{\mathsf{l}(32)} \oplus x_{\mathsf{r}(32)}, rk_{(64)}\right)\right)
\end{aligned}
$$

[2] An orthomorphism o on a group $(\mathcal{G}, +)$ is a permutation $x \mapsto \mathsf{o}(x)$ on $\mathcal{G}$ such that $x \mapsto \mathsf{o}(x) - x$ is also a permutation.

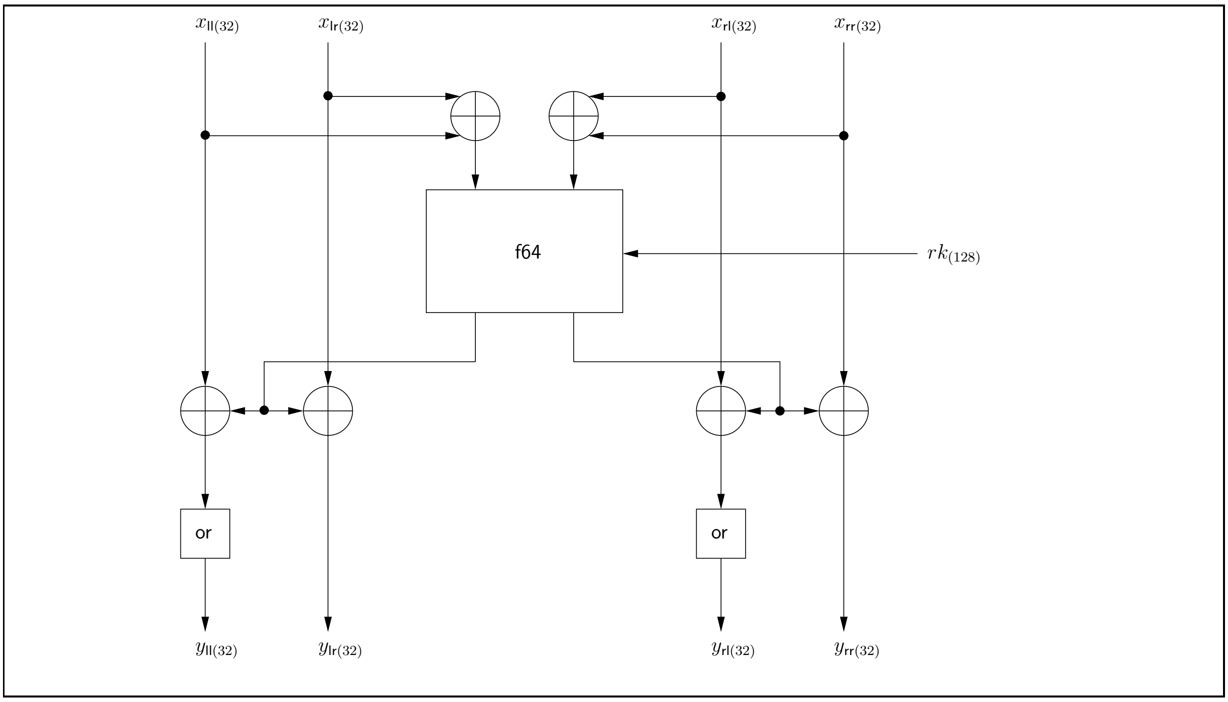

Figure 4: Round function elmor128

Finally, lmio64 is defined by

$$\begin{aligned} y_{(64)} &= y_{\mathsf{l}(32)} \| y_{\mathsf{r}(32)} = \mathsf{lmio64}\left(x_{\mathsf{l}(32)} \| x_{\mathsf{r}(32)}\right) \\ &= \mathsf{io}\left(x_{\mathsf{l}(32)} \oplus \mathsf{f32}\left(x_{\mathsf{l}(32)} \oplus x_{\mathsf{r}(32)}, rk_{(64)}\right)\right) \| \\ &\quad \left(x_{\mathsf{r}(32)} \oplus \mathsf{f32}\left(x_{\mathsf{l}(32)} \oplus x_{\mathsf{r}(32)}, rk_{(64)}\right)\right) \end{aligned}$$

where io is the inverse of the orthormorphism or.

2.2.2 Definitions of elmor128, elmid128, elmio128

In the 128-bit version of the algorithm, one uses three slightly different round functions, as in the 64-bit version. The first one, elmor128, illustrated in Fig. 4, is built as an *Extended Lai-Massey scheme* combined with two orthomorphisms or. This function transforms a 128-bit input $x_{(128)}$ split in four parts $x_{(128)} = x_{\mathsf{ll}(32)} \| x_{\mathsf{lr}(32)} \| x_{\mathsf{rl}(32)} \| x_{\mathsf{rr}(32)}$ and a 128-bit round key $rk_{(128)}$ in a 128-bit output $y_{(128)} = y_{\mathsf{ll}(32)} \| y_{\mathsf{lr}(32)} \| y_{\mathsf{rl}(32)} \| y_{\mathsf{rr}(32)}$ as follows:

$$\begin{aligned} y_{(128)} &= y_{\mathsf{ll}(32)} \| y_{\mathsf{lr}(32)} \| y_{\mathsf{rl}(32)} \| y_{\mathsf{rr}(32)} = \mathsf{elmor128}\left(x_{\mathsf{ll}(32)} \| x_{\mathsf{lr}(32)} \| x_{\mathsf{rl}(32)} \| x_{\mathsf{rr}(32)}\right) \\ &= \mathsf{or}\left(x_{\mathsf{ll}(32)} \oplus \mathsf{f64}\left((x_{\mathsf{ll}(32)} \oplus x_{\mathsf{lr}(32)}) \| (x_{\mathsf{rl}(32)} \oplus x_{\mathsf{rr}(32)}), rk_{(128)}\right)_{\mathsf{l}(32)}\right) \| \\ &\quad \left(x_{\mathsf{lr}(32)} \oplus \mathsf{f64}\left((x_{\mathsf{ll}(32)} \oplus x_{\mathsf{lr}(32)}) \| (x_{\mathsf{rl}(32)} \oplus x_{\mathsf{rr}(32)}), rk_{(128)}\right)_{\mathsf{l}(32)}\right) \| \\ &\quad \mathsf{or}\left(x_{\mathsf{rl}(32)} \oplus \mathsf{f64}\left((x_{\mathsf{ll}(32)} \oplus x_{\mathsf{lr}(32)}) \| (x_{\mathsf{rl}(32)} \oplus x_{\mathsf{rr}(32)}), rk_{(128)}\right)_{\mathsf{r}(32)}\right) \| \\ &\quad \left(x_{\mathsf{rr}(32)} \oplus \mathsf{f64}\left((x_{\mathsf{ll}(32)} \oplus x_{\mathsf{lr}(32)}) \| (x_{\mathsf{rl}(32)} \oplus x_{\mathsf{rr}(32)}), rk_{(128)}\right)_{\mathsf{r}(32)}\right) \end{aligned}$$

The $\mathsf{elmid128}$ function is a slightly modified version of $\mathsf{elmor128}$, namely it is the same one without the orthomorphism or:

$$\begin{aligned} y_{(128)} &= y_{\mathsf{ll}(32)}||y_{\mathsf{lr}(32)}||y_{\mathsf{rl}(32)}||y_{\mathsf{rr}(32)} = \mathsf{elmid128}\left(x_{\mathsf{ll}(32)}||x_{\mathsf{lr}(32)}||x_{\mathsf{rl}(32)}||x_{\mathsf{rr}(32)}\right) \\ &= \left(x_{\mathsf{ll}(32)} \oplus \mathsf{f64}\left((x_{\mathsf{ll}(32)} \oplus x_{\mathsf{lr}(32)})||(x_{\mathsf{rl}(32)} \oplus x_{\mathsf{rr}(32)}), rk_{(128)}\right)_{\mathsf{l}(32)}\right) || \\ &\quad \left(x_{\mathsf{lr}(32)} \oplus \mathsf{f64}\left((x_{\mathsf{ll}(32)} \oplus x_{\mathsf{lr}(32)})||(x_{\mathsf{rl}(32)} \oplus x_{\mathsf{rr}(32)}), rk_{(128)}\right)_{\mathsf{l}(32)}\right) || \\ &\quad \left(x_{\mathsf{rl}(32)} \oplus \mathsf{f64}\left((x_{\mathsf{ll}(32)} \oplus x_{\mathsf{lr}(32)})||(x_{\mathsf{rl}(32)} \oplus x_{\mathsf{rr}(32)}), rk_{(128)}\right)_{\mathsf{r}(32)}\right) || \\ &\quad \left(x_{\mathsf{rr}(32)} \oplus \mathsf{f64}\left((x_{\mathsf{ll}(32)} \oplus x_{\mathsf{lr}(32)})||(x_{\mathsf{rl}(32)} \oplus x_{\mathsf{rr}(32)}), rk_{(128)}\right)_{\mathsf{r}(32)}\right) \end{aligned}$$

Finally, $\mathsf{elmio128}$ is defined by

$$\begin{aligned} y_{(128)} &= y_{\mathsf{ll}(32)}||y_{\mathsf{lr}(32)}||y_{\mathsf{rl}(32)}||y_{\mathsf{rr}(32)} = \mathsf{elmio128}\left(x_{\mathsf{ll}(32)}||x_{\mathsf{lr}(32)}||x_{\mathsf{rl}(32)}||x_{\mathsf{rr}(32)}\right) \\ &= \mathsf{io}\left(x_{\mathsf{ll}(32)} \oplus \mathsf{f64}\left((x_{\mathsf{ll}(32)} \oplus x_{\mathsf{lr}(32)})||(x_{\mathsf{rl}(32)} \oplus x_{\mathsf{rr}(32)}), rk_{(128)}\right)_{\mathsf{l}(32)}\right) || \\ &\quad \left(x_{\mathsf{lr}(32)} \oplus \mathsf{f64}\left((x_{\mathsf{ll}(32)} \oplus x_{\mathsf{lr}(32)})||(x_{\mathsf{rl}(32)} \oplus x_{\mathsf{rr}(32)}), rk_{(128)}\right)_{\mathsf{l}(32)}\right) || \\ &\quad \mathsf{io}\left(x_{\mathsf{rl}(32)} \oplus \mathsf{f64}\left((x_{\mathsf{ll}(32)} \oplus x_{\mathsf{lr}(32)})||(x_{\mathsf{rl}(32)} \oplus x_{\mathsf{rr}(32)}), rk_{(128)}\right)_{\mathsf{r}(32)}\right) || \\ &\quad \left(x_{\mathsf{rr}(32)} \oplus \mathsf{f64}\left((x_{\mathsf{ll}(32)} \oplus x_{\mathsf{lr}(32)})||(x_{\mathsf{rl}(32)} \oplus x_{\mathsf{rr}(32)}), rk_{(128)}\right)_{\mathsf{r}(32)}\right) \end{aligned}$$

2.2.3 Definitions of or and io

The orthomorphism or is a function taking a 32-bit input $x_{(32)} = x_{\mathsf{l}(16)}||x_{\mathsf{r}(16)}$ and returning a 32-bit output $y_{(32)} = y_{\mathsf{l}(16)}||y_{\mathsf{r}(16)}$. It is defined as

$$y_{\mathsf{l}(16)}||y_{\mathsf{r}(16)} = \mathsf{or}\left(x_{\mathsf{l}(16)}||x_{\mathsf{r}(16)}\right) = x_{\mathsf{r}(16)}||\left(x_{\mathsf{l}(16)} \oplus x_{\mathsf{r}(16)}\right)$$

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

$$y_{\mathsf{l}(16)}||y_{\mathsf{r}(16)} = \mathsf{io}\left(x_{\mathsf{l}(32)}||x_{\mathsf{r}(32)}\right) = \left(x_{\mathsf{l}(16)} \oplus x_{\mathsf{r}(16)}\right)||x_{\mathsf{l}(16)}$$

2.2.4 Definition of f32

The function $\mathsf{f32}$ builds the core of $\mathsf{FOX64}/k/r$. It is built of three main parts: a substitution part, denoted $\mathsf{sigma4}$, a diffusion part, denoted $\mathsf{mu4}$, and a round key addition part (see Fig. 5). Formally, the $\mathsf{f32}$ function takes a 32-bit input $x_{(32)}$, a 64-bit round key $rk_{(64)} = rk_{0(32)}||rk_{1(32)}$ and returns a 32-bit output $y_{(32)}$. The $\mathsf{f32}$ function is then formally defined as

$$\begin{aligned} y_{(32)} &= \mathsf{f32}\left(x_{(32)}, rk_{(64)}\right) \\ &= \mathsf{sigma4}(\mathsf{mu4}(\mathsf{sigma4}(x_{(32)} \oplus rk_{0(32)})) \oplus rk_{1(32)}) \oplus rk_{0(32)} \end{aligned}$$

2.2.5 Definition of f64

The function $\mathsf{f64}$ builds the core of $\mathsf{FOX128}/k/r$. It is built of three main parts: a substitution part, denoted $\mathsf{sigma8}$, a diffusion part, denoted $\mathsf{mu8}$, and a round key addition part (see Fig. 6). Formally, the $\mathsf{f64}$ function takes a 64-bit input $x_{(64)}$, a 128-bit round key $rk_{(128)} = rk_{0(64)}||rk_{1(64)}$ and returns a 64-bit output $y_{(64)}$. The $\mathsf{f64}$ function is then defined as

$$\begin{aligned} y_{(64)} &= \mathsf{f64}\left(x_{(64)}, rk_{(128)}\right) \\ &= \mathsf{sigma8}(\mathsf{mu8}(\mathsf{sigma8}(x_{(64)} \oplus rk_{0(64)})) \oplus rk_{1(64)}) \oplus rk_{0(64)} \end{aligned}$$

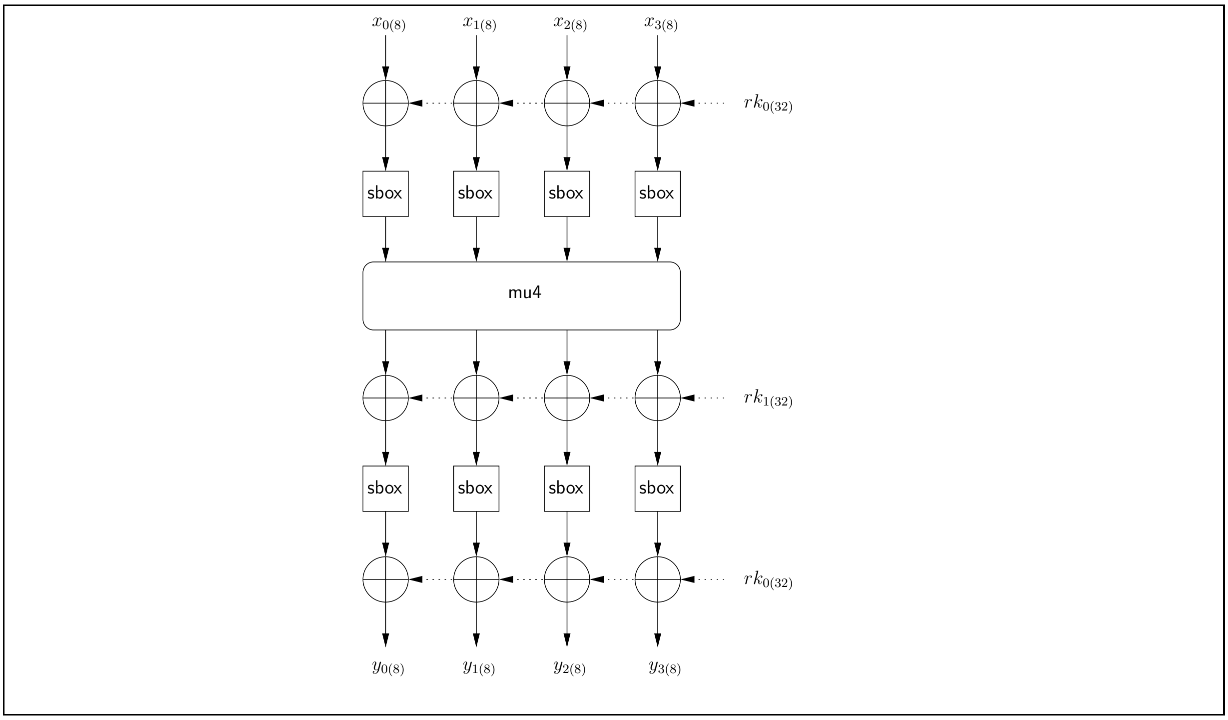

Figure 5: Function f32

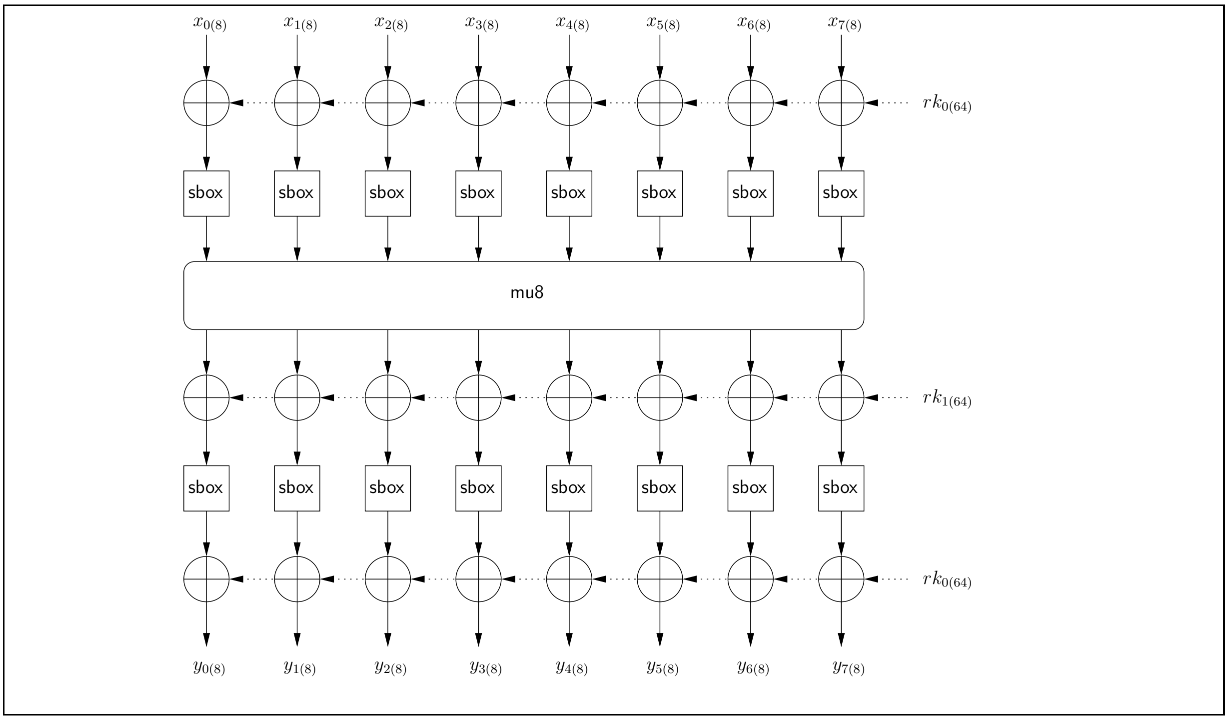

Figure 6: Function f64

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

Figure 7: Mapping sbox

2.2.6 Definition of sigma4, sigma8 and sbox

The function sigma4 takes a 32-bit input $x_{(32)} = x_{0(8)}||x_{1(8)}||x_{2(8)}||x_{3(8)}$ and returns a 32-bit output $y_{(32)}$. It is defined as

$$\begin{aligned} y_{(32)} &= \mathsf{sigma4}\left(x_{0(8)}||x_{1(8)}||x_{2(8)}||x_{3(8)}\right) \\ &= \mathsf{sbox}(x_{0(8)})||\mathsf{sbox}(x_{1(8)})||\mathsf{sbox}(x_{2(8)})||\mathsf{sbox}(x_{3(8)}) \end{aligned}$$

The function sigma8 takes a 64-bit input

$$x_{(64)} = x_{0(8)}||x_{1(8)}||x_{2(8)}||x_{3(8)}||x_{4(8)}||x_{5(8)}||x_{6(8)}||x_{7(8)}$$

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

$$\begin{aligned} y_{(64)} &= \mathsf{sigma8}\left(x_{0(8)}||x_{1(8)}||x_{2(8)}||x_{3(8)}||x_{4(8)}||x_{5(8)}||x_{6(8)}||x_{7(8)}\right) \\ &= \mathsf{sbox}(x_{0(8)})||\mathsf{sbox}(x_{1(8)})||\mathsf{sbox}(x_{2(8)})||\mathsf{sbox}(x_{3(8)})|| \\ &\quad\ \mathsf{sbox}(x_{4(8)})||\mathsf{sbox}(x_{5(8)})||\mathsf{sbox}(x_{6(8)})||\mathsf{sbox}(x_{7(8)}) \end{aligned}$$

Finally, the sbox function is the lookup-up table defined in Fig. 7. We read this table as follows: to compute sbox(4C), one selects first the row named 4. (*i.e.* the fifth row), and then one selects the column named .C (*i.e.* the thirteenth column) and we get finally

$$\mathsf{sbox}(\mathtt{4C}) = \mathtt{15}$$

2.2.7 Definition of mu4

The diffusive part of f32 is a linear $(4, 4)$-multipermutation defined on $\mathrm{GF}(2^8)$. Formally, it is a function taking a 32-bit input

$$x_{(32)} = x_{0(8)}||x_{1(8)}||x_{2(8)}||x_{3(8)}$$

and returning a 32-bit output

$$y_{(32)} = y_{0(8)}||y_{1(8)}||y_{2(8)}||y_{3(8)}$$

and defined by

$$\begin{pmatrix} y_{0(8)} \\ y_{1(8)} \\ y_{2(8)} \\ y_{3(8)} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \alpha \\ 1 & c & \alpha & 1 \\ c & \alpha & 1 & 1 \\ \alpha & 1 & c & 1 \end{pmatrix} \times \begin{pmatrix} x_{0(8)} \\ x_{1(8)} \\ x_{2(8)} \\ x_{3(8)} \end{pmatrix}$$

where

$$c = \alpha^{-1} + 1 = \alpha^7 + \alpha^6 + \alpha^5 + \alpha^4 + \alpha^3 + \alpha^2 + 1$$

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

2.2.8 Definition of mu8

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

$$x_{(64)} = x_{0(8)}||x_{1(8)}||x_{2(8)}||x_{3(8)}||x_{4(8)}||x_{5(8)}||x_{6(8)}||x_{7(8)}$$

and returning a 64-bit output

$$y_{(64)} = y_{0(8)}||y_{1(8)}||y_{2(8)}||y_{3(8)}||y_{4(8)}||y_{5(8)}||y_{6(8)}||y_{7(8)}$$

f64 is defined as

$$\begin{pmatrix} y_{0(8)} \\ y_{1(8)} \\ y_{2(8)} \\ y_{3(8)} \\ y_{4(8)} \\ y_{5(8)} \\ y_{6(8)} \\ y_{7(8)} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & a \\ 1 & a & b & c & d & e & f & 1 \\ a & b & c & d & e & f & 1 & 1 \\ b & c & d & e & f & 1 & a & 1 \\ c & d & e & f & 1 & a & b & 1 \\ d & e & f & 1 & a & b & c & 1 \\ e & f & 1 & a & b & c & d & 1 \\ f & 1 & a & b & c & d & e & 1 \end{pmatrix} \times \begin{pmatrix} x_{0(8)} \\ x_{1(8)} \\ x_{2(8)} \\ x_{3(8)} \\ x_{4(8)} \\ x_{5(8)} \\ x_{6(8)} \\ x_{7(8)} \end{pmatrix}$$

where

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

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

2.3 Key-Schedule Algorithms

The key schedule is the algorithm which derives the subkey material

$$rk_{(r\cdot 64)} = rk_{0(64)}||rk_{1(64)}||\ldots||rk_{r-1(64)}$$

and

$$rk_{(r\cdot 128)} = rk_{0(128)}||rk_{1(128)}||\ldots||rk_{r-1(128)}$$

(for FOX64 and FOX128, respectively) from the key $k_{(\ell)}$.

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

Figure 8: Key-Schedule Algorithms Characteristics

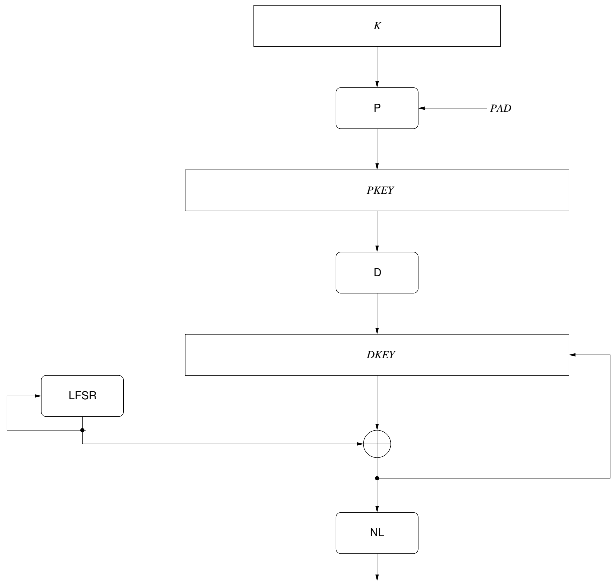

Figure 9: Key-Schedule Algorithm (High-Level Overview)

2.3.1 General Overview

A FOX key $k_{(\ell)}$ must have a bit-length ℓ such that $0 \leq \ell \leq 256$, and ℓ 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 table in Fig. 8 defines precisely the relation between the key size, the block size, the constant ek and the key-schedule algorithm version.

The three different versions of the key-schedule algorithm are constituted of four main parts: a padding part, denoted P, expanding $k_{(\ell)}$ 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. 9 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 ℓ 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.

2.3.2 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 ℓ bits, with $0 \leq \ell \leq 128$ and a number of rounds r. It returns in output r 64-bit subkeys. KS64 is formally defined in Alg. 2.

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

```
/* Preprocessing */
```
$pkey \leftarrow \mathsf{P}(k)$
$mkey \leftarrow \mathsf{M}(pkey)$
```
/* Initialization of the loop */
```
$i \leftarrow 1$
```
/* Loop */
```
while $i \leq r$ **do**
  $dkey \leftarrow \mathsf{D}(mkey, i, r)$
  Output $rk_{i-1(x)} \leftarrow \mathsf{NLx}(dkey)$
  $i \leftarrow i + 1$
end while

Algorithm 2 Key-Schedule Algorithm KS64

```
/* Preprocessing */
```
if $\ell < ek$ **then**
  $pkey = \mathsf{P}(k)$
  $mkey = \mathsf{M}(pkey)$
else
  $pkey = k$
  $mkey = pkey$
end if
```
/* Initialization of the loop */
```
$i = 1$
```
/* Loop */
```
while $i \leq r$ **do**
  $dkey = \mathsf{D}(mkey, i, r)$
  Output $rk_{i-1(64)} = \mathsf{NL64}(dkey)$
  $i = i + 1$
end while

2.3.3 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 ℓ bits, with $136 \leq \ell \leq 256$ and a number of rounds r. It returns in output r 64-bit subkeys. KS64h is formally defined in Alg. 3.

Algorithm 3 Key-Schedule Algorithm KS64h

```
/* Preprocessing */
if ℓ < ek then
   pkey = P(k)
   mkey = M(pkey)
else
   pkey = k
   mkey = pkey
end if
/* Initialization of the loop */
i = 1
/* Loop */
while i ≤ r do
   dkey = D(mkey, i, r)
   Output rk_{i−1(64)} = NL64h(dkey)
   i = i + 1
end while
```

2.3.4 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 ℓ bits, with $0 \leq \ell \leq 256$ and a number of rounds r. It returns in output r 128-bit subkeys. KS128 is formally defined in Alg. 4.

Algorithm 4 Key-Schedule Algorithm KS128

```
/* Preprocessing */
if ℓ < ek then
   pkey = P(k)
   mkey = M(pkey)
else
   pkey = k
   mkey = pkey
end if
/* Initialization of the loop */
i = 1
/* Loop */
while i ≤ r do
   dkey = D(mkey, i, r)
   Output rk_{i−1(128)} = NL128(dkey)
   i = i + 1
end while
```

2.3.5 Definition of P

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

---

Algorithm 5 P-Part

Output $pkey = k || \mathsf{pad}_{[0 \ldots ek-\ell-1]}$

---

constant value is defined in the following section.

2.3.6 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, it is the concatenation of the four following 64-bit constants

$$\begin{array}{rcll} \mathsf{pad} & = & \texttt{0xB7E151628AED2A6A} & || \\ & & \texttt{0xBF7158809CF4F3C7} & || \\ & & \texttt{0x62E7160F38B4DA56} & || \\ & & \texttt{0xA784D9045190CFEF} & \end{array}$$

2.3.7 Definition of M

The M-part is used to mix the padded key $pkey$, such that the constant words are mixed uo by using the randomness provided by the key. 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

$$mkey_{i(8)} = pkey_{i(8)} \oplus \left(mkey_{i-1(8)} + mkey_{i-2(8)} \bmod 2^8 \right) \quad 0 \leq i \leq \frac{ek}{8} - 1$$

with the convention that

$$\mathrm{mkey}_{-2(8)} = \texttt{0x6A} \text{ and } \mathrm{mkey}_{-1(8)} = \texttt{0x76}$$

Note here that $+$ denotes the addition performed modulo 2^8 while $\oplus$ denotes the addition in $\mathrm{GF}\left(2^8\right)$, which is actually a XOR operation.

2.3.8 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 $\left\lfloor \frac{ek}{24} \right\rfloor$ 24-bit values $mkey_{j(24)}$, with $0 \leq j \leq \left\lfloor \frac{ek}{24} \right\rfloor - 1$ concatenated with one residue byte $mkeyrb_{(8)}$ (if $ek = 128$) or two residue bytes $mkeyrb_{(16)}$ (if $ek = 256$), and is modified according to

$$dkey_{j(24)} = mkey_{j(24)} \oplus \mathsf{LFSR}\left((i-1) \cdot \left\lceil \frac{ek}{24} \right\rceil + j, r \right)$$

for $0 \leq j \leq \lfloor \frac{ek}{24} \rfloor - 1$; the $dkeyrb_{(8)}$ value ($dkeyrb_{(16)}$) is obtained by XORing the most 8 (16) significant bits of $\mathsf{LFSR}((i-1) \cdot \lceil \frac{ek}{24} \rceil + \lfloor \frac{ek}{24} \rfloor, r)$ with $mkeyrb_{(8)}$ ($mkeyrb_{(16)}$), respectively. The remaining 16 (8) bits of the LFSR routine output are discarded.

2.3.9 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).

Definition 2.1 (Irreducible Polynomial $\mathsf{PKS}(\xi)$). *The polynomial representing* GF (2^{24}) *in the FOX block cipher family is the irreducible polynomial over* GF (2) *defined by*

$$\mathsf{PKS}(\xi) = \xi^{24} + \xi^4 + \xi^3 + \xi + 1$$

The register is initially seeded with the value $\texttt{0x6A}||r_{(8)}||\overline{r_{(8)}}$, where $r_{(8)}$ is expressed as an 8-bit value, and $\overline{r_{(8)}}$ is its bitwise complemented version (i.e. $r_{(8)} = \overline{r_{(8)}} \oplus \texttt{0xFF}$). LSFR is described formally in Alg. 6.

Algorithm 6 LFSR Algorithm

```
/* Initialization */
reg = 0x6A||r||r̄
/* Pre-Clocking */
p = 0
while p < c do
  p = p + 1
  if (reg AND 0x800000) ≠ 0x000000 then
    reg = (reg ≪ 1) ⊕ 0x00001B
  else
    reg = (reg ≪ 1)
  end if
end while
Output reg
```

2.3.10 Definition of NL64

The NL64-part takes a single input: the 128-bit value *dkey* corresponding to the current round. The *dkey* value passes through a substitution layer (made of four parallel sigma4 functions), a diffusion layer (made of four parallel mu4 functions) and a mixing layer called mix64. Then, the constant $\mathsf{pad}_{[0\ldots127]}$ 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 lmor64 round function, where the subkey is the left half of the *dkey* value and second by a lmid64 function, where the subkey is the right half of *dkey*. The resulting value is defined to be the 64-bit round key. Fig. 10 illustrates the NL64 process and Alg. 7 describes it formally.

2.3.11 Definition of NL64h

The NL64h-part takes a single input: the 256-bit value *dkey* corresponding to the current round. The *dkey* value passes through a substitution layer (made of eight parallel sigma4 functions), a

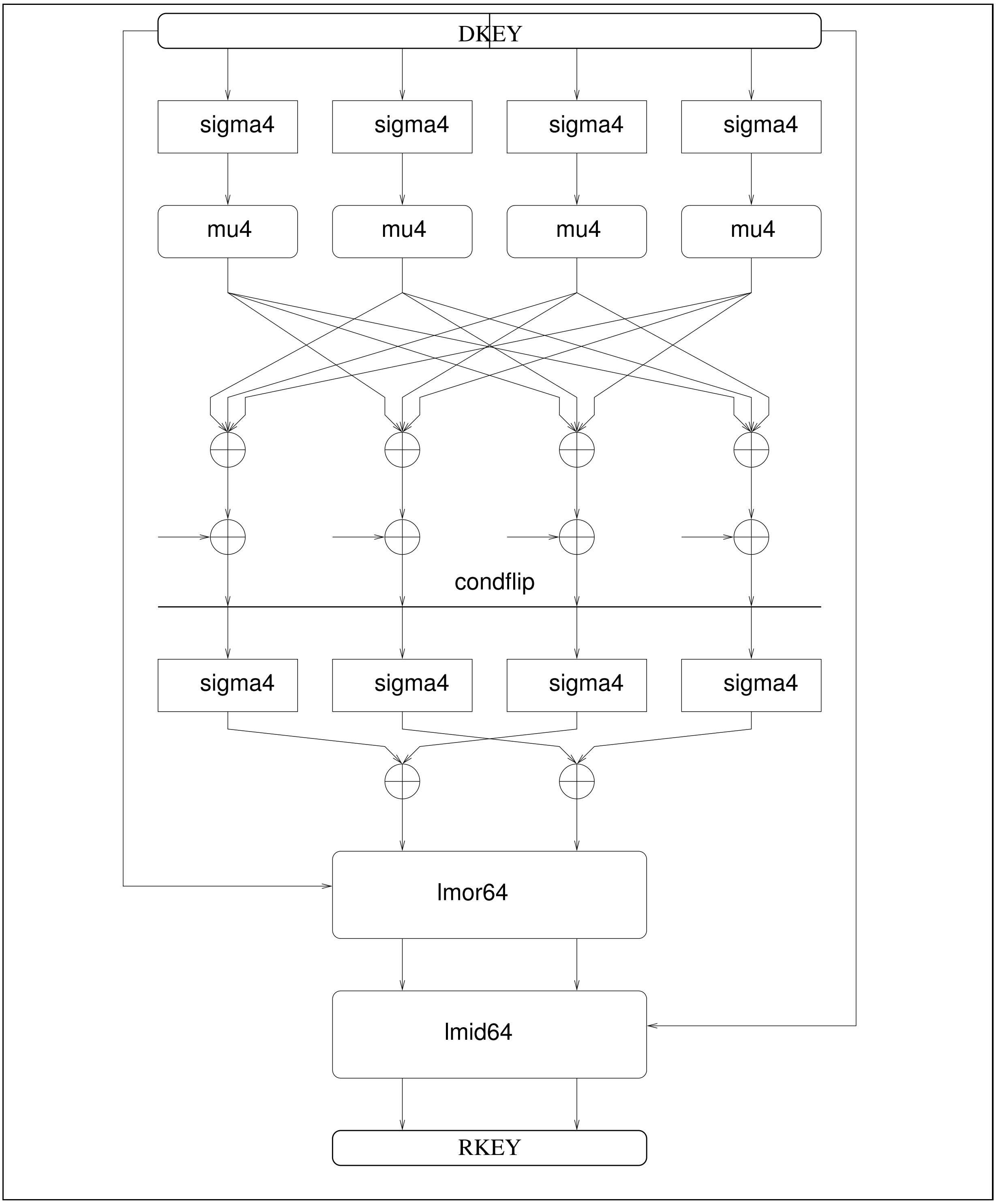

Figure 10: NL64 Part

Algorithm 7 NL64 Part

$t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}$=$dkey$
$t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}$=$\mathsf{sigma4}(t_{0(32)})||\mathsf{sigma4}(t_{1(32)})||\mathsf{sigma4}(t_{2(32)})||\mathsf{sigma4}(t_{3(32)})$
$t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}$=$\mathsf{mu4}(t_{0(32)})||\mathsf{mu4}(t_{1(32)})||\mathsf{mu4}(t_{2(32)})||\mathsf{mu4}(t_{3(32)})$
$t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}$=$\mathsf{mix64}(t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)})$
$t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}$=$(t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)})\oplus\mathsf{pad}_{[0..127]}$
if $k = ek$ **then**
  $t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}$=$\overline{t_{0(32)}}||\overline{t_{1(32)}}||\overline{t_{2(32)}}||\overline{t_{3(32)}}$
end if
$t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}$=$\mathsf{sigma4}(t_{0(32)})||\mathsf{sigma4}(t_{1(32)})||\mathsf{sigma4}(t_{2(32)})||\mathsf{sigma4}(t_{3(32)})$
$t_{0(32)}||t_{1(32)}$=$(t_{0(32)}\oplus t_{2(32)})||(t_{1(32)}\oplus t_{3(32)})$
$t_{0(32)}||t_{1(32)}$=$\mathsf{lmor64}(t_{0(32)}||t_{1(32)},dkey_{[0...63]})$
$t_{0(32)}||t_{1(32)}$=$\mathsf{lmid64}(t_{0(32)}||t_{1(32)},dkey_{[64...127]})$
Output $t_{0(32)}||t_{1(32)}$ as round subkey.

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. 11 illustrates the NL64h process and Alg. 8 describes it formally.

2.3.12 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 $dkey$. The resulting value is defined to be the 128-bit round key. Fig. 12 illustrates the NL128 process and Alg. 9 describes it formally.

2.3.13 Definition of mix64

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

$$x = x_{0(32)}||x_{1(32)}||x_{2(32)}||x_{3(32)}$$

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

$$\begin{aligned}
y_{0(32)} &= x_{1(32)} \oplus x_{2(32)} \oplus x_{3(32)} \\
y_{1(32)} &= x_{0(32)} \oplus x_{2(32)} \oplus x_{3(32)} \\
y_{2(32)} &= x_{0(32)} \oplus x_{1(32)} \oplus x_{3(32)} \\
y_{3(32)} &= x_{0(32)} \oplus x_{1(32)} \oplus x_{2(32)}
\end{aligned}$$

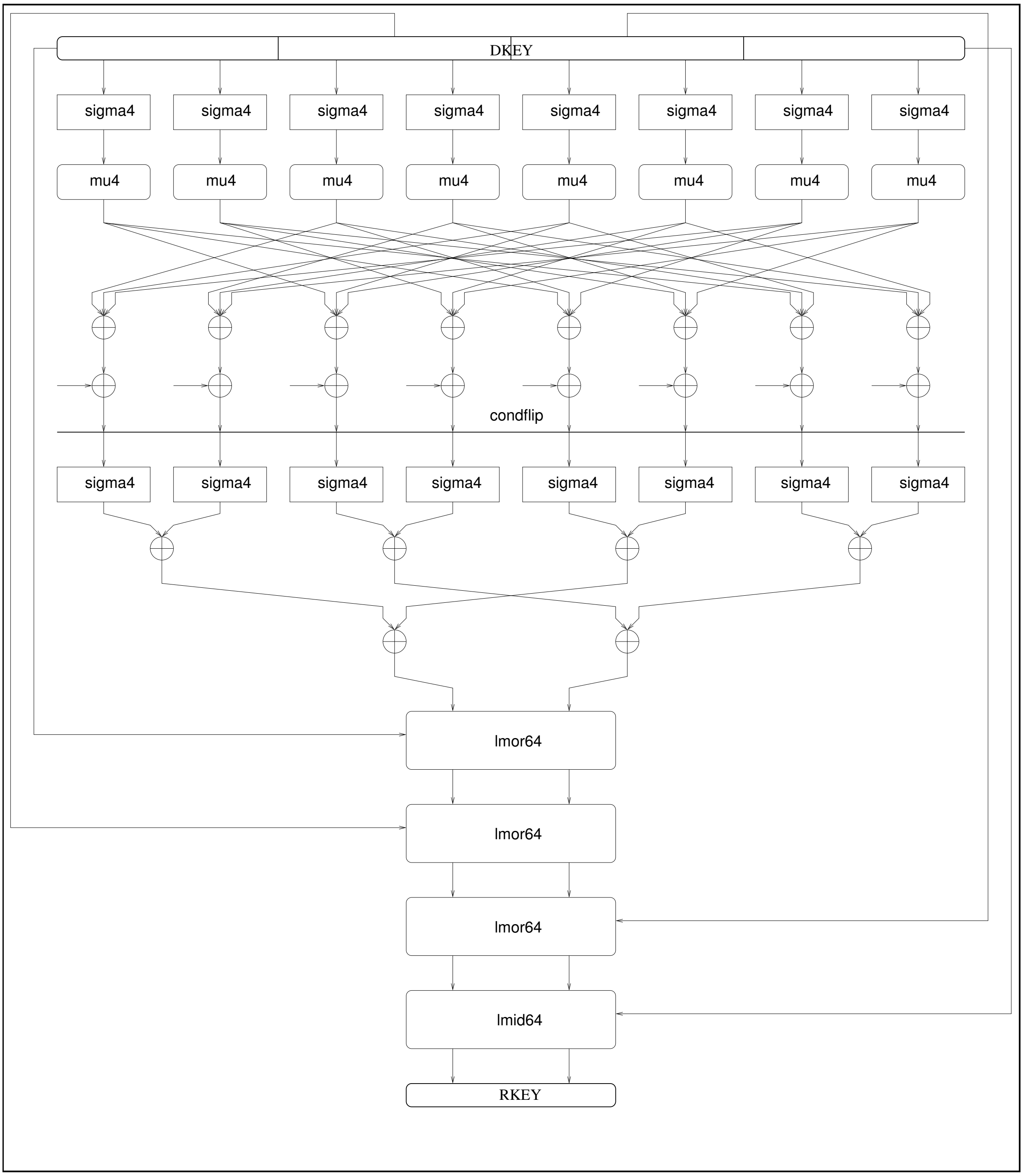

Figure 11: NL64h Part

Algorithm 8 NL64h Part

/* Initialization */

$t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}||t_{4(32)}||t_{5(32)}||t_{6(32)}||t_{7(32)}$=$dkey$

/* Substitution Layer */

$t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}$=sigma4($t_{0(32)}$)||sigma4($t_{1(32)}$)||sigma4($t_{2(32)}$)||sigma4($t_{3(32)}$)

$t_{4(32)}||t_{5(32)}||t_{6(32)}||t_{7(32)}$=sigma4($t_{4(32)}$)||sigma4($t_{5(32)}$)||sigma4($t_{6(32)}$)||sigma4($t_{7(32)}$)

/* Diffusion Layer */

$t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}$=mu4($t_{0(32)}$)||mu4($t_{1(32)}$)||mu4($t_{2(32)}$)||mu4($t_{3(32)}$)

$t_{4(32)}||t_{5(32)}||t_{6(32)}||t_{7(32)}$=mu4($t_{4(32)}$)||mu4($t_{5(32)}$)||mu4($t_{6(32)}$)||mu4($t_{7(32)}$)

$t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}||t_{4(32)}||t_{5(32)}||t_{6(32)}||t_{7(32)}$=
 mix64h($t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}||t_{4(32)}||t_{5(32)}||t_{6(32)}||t_{7(32)}$)

$t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}||t_{4(32)}||t_{5(32)}||t_{6(32)}||t_{7(32)}$=
 ($t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}||t_{4(32)}||t_{5(32)}||t_{6(32)}||t_{7(32)}$)$\oplus$pad

if $k = ek$ **then**

 $t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}||t_{4(32)}||t_{5(32)}||t_{6(32)}||t_{7(32)}=\overline{t_{0(32)}}||\overline{t_{1(32)}}||\overline{t_{2(32)}}||\overline{t_{3(32)}}||\overline{t_{4(32)}}||\overline{t_{5(32)}}||\overline{t_{6(32)}}||\overline{t_{7(32)}}$

end if

/* Substitution Layer */

$t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}$=sigma4($t_{0(32)}$)||sigma4($t_{1(32)}$)||sigma4($t_{2(32)}$)||sigma4($t_{3(32)}$)

$t_{4(32)}||t_{5(32)}||t_{6(32)}||t_{7(32)}$=sigma4($t_{4(32)}$)||sigma4($t_{5(32)}$)||sigma4($t_{6(32)}$)||sigma4($t_{7(32)}$)

/* Hashing Layer */

$t_{0(32)}||t_{1(32)}||t_{2(32)}||t_{3(32)}=(t_{0(32)}\oplus t_{1(32)})||(t_{2(32)}\oplus t_{3(32)})||(t_{4(32)}\oplus t_{5(32)})||(t_{6(32)}\oplus t_{7(32)})$

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

/* Encryption Layer */

$t_{0(32)}||t_{1(32)}$=lmor64($t_{0(32)}||t_{1(32)}$,$dkey_{[0...63]}$)

$t_{0(32)}||t_{1(32)}$=lmor64($t_{0(32)}||t_{1(32)}$,$dkey_{[64...127]}$)

$t_{0(32)}||t_{1(32)}$=lmor64($t_{0(32)}||t_{1(32)}$,$dkey_{[128...191]}$)

$t_{0(32)}||t_{1(32)}$=lmid64($t_{0(32)}||t_{1(32)}$,$dkey_{[192...256]}$)

Output $t_{0(32)}||t_{1(32)}$ as round subkey.

Algorithm 9 NL128 Part

$t_{0(64)}||t_{1(64)}||t_{2(64)}||t_{3(64)}$=$dkey$

$t_{0(64)}||t_{1(64)}||t_{2(64)}||t_{3(64)}$=sigma8($t_{0(64)}$)||sigma8($t_{1(64)}$)||sigma8($t_{2(64)}$)||sigma8($t_{3(64)}$)

$t_{0(64)}||t_{1(64)}||t_{2(64)}||t_{3(64)}$=mu8($t_{0(64)}$)||mu8($t_{1(64)}$)||mu8($t_{2(64)}$)||mu8($t_{3(64)}$)

$t_{0(64)}||t_{1(64)}||t_{2(64)}||t_{3(64)}$=mix128($t_{0(64)}||t_{1(64)}||t_{2(64)}||t_{3(64)}$)

$t_{0(64)}||t_{1(64)}||t_{2(64)}||t_{3(64)}$=($t_{0(64)}||t_{1(64)}||t_{2(64)}||t_{3(64)}$)$\oplus$pad

if $k = ek$ **then**

 $t_{0(64)}||t_{1(64)}||t_{2(64)}||t_{3(64)}=\overline{t_{0(64)}}||\overline{t_{1(64)}}||\overline{t_{2(64)}}||\overline{t_{3(64)}}$

end if

$t_{0(64)}||t_{1(64)}||t_{2(64)}||t_{3(64)}$=sigma8($t_{0(64)}$)||sigma8($t_{1(64)}$)||sigma8($t_{2(64)}$)||sigma8($t_{3(64)}$)

$t_{0(64)}||t_{1(64)}=(t_{0(64)}\oplus t_{2(64)})||(t_{1(64)}\oplus t_{3(64)})$

$t_{0(64)}||t_{1(64)}$=elmor128($t_{0(64)}||t_{1(64)}$,$dkey_{[0...127]}$)

$t_{0(64)}||t_{1(64)}$=elmid128($t_{0(64)}||t_{1(64)}$,$dkey_{[128...255]}$)

Output $t_{0(64)}||t_{1(64)}$ as round subkey.

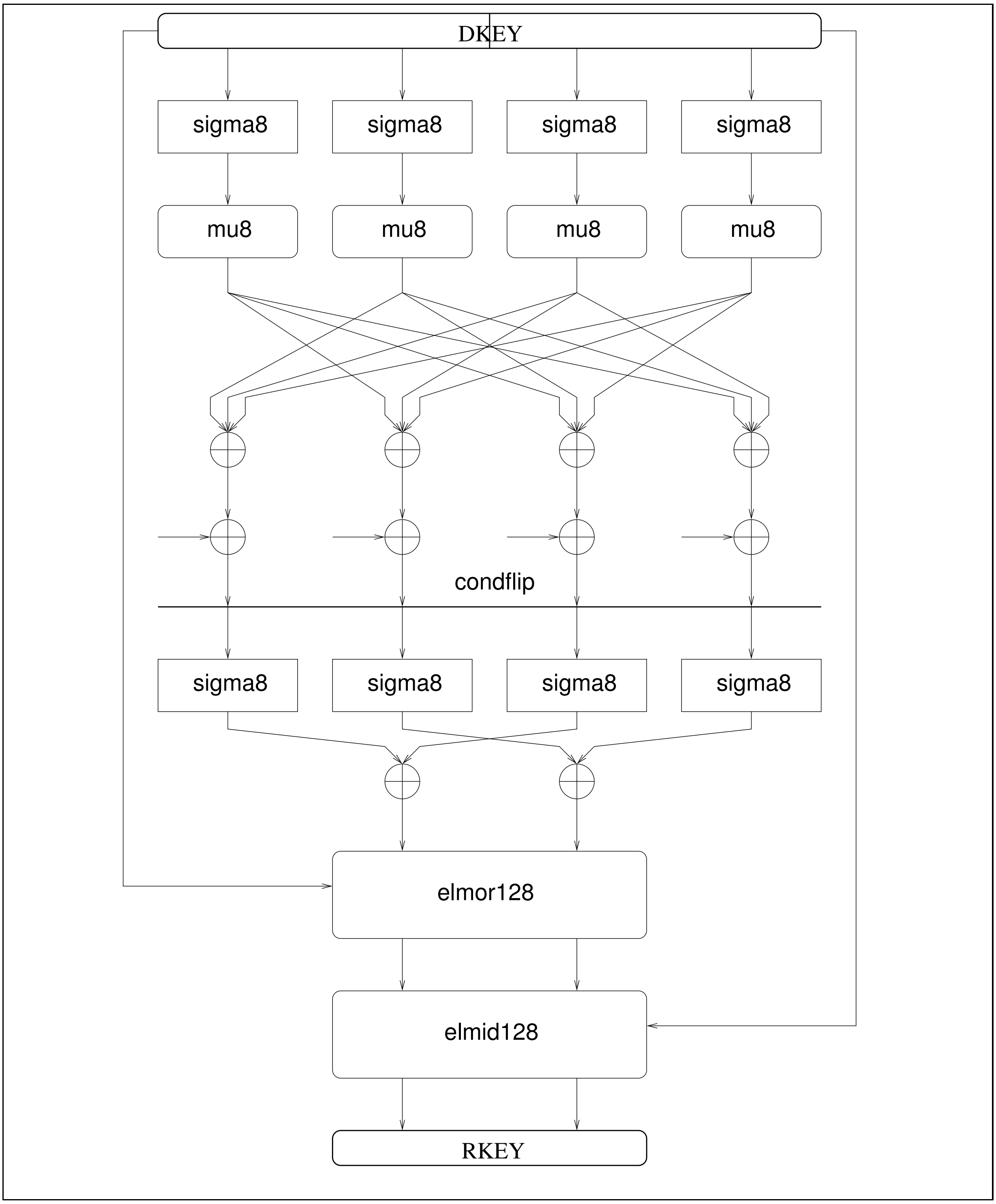

Figure 12: NL128 Part

2.3.14 Definition of mix64h

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

$$x = x_{0(32)}||x_{1(32)}||x_{2(32)}||x_{3(32)}||x_{4(32)}||x_{5(32)}||x_{6(32)}||x_{7(32)}$$

the mix64h function consists in processing it by the following relations, resulting in an output vector denoted

$$y = y_{0(32)}||y_{1(32)}||y_{2(32)}||y_{3(32)}||y_{4(32)}||y_{5(32)}||y_{6(32)}||y_{7(32)}$$

More formally, mix64h is defined as

$$\begin{aligned}
y_{0(32)} &= x_{2(32)} \oplus x_{4(32)} \oplus x_{6(32)} \\
y_{1(32)} &= x_{3(32)} \oplus x_{5(32)} \oplus x_{7(32)} \\
y_{2(32)} &= x_{0(32)} \oplus x_{4(32)} \oplus x_{6(32)} \\
y_{3(32)} &= x_{1(32)} \oplus x_{5(32)} \oplus x_{7(32)} \\
y_{4(32)} &= x_{0(32)} \oplus x_{2(32)} \oplus x_{6(32)} \\
y_{5(32)} &= x_{1(32)} \oplus x_{3(32)} \oplus x_{7(32)} \\
y_{6(32)} &= x_{0(32)} \oplus x_{2(32)} \oplus x_{4(32)} \\
y_{7(32)} &= x_{1(32)} \oplus x_{3(32)} \oplus x_{5(32)}
\end{aligned}$$

2.3.15 Definition of mix128

Given an input vector of four 64-bit values, denoted $x = x_{0(64)}||x_{1(64)}||x_{2(64)}||x_{3(64)}$, the mix64 function consists in processing it by the following relations, resulting in an output vector denoted $y = y_{0(64)}||y_{1(64)}||y_{2(64)}||y_{3(64)}$. More formally, mix128 is defined as

$$\begin{aligned}
y_{0(64)} &= x_{1(64)} \oplus x_{2(64)} \oplus x_{3(64)} \\
y_{1(64)} &= x_{0(64)} \oplus x_{2(64)} \oplus x_{3(64)} \\
y_{2(64)} &= x_{0(64)} \oplus x_{1(64)} \oplus x_{3(64)} \\
y_{3(64)} &= x_{0(64)} \oplus x_{1(64)} \oplus x_{2(64)}
\end{aligned}$$

3 Rationales

In this part, we describe several rationales about important components building the FOX family of block ciphers.

3.1 Non-Linear Mapping sbox

As outlined earlier, our primary goal was to avoid a purely algebraic construction for the S-box; a secondary goal was the possibility to implement it in a very efficient way on hardware using ASIC or FPGA technologies. The sbox function is a non-linear bijective mapping on 8-bit values. It consists of a Lai-Massey scheme with 3 rounds taking three different substitution boxes as round function where the orthormorphism of the third round is omitted; these "small" S-boxes are denoted S_1, S_2 and S_3, and their content is given in Fig. 13. 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. We describe now the generation process of the sbox transformation.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$\mathsf{S}_1(x)$	2	5	1	9	E	A	C	8	6	4	7	F	D	B	0	3
$\mathsf{S}_2(x)$	B	4	1	F	0	3	E	D	A	8	7	5	C	2	9	6
$\mathsf{S}_3(x)$	D	A	B	1	4	3	8	9	5	7	2	C	F	0	6	E

Figure 13: The three small S-boxes of FOX.

First a set of three different candidates for small substitution boxes, each having a $\mathrm{LP}_{\max}$ and a $\mathrm{DP}_{\max}$ smaller than 2^{-2} were pseudo-randomly chosen, where

$$\begin{aligned}\mathrm{LP}^{\mathsf{f}}(\mathbf{a},\mathbf{b}) &= \left(2\Pr_X[\mathbf{a}\bullet X = \mathbf{b}\bullet \mathsf{f}_k(X)] - 1\right)^2 \\ \mathrm{LP}^{\mathsf{f}}_{\max} &= \max_{\mathbf{a},\mathbf{b}\neq\mathbf{0}} \mathrm{LP}^{\mathsf{f}}(\mathbf{a},\mathbf{b})\end{aligned}$$

with $\bullet$ denoting the scalar product over GF (2)-vectors, and

$$\begin{aligned}\mathrm{DP}^{\mathsf{f}}(a,b) &= \Pr_X\left[\mathsf{f}_k\left(X\oplus a\right) = \mathsf{f}_k(X)\oplus b\right] \\ \mathrm{DP}^{\mathsf{f}}_{\max} &= \max_{a\neq 0,b} \mathrm{DP}^{\mathsf{f}}(a,b)\end{aligned}$$

Then, the candidate sbox mapping was evaluated and tested regarding its $\mathrm{LP}_{\max}$ and $\mathrm{DP}_{\max}$ values until a good candidate was found. The chosen sbox satisfies $\mathrm{DP}^{\mathsf{sbox}}_{\max} = \mathrm{LP}^{\mathsf{sbox}}_{\max} = 2^{-4}$ and its algebraic degree is equal to 6.

3.2 Linear Multipermutations mu4/mu8

Both mu4 and mu8 are *linear multipermutations*. This kind of construction was early recognized as being optimal for which regards its diffusion properties (see [SV95, Vau95]). As explained in [JV04b], not all constructions are very efficient to implement, especially on low-end smartcard, which have usually very few available memory and computational power. We have thus chosen a circulating-like construction. Furthermore, in order to be efficiently implementable, the elements of the matrix, which are elements of $\mathrm{GF}(2^8)$, should be efficient to multiply to. The only really efficient operations are the addition, the multiplication by α and the division by α. Note that $\alpha^7+\alpha = \alpha^{-1}+\alpha^{-2}$, $\alpha^7+\alpha^6+\alpha^5+\alpha^4+\alpha^3+\alpha^2 = \alpha^{-1}$, and that $\alpha^6+\alpha^5+\alpha^4+\alpha^3+\alpha^2+\alpha = \alpha^{-2}$.

3.3 Key-Schedule Algorithms

The FOX key-schedule algorithms were designed with several rationales in mind: first, the function, which takes a key k and the round number r and returns r subkeys should be a cryptographic pseudorandom, collision resistant and one-way function. Second, the sequence of subkeys should be generated in any direction without any complexity penalty. Third, all the bytes of *mkey* should be randomized even when the key size is strictly smaller than *ek*. Finally, the key-schedule algorithm should resist *related-cipher attacks* as described by Wu in [Wu02], since FOX can possibly use different number of rounds.

We are convinced that "strong" key-schedule algorithms have significant advantages in terms of security, even if the price to pay is a smaller key agility, as discussed earlier. In the case of FOX, we believe that the time needed to compute the subkeys, which is about equal to the time needed to encrypt 6 blocks of data (in the case of FOX64 with keys strictly larger than 128 bit, it takes the time to encrypt 12 blocks of data) remains acceptable in all kinds of applications.

During the AES effort, it was suggested that an example of extreme case would be a high-speed network switch having to maintain a million of contexts and switching bewteen them every four blocks of data. Under such extreme constraints, one can still keep in memory one million fully expanded keys at a negligible cost.

The second central property of FOX key-schedule algorithms is ensured by the LFSR construction. As it is possible to back-clock it easily, the subkey generation process can be computed in the encryption as well as in the decryption direction with no loss of speed. The third property is ensured by our "Fibonacci-like" construction (which is a bijective mapping). Furthermore, *mkey* is expanded by XORing constants depending on r and ek with *no overlap* on these constants sequences (this was checked experimentally). Finally, the fourth property is ensured by the dependency of the subkey sequence to the actual round number of the algorithm instance for which the sequence will be used.

We state now a sequence of properties of the building blocks of the key-schedule algorithm.

3.3.1 P-Part

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

Lemma 3.1. *It is impossible to find two values of k with a length strictly smaller than ek 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 ℓ 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. □

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

3.3.2 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 3.2. *The M-part is a permutation.*

Proof. The lemma follows directly from the fact that the M-part is an invertible application. □

3.3.3 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 for all round numbers $1 \leq i \leq r$ are unique.

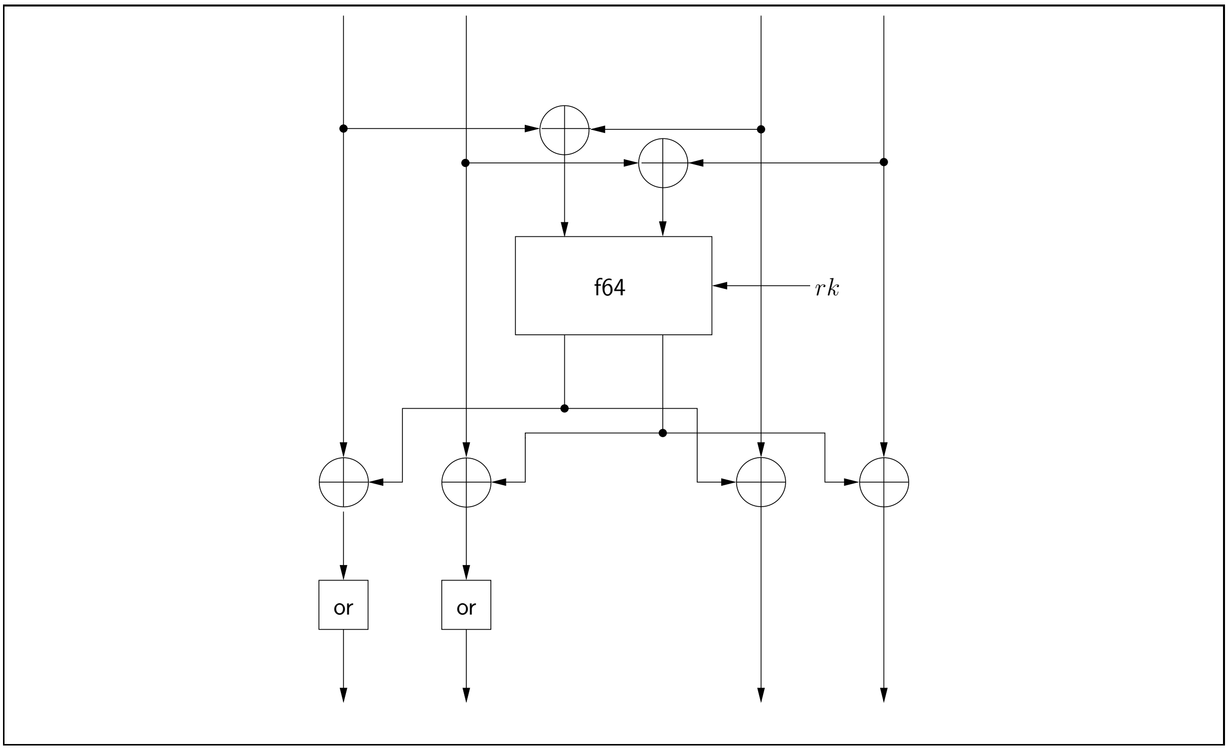

Figure 14: An alternate view of an extended Lai-Massey scheme

3.3.4 NLx-Part

The goal of the NL part is to generate a pseudorandom stream of data as "cryptographically secure" as possible and as fast as possible; it is actually the one-way part of the key-schedule. For this, it re-uses the round functions in its core, and it needs only a few supplementary operations.

3.4 Security Foundations

3.4.1 Security Properties of the Lai-Massey Scheme

Although less popular than the Feistel scheme or SPN structures, the Lai-Massey scheme offers similar (super-) pseudorandomness and decorrelation inheritance properties, as was demonstrated by Vaudenay [Vau00]. Note that we will indifferently use the term "Lai-Massey scheme" to denote both versions, as we can see the Extended Lai-Massey scheme as a Lai-Massey scheme: we can swap the two inner inputs as in Fig. 14, and we note that the function $(x, y) \mapsto \mathsf{or32}(x)||\mathsf{or32}(y)$ builds an orthomorphism (see Lem. 3.3).

Lemma 3.3. *The application defined by*

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

is an orthomorphism, where $\mathsf{or}(.)$ *is the orthomorphism defined in §2.2.3.*

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

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

and that io is a permutation, too. Now, we have to check that

$$(x, y) \mapsto (\mathsf{or}(x) \oplus x, \mathsf{or}(y) \oplus y) \tag{2}$$

is also a permutation. This follows easily from the fact that Eq. (2) is an invertible application. □

From this point, we will make use of the following notation: given an orthomorphism o on a group $(\mathcal{G}, +)$ and given r functions $\mathsf{f}_1, \mathsf{f}_2, \ldots, \mathsf{f}_r$ on $\mathcal{G}$, we note an r-rounds Lai-Massey scheme using the r functions and the orthomorphism by $\Lambda^{\mathsf{o}}(\mathsf{f}_1, \ldots, \mathsf{f}_r)$. Then the following results are two Luby-Rackoff-like [LR88] results on the Lai-Massey scheme. We refer to [Vau00, Vau03] for proofs thereof.

Theorem 3.1 (Vaudenay). *Let f_1^*, f_2^* and f_3^* be three independent random functions uniformly distributed on a group $(\mathcal{G}, +)$. Let o be an orthomorphism on $\mathcal{G}$. For any distinguisher limited to d chosen plaintexts, where $g = |\mathcal{G}|$ denotes the cardinality of the group, between $\Lambda^{\mathsf{o}}(\mathsf{f}_1^*, \mathsf{f}_2^*, \mathsf{f}_3^*)$ and a uniformly distributed random permutation c^*, we have*

$$\mathrm{Adv}(\Lambda^{\mathsf{o}}(\mathsf{f}_1^*, \mathsf{f}_2^*, \mathsf{f}_3^*), \mathsf{c}^*) \leq d(d-1)(g^{-1} + g^{-2}).$$

Theorem 3.2 (Vaudenay). *If $\mathsf{f}_1, \ldots, \mathsf{f}_r$ are $r \geq 3$ independent random functions on a group $(\mathcal{G}, +)$ of order g such that $\mathrm{Adv}(\mathsf{f}_i, \mathsf{f}_i^*) \leq \frac{\varepsilon}{2}$ for any adaptive distinguisher between f_i and f_i^* limited to d queries for $1 \leq i \leq r$ and if o is an orthomorphism on $\mathcal{G}$, we have*

$$\mathrm{Adv}(\Lambda^{\mathsf{o}}(\mathsf{f}_1, \ldots, \mathsf{f}_r), \mathsf{c}^*) \leq \frac{1}{2}(3\varepsilon + d(d-1)(2g^{-1} + g^{-2}))^{\lfloor \frac{r}{3} \rfloor}.$$

Basically, the first result proves that the Lai-Massey scheme provides pseudorandomness on three rounds unless the f_i's are weak, like for the Feistel scheme [Fei73]. Super-pseudorandomness corresponds to cases where a distinguisher can query chosen ciphertexts as well; in this scenario, the previous result holds when we consider $\Lambda^{\mathsf{o}}(\mathsf{f}_1^*, \ldots, \mathsf{f}_4^*)$ with a fourth round. The second result proves that the decorrelation bias of the round functions of a Lai-Massey scheme is inherited by the whole structure: provided the f_i's are strong, so is the Lai-Massey scheme; in other words, a potential cryptanalysis will not be able to exploit the Lai-Massey's scheme only, but it will have to take advantage of weaknesses of the round functions' internal structure. We would like to stress out the importance of the orthomorphism o: by omitting it, it is possible to distinguish a Lai-Massey scheme using pseudorandom functions from a pseudorandom permutation with overwhelming probability, and this for any number of rounds. Indeed, denoting the input and the output of a Lai-Massey scheme by $x_{\mathsf{l}}||x_{\mathsf{r}}$ and $y_{\mathsf{l}}||y_{\mathsf{r}}$, respectively, the following equation holds with probability one:

$$x_{\mathsf{l}} \ominus x_{\mathsf{r}} = y_{\mathsf{l}} \ominus y_{\mathsf{r}} \tag{3}$$

where $\ominus$ denote the inverse of the additive group law used in the scheme.

One should not misinterpret the results in the Luby-Rackoff scenario in terms of the overall block cipher security: FOX's round functions are far to be indistinguishable from random functions, as it is the case of DES round functions, for instance: the fact that DES is vulnerable to linear and differential cryptanalysis does not contradict Luby-Rackoff results. However, Th. 3.1 and Th. 3.2 give proper credit to the high-level structure of FOX.

3.4.2 Resistance w/r to Linear and Differential Cryptanalysis

It is possible to prove some important results about the security of both $\mathsf{f32}$ and $\mathsf{f64}$ functions towards linear and differential cryptanalysis, too. 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$^+$01] by Hong *et al.* For the sake of completeness, we recall the framework of consideration and the results they obtained using it. Then, we apply their result to the round functions of FOX, and we draw some conclusions about its security towards linear and differential cryptanalysis in functions of the round number. This will help us to fix the minimal number of rounds which results in a sufficient level of security.

Let S_i denote an $m \times m$ bijective substitution box, that is a bijection on $\{0,1\}^m$. We consider a standard kSPkSk structure (i.e. the one of FOX's round functions) on $m \times n$ bit strings, namely a key addition layer, a substitution layer, a diffusion layer, followed by a second key addition layer, a substitution layer, and a final key addition layer. We assume that the substitution layer consists of the parallel evaluation of n $m \times m$ S-boxes S_i for $1 \leq i \leq n$, that the diffusion layer can be expressed as an invertible $n \times n$ MDS matrix $\mathbf{M}$ with coefficients in $\mathrm{GF}(2^m)$, and that the key addition layer consists of XORing a mn-bit subkey to the state. Let us furthermore denote by

$$\pi_{\mathrm{DP}}^{\mathsf{S}} = \max_{1 \leq i \leq n} \mathrm{DP}_{\max}^{\mathsf{S}_i} \text{ and } \pi_{\mathrm{LP}}^{\mathsf{S}} = \max_{1 \leq i \leq n} \mathrm{LP}_{\max}^{\mathsf{S}_i}$$

the respective maximal differential and linear probabilities we can find in the S-boxes S_i. Finally, let us denote by

$$\beta = \mathfrak{B}(\mathbf{M}) = n + 1$$

the branch number of the diffusion layer $\mathbf{M}$ (according to [DR02]), which is defined to be maximal. Then the following theorem due to Hong. *et al.* [HLL$^+$01] states upper bounds on the maximal differential and linear hull probabilities, respectively.

Theorem 3.3 (Hong *et al.*[HLL$^+$01]). *In a* kSPkSk *structure, if the round subkeys are statistically independent and uniformly distributed, then the probability of each differential with respect to $\oplus$ is upper bounded by*

$$\left(\pi_{\mathrm{DP}}^{\mathsf{S}}\right)^{\beta-1},$$

while the probability of each linear hull is upper bounded by

$$\left(\pi_{\mathrm{LP}}^{\mathsf{S}}\right)^{\beta-1}.$$

In the case of FOX64, since $\mathrm{DP}_{\max}^{\mathsf{sbox}} = \mathrm{LP}_{\max}^{\mathsf{sbox}} = 2^{-4}$, since mu4 (resp. mu8) has a branch number equal to five (resp. nine), and since one can assume that the subkeys are uniformly distributed and statistically independent, due to the nature of the key-schedule algorithm, one can reasonably apply Th. 3.3 and get the following result.

Theorem 3.4. *If the round subkeys are statistically independent and uniformly distributed, then the following bounds hold:*

$$\mathrm{LP}_{\max}^{\mathsf{f32}} = \mathrm{DP}_{\max}^{\mathsf{f32}} \leq 2^{-16},$$

and

$$\mathrm{LP}_{\max}^{\mathsf{f64}} = \mathrm{DP}_{\max}^{\mathsf{f64}} \leq 2^{-32}.$$

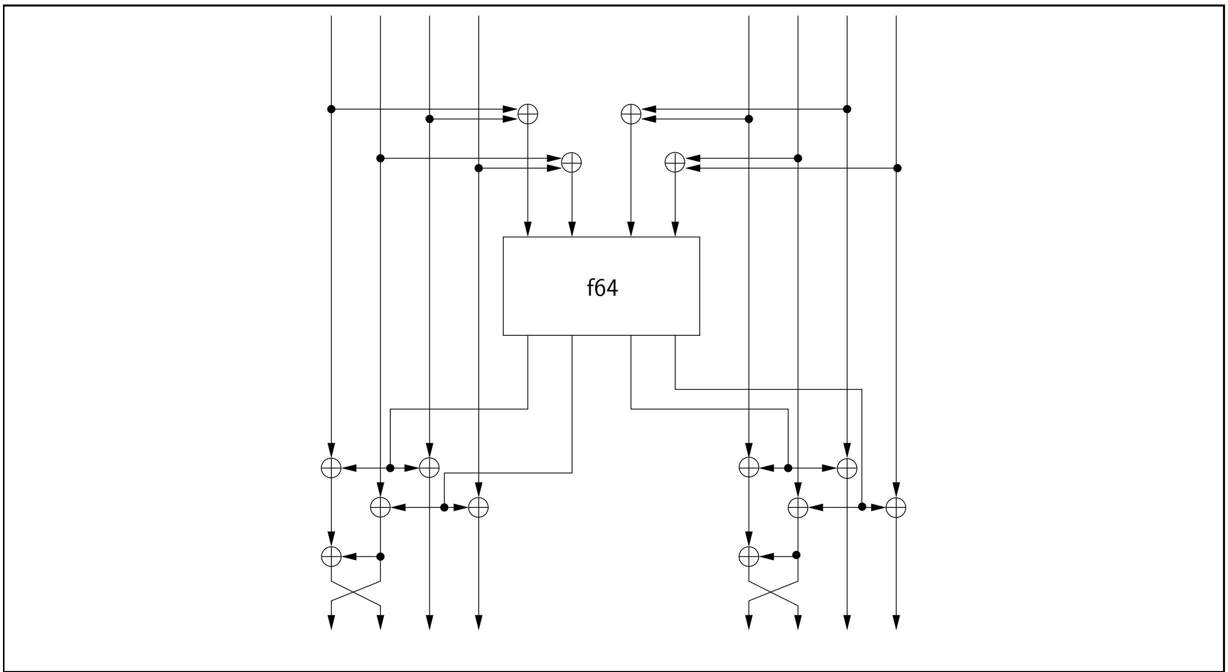

Figure 15: A detailed view of an extended Lai-Massey scheme

Let us now focus on embedding the round functions in the skeletons. For the sake of clarity[3], we prove now some interesting properties of an Extended Lai-Massey scheme regarding differential and linear characteristics.

Lemma 3.4. *In the Extended Lai-Massey scheme as defined in §2.1.2, any* differential *characteristic on two rounds must involve at least one* f64*-function.*

Proof. We follow a top-down approach. If we stack up two rounds of an Extended Lai-Massey scheme (see Fig. 15 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. □

Lemma 3.5. *In the Extended Lai-Massey scheme as defined in Fig. 2.1.2, any* linear *characteristic on two rounds must involve at least one function* f64*.*

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 $c = d = 0$, which is a contradiction to our assumption, and the theorem follows. □

[3]These properties are actually trivial to prove in the case of a simple Lai-Massey scheme, and as discussed in §3.4.1, the Extended Lai-Massey scheme can be viewed as a simple Lai-Massey scheme.

By considering Th. 3.3, Lem. 3.4, and Lem. 3.5 together, we have thus the following result.

Theorem 3.5. *The differential (resp. linear) probability of any single-path characteristic in* FOX64$/k/r$ *is upper bounded by* $(\mathrm{DP}_{\max}^{\mathsf{sbox}})^{2r}$ *(resp.* $(\mathrm{LP}_{\max}^{\mathsf{sbox}})^{2r}$*). Similarly, the bounds are* $(\mathrm{DP}_{\max}^{\mathsf{sbox}})^{4r}$ *(resp.* $(\mathrm{LP}_{\max}^{\mathsf{sbox}})^{4r}$*) for* FOX128$/k/r$.

Note that it is a kind of "hybrid" proof of security towards linear and differential cryptanalysis, as we have considered differential and linear hulls in the round functions, but characteristics in the high-level schemes. Thus, we have in reality slightly stronger results that the ones stated in Th. 3.5. Finally, we conclude that it is impossible to find any useful differential or linear characteristic after 8 rounds for both FOX64 and FOX128. Hence, a minimal number of 12 rounds provides a minimal safety margin.

3.4.3 Resistance Towards Other Attacks

In this part, we discuss the resistance of FOX towards various types of attacks.

Statistical Attacks Due to the very high diffusion properties of FOX's round functions, the high algebraic degree of the sbox mapping, and the high number of rounds, we are strongly convinced that FOX will resist to known variants of linear and differential cryptanalysis (like differential-linear cryptanalysis [LH94, BDK02], boomerang [Wag99] and rectangle [BDK01] attacks), as well as generalizations thereof, like Knudsen's truncated and higher-order differentials [Knu95], impossible differentials [BBS99], and Harpes' partitioning cryptanalysis [HM97], for instance.

Slide and Related-Key Attacks Slide attacks [BW99, BW00] exploit periodic key-schedule algorithms, which is not a property of FOX's key-schedule algorithms. Furthermore, due to very good diffusion and the high non-linearity of the key-schedule, related-key attacks are very unlikely to be effective against FOX.

Interpolation and Algebraic Attacks Interpolation attacks [JK97] take advantage of S-boxes exhibiting a simple algebraic structure. Since FOX's non-linear mapping sbox does not possess any simple relation over GF(2) or GF(2^8), such attacks are certainly not effective.

One of our main concerns was to avoid a pure algebraic construction for the sbox mapping, as it is the case for a large number of modern designs of block ciphers. Although such S-boxes have many interesting non-linear properties, they probably form the best conditions to express a block cipher as a system of sparse, over-defined low-degree multivariate polynomial equations over GF(2) or GF(2^8); this fact may lead to effective attacks, as argued by Courtois and Pieprzyk in [CP02].

Not choosing an algebraic construction for sbox does not necessarily ensure security towards algebraic attacks. Note that we base our non-linear mapping on "small" permutations, mapping 4 bits to 4 bits, and that, according to [CP02], *any* such mapping can always be written as an overdefined system of *at least* 21 quadratic equations: let us denote the input (resp. the output) of such a small S-box by $x_1||x_2||x_3||x_4$ (resp. by $y_1||y_2||y_3||y_4$), and if we consider a 16×37 matrix containing in each row the values of the $t = 37$ monomials

$$\{1, x_1, \ldots, x_4, y_1, \ldots, y_4, x_1x_2, \ldots, x_1y_1, \ldots, y_3y_4\}$$

for each of the 16 possible entries, we note that its rank can be at most 16, thus, for any S-box, there will be at least $\rho \geq 37 - 16 = 21$ quadratic equations. We have checked that the rank of these matrices for FOX's small S-boxes S_1, S_2, and S_3 are equal to 16, and there exist thus 21 quadratic equations describing it; furthermore, we are not aware of any quadratic relation over GF $\left(2^8\right)$ for sbox. Following the very same methodology than [CP02], it appears that XSL attacks *would* break members of the FOX family within a complexity[4] of 2^{139} to 2^{156}, depending on the block size and on the rounds number.

Namely, we can construct an overdefined multivariate system of quadratic equations describing FOX using the XSL approach, which aims at recovering all the subkeys, without taking care of the key-schedule algorithm. Let us assume that FOX has r rounds, and thus r subkeys with the same size than the plaintext. We need hence r known plaintext-ciphertext pairs to uniquely determine the key. We use from now on the same notations than in [CP02]. S is defined to be the total number of substitution boxes considered during an attack. Hence,

$$S_{\mathsf{FOX64}} = 3 \cdot 8 \cdot r^2$$

for FOX64, and

$$S_{\mathsf{FOX128}} = 3 \cdot 16 \cdot r^2$$

as each substitution box sbox is built from three small S-boxes on $\{0,1\}^s$, with $s = 4$. Let t denote the number of monomials (i.e. $t = 37$ in our case), let t' being the number of terms in the basis for one S-box that can be multiplied by some fixed variable and are still in the basis (we have $t' = 5$ in the case of FOX). Then, Courtois and Pieprzyk [CP02] estimate that the complexity of a XSL attack can be estimated to

$$T^{\omega} \text{ with } T \approx (t-\rho)^P \cdot \binom{S}{P}$$

where ω is the best possible exponent for Gaussian elimination, T represents the total number of terms, and where

$$P = \frac{t-\rho}{s + \frac{t'}{S}}$$

In the case of FOX, we get

$$P = \frac{16}{4 + \frac{5}{24r^2}} < 4$$

According to Courtois and Pieprzyk [CP02], in order that the attack works, as difference operation) it is necessary to choose P such that

$$\frac{R}{T - T'} \geq 1 \tag{4}$$

where

$$R \approx S \cdot s(t-\rho)^{P-1} \cdot \binom{S}{P-1}$$

represents the total number of equations, and

$$T' \approx t'(t-\rho)^{P-1} \cdot \binom{S-1}{P-1}$$

is the total number of terms in the basis that can be multiplied by some fixed variable and are still in the basis. Eq. (4), in the case which interests us, is already fullfiled for $P = 4$, but $R \approx 1$. As the overall complexity of the attack is very sensitive to the value of P, and according to Courtois and Pieprzyk [CP02],

[4] Under the most pessimistic hypotheses.

		$P=4$		$P=5$	
	S	$\omega=2.376$	$\omega=3$	$\omega=2.376$	$\omega=3$
FOX64/k/12	3456	2^{139}	2^{175}	2^{171}	2^{216}
FOX64/k/16	6144	2^{147}	2^{185}	2^{181}	2^{228}
FOX128/k/12	6912	2^{148}	2^{187}	2^{183}	2^{231}
FOX128/k/16	12288	2^{156}	2^{197}	2^{192}	2^{243}

Figure 16: Estimations of the complexity of Courtois-Pieprzyk attacks against FOX

> *Though XSL attacks will probably always work for some P, we considered the minimum value P for which $\frac{R}{T-T'} \geq 1$. This condition is necessary, but probably not sufficient.*

we will consider the cases $P=4$ as well as $P=5$ in our estimations of the complexity of applying algebraic attacks to FOX.

Another subject of controversy is the value of ω, i.e. the complexity exponent of a Gaussian reduction. Courtois and Pieprzyk [CP02] assume that $\omega = 2.376$, which is the best known value obtained by Coppersmith and Winograd [CW90]. According to [CP02], the constant factor in this algorithm is unknown to the authors of [CW90], and is expected to be very big. Accordingly, it is disputed whether such an algorithm can be applied efficiently in practice. For this reason, we will consider both $\omega = 2.376$ and $\omega = 3$ in our estimations.

A summary of our estimations is given in Fig. 16. At the light of the previous discussion, we should interpret these figures with an extreme care: on the one hand, the real complexity of XSL attacks is by no means clear at the time of writing and is the subject of much controversy [MR03]; one the other hand, we feel that the advantages of a small hardware footprint overcome such a (possible) security decrease.

Integral Attacks Integral attacks [KW02] apply to ciphers operating on well-aligned data, like SPN structures. As the round functions of FOX are SPNs, one can wonder whether it is possible to find an integral distinguisher on the whole structure and we show now that it is indeed the case. Let us consider the case of FOX64: we denote the input bytes by $x_{i(8)}$ with $0 \leq i \leq 7$ and the output of the third round lmid64 by $y_{i(8)}$ with $0 \leq i \leq 7$. We have the following integral distinguisher on 3 rounds of FOX64.

Theorem 3.6. *Let $x_{3(8)} = a$, $x_{7(8)} = a \oplus c$, and $x_{i(8)} = c$ for $i = 0,1,2,4,5,6$, where c is an arbitrary constant. We consider plaintext structures $x^{(j)}$ for $1 \leq j \leq 256$ where a takes all 256 possibles byte values. Then,*

$$\bigoplus_{j=1}^{256} y_0^{(j)} \oplus y_6^{(j)} = 0 \text{ and } \bigoplus_{j=1}^{256} y_1^{(j)} \oplus y_7^{(j)} = 0$$

as well as

$$\bigoplus_{j=1}^{256} y_0^{(j)} \oplus y_2^{(j)} \oplus y_4^{(j)} = 0 \text{ and } \bigoplus_{j=1}^{256} y_1^{(j)} \oplus y_3^{(j)} \oplus y_5^{(j)} = 0.$$

Proof. See Fig. 17, where "C" denotes a constant byte, "A" denotes an active byte, and "S" denotes a byte, whose sum under the structure is equal to zero. □

This integral distinguisher can be used to break (four, five) six rounds of FOX64 (by guessing the one, two, or three last round keys and testing the integral criterion for each subkey candidate on a few structures of plaintexts) within a complexity of about (2^{72}, 2^{136}) 2^{200} partial decryptions and negligible memory. A similar property may be used to break up to 4 rounds of FOX128 (by guessing the last round key) with a complexity of about 2^{136} operations and negligible memory.

4 Implementation

In this part, we discuss several issues about the implementation of the FOX family on low-end 8-bit architectures and on high-end 32/64-bit ones. Finally, we give results about the performances of various implementations we have written on different platforms.

4.1 8-bit Architectures

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.

4.1.1 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 four different (the last one concerning uniquely FOX128) strategies using various amounts of precomputed data to implement these mappings; they are summarized in Fig. 18. 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 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 §3.1, page 24, and all the operations in GF $\left(2^8\right)$. The sole needed constants are the small substitution boxes S_1, S_2 and S_3 (see Fig. 13). Strategy A is clearly the slowest one. 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: $\mathsf{talpha}(x)$ is a function mapping an element x to $\alpha \cdot x$, with the multiplication in GF $\left(2^8\right)$; $\mathsf{dalpha}(x)$ is a function mapping an element $x \in \mathrm{GF}\left(2^8\right)$ 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:

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

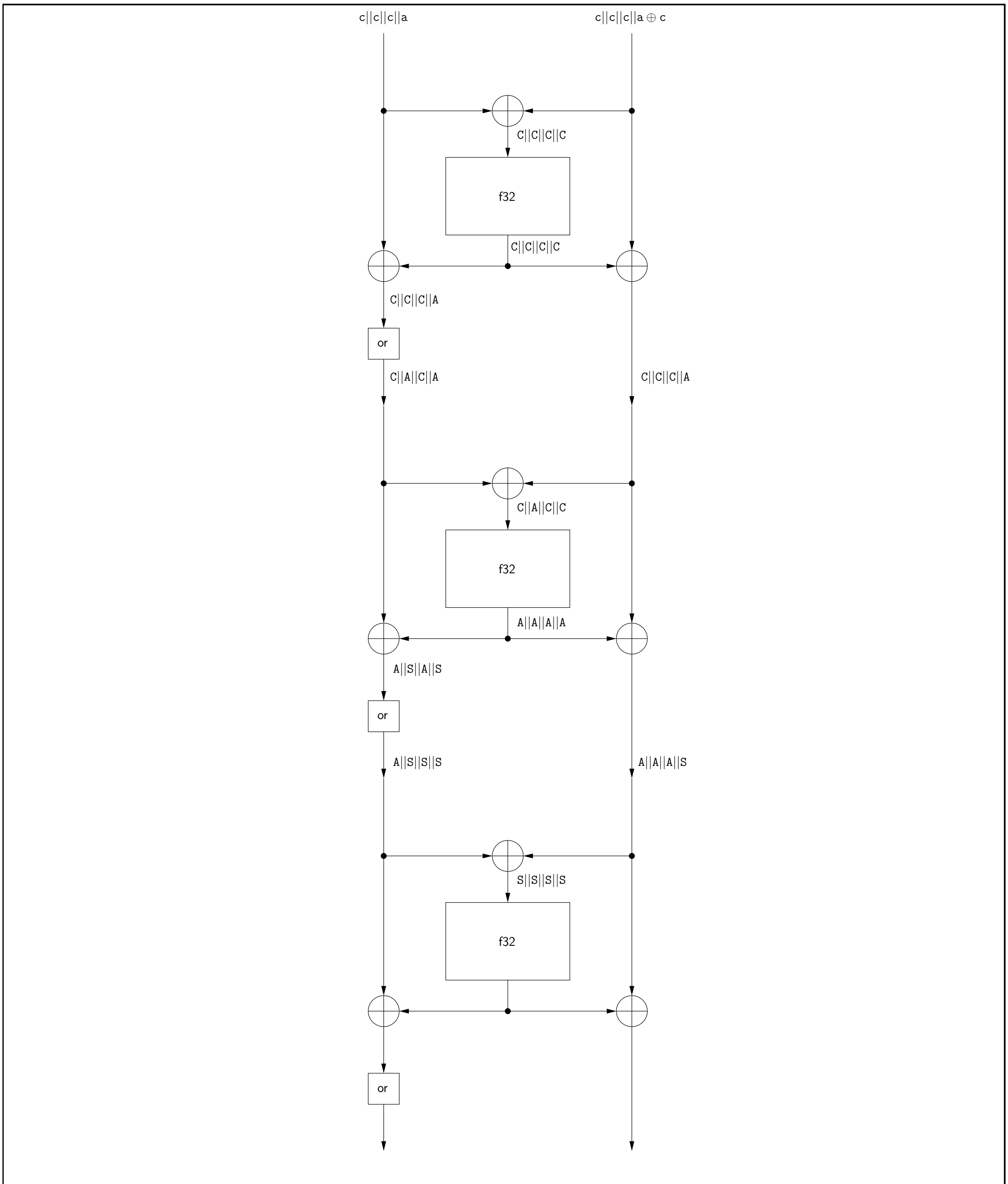

Figure 17: Integral Distinguisher in 3 rounds of FOX64.

Strategy	Precomputations	Data size
A	No precomputed data	24 B
B	sbox	256 B
C	sbox, talpha, dalpha	768 B
D	sbox, stalpha, sdalpha, stalpha2, sdalpha2	1280 B

Figure 18: Four different strategies to implement FOX on low-end microprocessors

The implementation of the sigma4/mu4 layer is relatively straighforward:

$$\begin{aligned}
y_{0(8)} &= \mathsf{sbox}(x_{0(8)}) \oplus \mathsf{sbox}(x_{1(8)}) \oplus \mathsf{sbox}(x_{2(8)}) \oplus \\
&\quad \alpha \cdot \mathsf{sbox}(x_{3(8)}) \\
y_{1(8)} &= \mathsf{sbox}(x_{0(8)}) \oplus \mathsf{sbox}(x_{1(8)}) \oplus \mathsf{sbox}(x_{3(8)}) \oplus \alpha \cdot \mathsf{sbox}(x_{2(8)}) \\
&\quad \oplus \alpha^{-1} \cdot \mathsf{sbox}(x_{1(8)}) \\
y_{2(8)} &= \mathsf{sbox}(x_{0(8)}) \oplus \mathsf{sbox}(x_{2(8)}) \oplus \mathsf{sbox}(x_{3(8)}) \oplus \alpha \cdot \mathsf{sbox}(x_{1(8)}) \\
&\quad \oplus \alpha^{-1} \cdot \mathsf{sbox}(x_{0(8)}) \\
\\
y_{3(8)} &= \mathsf{sbox}(x_{1(8)}) \oplus \mathsf{sbox}(x_{2(8)}) \oplus \mathsf{sbox}(x_{3(8)}) \oplus \alpha \cdot \mathsf{sbox}(x_{0(8)}) \\
&\quad \oplus \alpha^{-1} \cdot \mathsf{sbox}(x_{2(8)})
\end{aligned}$$

By carefully rewriting the above equations and by re-using some temporary results, one can easily minimize the number of sbox, talpha, dalpha evaluations and the number of $\oplus$ operations. However, the resulting implementation is strongly dependent of the chosen strategy.

The implementation of the sigma8/mu8 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)} &= \mathsf{sbox}(x_{0(8)}) \oplus \mathsf{sbox}(x_{1(8)}) \oplus \mathsf{sbox}(x_{2(8)}) \oplus \mathsf{sbox}(x_{3(8)}) \oplus \\
&\quad \mathsf{sbox}(x_{4(8)}) \oplus \mathsf{sbox}(x_{5(8)}) \oplus \mathsf{sbox}(x_{6(8)}) \oplus \alpha \cdot \mathsf{sbox}(x_{7(8)}) \\
y_{1(8)} &= \mathsf{sbox}(x_{0(8)}) \oplus \mathsf{sbox}(x_{1(8)}) \oplus \mathsf{sbox}(x_{7(8)}) \oplus \\
&\quad \alpha \cdot \big(\mathsf{sbox}(x_{1(8)}) \oplus \mathsf{sbox}(x_{3(8)}) \oplus \alpha \cdot \mathsf{sbox}(x_{4(8)}\big) \oplus \\
&\quad \alpha^{-1} \cdot \big(\mathsf{sbox}(x_{2(8)}) \oplus \mathsf{sbox}(x_{5(8)}) \oplus \alpha^{-1} \cdot (\mathsf{sbox}(x_{2(8)}) \oplus \mathsf{sbox}(x_{6(8)}))\big) \\
y_{2(8)} &= \mathsf{sbox}(x_{0(8)}) \oplus \mathsf{sbox}(x_{6(8)}) \oplus \mathsf{sbox}(x_{7(8)}) \oplus \\
&\quad \alpha \cdot \big(\mathsf{sbox}(x_{0(8)}) \oplus \mathsf{sbox}(x_{2(8)}) \oplus \alpha \cdot \mathsf{sbox}(x_{3(8)}\big) \oplus \\
&\quad \alpha^{-1} \cdot \big(\mathsf{sbox}(x_{1(8)}) \oplus \mathsf{sbox}(x_{4(8)}) \oplus \alpha^{-1} \cdot (\mathsf{sbox}(x_{1(8)}) \oplus \mathsf{sbox}(x_{5(8)}))\big) \\
y_{3(8)} &= \mathsf{sbox}(x_{5(8)}) \oplus \mathsf{sbox}(x_{6(8)}) \oplus \mathsf{sbox}(x_{7(8)}) \oplus \\
&\quad \alpha \cdot \big(\mathsf{sbox}(x_{1(8)}) \oplus \mathsf{sbox}(x_{6(8)}) \oplus \alpha \cdot \mathsf{sbox}(x_{2(8)}\big) \oplus \\
&\quad \alpha^{-1} \cdot \big(\mathsf{sbox}(x_{0(8)}) \oplus \mathsf{sbox}(x_{3(8)}) \oplus \alpha^{-1} \cdot (\mathsf{sbox}(x_{0(8)}) \oplus \mathsf{sbox}(x_{4(8)}))\big) \\
y_{4(8)} &= \mathsf{sbox}(x_{4(8)}) \oplus \mathsf{sbox}(x_{5(8)}) \oplus \mathsf{sbox}(x_{7(8)}) \oplus \\
&\quad \alpha \cdot \big(\mathsf{sbox}(x_{0(8)}) \oplus \mathsf{sbox}(x_{5(8)}) \oplus \alpha \cdot \mathsf{sbox}(x_{1(8)}\big) \oplus \\
&\quad \alpha^{-1} \cdot \big(\mathsf{sbox}(x_{2(8)}) \oplus \mathsf{sbox}(x_{6(8)}) \oplus \alpha^{-1} \cdot (\mathsf{sbox}(x_{3(8)}) \oplus \mathsf{sbox}(x_{6(8)}))\big) \\
y_{5(8)} &= \mathsf{sbox}(x_{3(8)}) \oplus \mathsf{sbox}(x_{4(8)}) \oplus \mathsf{sbox}(x_{7(8)}) \oplus \\
&\quad \alpha \cdot \big(\mathsf{sbox}(x_{4(8)}) \oplus \mathsf{sbox}(x_{6(8)}) \oplus \alpha \cdot \mathsf{sbox}(x_{0(8)}\big) \oplus \\
&\quad \alpha^{-1} \cdot \big(\mathsf{sbox}(x_{1(8)}) \oplus \mathsf{sbox}(x_{5(8)}) \oplus \alpha^{-1} \cdot (\mathsf{sbox}(x_{2(8)}) \oplus \mathsf{sbox}(x_{5(8)}))\big) \\
y_{6(8)} &= \mathsf{sbox}(x_{2(8)}) \oplus \mathsf{sbox}(x_{3(8)}) \oplus \mathsf{sbox}(x_{7(8)}) \oplus \\
&\quad \alpha \cdot \big(\mathsf{sbox}(x_{3(8)}) \oplus \mathsf{sbox}(x_{5(8)}) \oplus \alpha \cdot \mathsf{sbox}(x_{6(8)}\big) \oplus \\
&\quad \alpha^{-1} \cdot \big(\mathsf{sbox}(x_{0(8)}) \oplus \mathsf{sbox}(x_{4(8)}) \oplus \alpha^{-1} \cdot (\mathsf{sbox}(x_{1(8)}) \oplus \mathsf{sbox}(x_{4(8)}))\big) \\
y_{7(8)} &= \mathsf{sbox}(x_{1(8)}) \oplus \mathsf{sbox}(x_{2(8)}) \oplus \mathsf{sbox}(x_{7(8)}) \oplus \\
&\quad \alpha \cdot \big(\mathsf{sbox}(x_{2(8)}) \oplus \mathsf{sbox}(x_{4(8)}) \oplus \alpha \cdot \mathsf{sbox}(x_{5(8)}\big) \oplus \\
&\quad \alpha^{-1} \cdot \big(\mathsf{sbox}(x_{3(8)}) \oplus \mathsf{sbox}(x_{6(8)}) \oplus \alpha^{-1} \cdot (\mathsf{sbox}(x_{0(8)}) \oplus \mathsf{sbox}(x_{3(8)}))\big)
\end{aligned}$$

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

We give now a constant-time implementation of talpha and dalpha. The routines talpha2 and dalpha2 can be implemented by iterating twice talpha and dalpha, respectively. Note that these implementations do 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
```

4.2 32/64-bit Architectures

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).

4.2.1 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.

4.2.2 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 the temporary result obtained after the mu4 application by $t_{0(8)}||t_{1(8)}||t_{2(8)}||t_{3(8)}$. Let $rk_{0(8)}||rk_{1(8)}||rk_{2(8)}||rk_{3(8)}$ denote the first half of the round key. Finally, let $v_{i(8)} = x_{i(8)} \oplus rk_{i(8)}$ for $0 \leq i \leq 3$. We have

$$\begin{pmatrix} t_{0(8)} \\ t_{1(8)} \\ t_{2(8)} \\ t_{3(8)} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \alpha \\ 1 & c & \alpha & 1 \\ c & \alpha & 1 & 1 \\ \alpha & 1 & c & 1 \end{pmatrix} \times \begin{pmatrix} \mathsf{sbox}(v_{0(8)}) \\ \mathsf{sbox}(v_{1(8)}) \\ \mathsf{sbox}(v_{2(8)}) \\ \mathsf{sbox}(v_{3(8)}) \end{pmatrix}$$

This equation may be rewritten as

$$\begin{aligned} \begin{pmatrix} t_{0(8)} \\ t_{1(8)} \\ t_{2(8)} \\ t_{3(8)} \end{pmatrix} &= \mathsf{sbox}(v_{0(8)}) \times \begin{pmatrix} 1 \\ 1 \\ c \\ \alpha \end{pmatrix} \oplus \mathsf{sbox}(v_{1(8)}) \times \begin{pmatrix} 1 \\ c \\ \alpha \\ 1 \end{pmatrix} \oplus \\ & \mathsf{sbox}(v_{2(8)}) \times \begin{pmatrix} 1 \\ \alpha \\ 1 \\ c \end{pmatrix} \oplus \mathsf{sbox}(v_{3(8)}) \times \begin{pmatrix} \alpha \\ 1 \\ 1 \\ 1 \end{pmatrix} \end{aligned}$$

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

$$\mathtt{TBSM}_0[\mathsf{a}] = \begin{pmatrix} 1 \cdot \mathsf{sbox}(\mathsf{a}) \\ 1 \cdot \mathsf{sbox}(\mathsf{a}) \\ c \cdot \mathsf{sbox}(\mathsf{a}) \\ \alpha \cdot \mathsf{sbox}(\mathsf{a}) \end{pmatrix}, \qquad \mathtt{TBSM}_1[\mathsf{a}] = \begin{pmatrix} 1 \cdot \mathsf{sbox}(\mathsf{a}) \\ c \cdot \mathsf{sbox}(\mathsf{a}) \\ \alpha \cdot \mathsf{sbox}(\mathsf{a}) \\ 1 \cdot \mathsf{sbox}(\mathsf{a}) \end{pmatrix}$$

$$\mathtt{TBSM}_2[\mathsf{a}] = \begin{pmatrix} 1 \cdot \mathsf{sbox}(\mathsf{a}) \\ \alpha \cdot \mathsf{sbox}(\mathsf{a}) \\ 1 \cdot \mathsf{sbox}(\mathsf{a}) \\ c \cdot \mathsf{sbox}(\mathsf{a}) \end{pmatrix}, \qquad \mathtt{TBSM}_3[\mathsf{a}] = \begin{pmatrix} \alpha \cdot \mathsf{sbox}(\mathsf{a}) \\ 1 \cdot \mathsf{sbox}(\mathsf{a}) \\ 1 \cdot \mathsf{sbox}(\mathsf{a}) \\ 1 \cdot \mathsf{sbox}(\mathsf{a}) \end{pmatrix}$$

and write

$$\begin{pmatrix} t_{0(8)} \\ t_{1(8)} \\ t_{2(8)} \\ t_{3(8)} \end{pmatrix} = \mathtt{TBSM}_0[v_{0(8)}] \oplus \mathtt{TBSM}_1[v_{1(8)}] \oplus \mathtt{TBSM}_2[v_{2(8)}] \oplus \mathtt{TBSM}_3[v_{3(8)}]$$

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

$$\mathtt{TBS}_0[\mathsf{a}] = \begin{pmatrix} \mathsf{sbox}(\mathsf{a}) \\ 0 \\ 0 \\ 0 \end{pmatrix}, \qquad \mathtt{TBS}_1[\mathsf{a}] = \begin{pmatrix} 0 \\ \mathsf{sbox}(\mathsf{a}) \\ 0 \\ 0 \end{pmatrix}$$

$$\mathtt{TBS}_2[\mathsf{a}] = \begin{pmatrix} 0 \\ 0 \\ \mathsf{sbox}(\mathsf{a}) \\ 0 \end{pmatrix}, \qquad \mathtt{TBS}_3[\mathsf{a}] = \begin{pmatrix} 0 \\ 0 \\ 0 \\ \mathsf{sbox}(\mathsf{a}) \end{pmatrix}$$

and write

$$\begin{pmatrix} w_{0(8)} \\ w_{1(8)} \\ w_{2(8)} \\ w_{3(8)} \end{pmatrix} = \mathtt{TBS}_0[u_{0(8)}] \oplus \mathtt{TBS}_1[u_{1(8)}] \oplus \mathtt{TBS}_2[u_{2(8)}] \oplus \mathtt{TBS}_3[u_{3(8)}]$$

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	2×4	4	8192
FOX128	2×8	8	32768

Depending on the target processor, the nearest cache (i.e. the fastest memory) size may be smaller than 32768 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 by denoted *half-precomputed implementation*. This allows to reduce the fast memory needs to 4096 and 16384 bytes, respectively. Fig. 19 summarizes the best strategies for various amounts of L1 cache memory.

For most modern microprocessors (denoted by $*$ in Fig. 19), a fully-precomputed implementation of FOX64 and FOX128 is probably the fastest possible solution. For the processors denoted by $\bullet$, a half-precompted implementation is likely the best solution. The supplementary masking operations may be furthermore used to increase the instructions throughput on pipelined architectures.

Some microprocessors have a very small L1 data cache (they are denoted in $\star$ in Fig. 19). In the case of FOX128, even a half-precomputed implementation will result in many caches misses, inducing a performance penalty. For early versions of 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 8192 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. 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.

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/SSE3 instruction sets may be used to "emulate" a 64-bit microprocessor. Furthermore, by expressing the Extended Lai-Massey scheme as in Fig. 14, one can compute very efficiently the two orthomorphisms as a single one on 64-bit architectures.

In order to get the best performances for FOX implementations written in a high-level language, one can get large 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.

4.2.3 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

Processor	cache size [kB]	Note	Best Strategy
Alpha 21164	8	(data)	$\star$
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)	$\bullet$
Intel Pentium IV	8/16 (Prescott)	(data)	$\star$ / $\bullet$
Intel Xeon	8	(data)	$\star$
Intel Itanium	16	(data)	$\bullet$
Intel Itanium2	16	(data)	$\bullet$
PowerPC G4	32	(data + code)	$\bullet$
PowerPC G5	32	(data)	$*$
UltraSparc II	16	(data)	$\bullet$
UltraSparc III	64	(data)	$*$

Figure 19: Best implementation strategies on 32/64-bit microprocessors

common features.

References

[BBS99] E. Biham, A. Biryukov, and A. Shamir. Cryptanalysis of Skipjack reduced to 31 rounds using impossible differentials. In J. Stern, editor, *Advances in Cryptology – EUROCRYPT'99: International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 1999. Proceedings*, volume 1592 of *Lecture Notes in Computer Science*, pages 12–23. Springer-Verlag, 1999.

[BDK01] E. Biham, O. Dunkelman, and N. Keller. The rectangle attack - rectangling the Serpent. In B. Pfitzmann, editor, *Advances in Cryptology – EUROCRYPT 2001: International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 2001. Proceedings*, volume 2045 of *Lecture Notes in Computer Science*, pages 340–357. Springer-Verlag, 2001.

[BDK02] E. Biham, O. Dunkelman, and N. Keller. Enhancing differential-linear cryptanalysis. In Y. Zheng, editor, *Advances in Cryptology – ASIACRYPT 2002: 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1-5, 2002. Proceedings*, volume 2501 of *Lecture Notes in Computer Science*, pages 254–266. Springer-Verlag, 2002.

[BW99] A. Biryukov and D. Wagner. Slide attacks. In L. Knudsen, editor, *Fast Software Encryption: 6th International Workshop, FSE'99, Rome, Italy, March 1999. Proceedings*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259. Springer-Verlag, 1999.

[BW00] A. Biryukov and D. Wagner. Advanced slide attacks. In B. Preneel, editor, *Advances in Cryptology – EUROCRYPT 2000: International Conference on the Theory and Application of Cryptographic Techniques, Bruges, Belgium, May 2000. Proceedings*, volume 1807 of *Lecture Notes in Computer Science*, pages 589–606. Springer-Verlag, 2000.

[CP02] N. Courtois and J. Pieprzyk. Cryptanalysis of block ciphers with overdefined systems of equations. In Y. Zheng, editor, *Advances in Cryptology – ASIACRYPT 2002: 8th International Conference on the Theory and Application of Cryptology and Information Security, Queenstown, New Zealand, December 1-5, 2002. Proceedings*, volume 2501 of *Lecture Notes in Computer Science*, pages 267–287. Springer-Verlag, 2002.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

[DR02] J. Daemen and V. Rijmen. *The Design of Rijndael.* Information Security and Cryptography. Springer, 2002.

[Fei73] H. Feistel. Cryptography and data security. *Scientific American*, 228(5):15–23, 1973.

[HLL+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 B. Schneier, editor, *Fast Software Encryption: 7th International Workshop, FSE 2000, New York, NY, USA, April 2000. Proceeding*, volume 1978 of *Lecture Notes in Computer Science*, pages 273–283. Springer-Verlag, 2001.

[HM97] C. Harpes and J. Massey. Partitioning cryptanalysis. In E. Biham, editor, *Fast Software Encryption: 4th International Workshop, FSE'97, Haifa, Israel, January*

1997. Proceedings, volume 1267 of *Lecture Notes in Computer Science*, pages 13–27. Springer-Verlag, 1997.

[JK97] T. Jakobsen and L. Knudsen. The interpolation attack against block ciphers. In E. Biham, editor, *Fast Software Encryption: 4th International Workshop, FSE'97, Haifa, Israel, January 1997. Proceedings*, volume 1267 of *Lecture Notes in Computer Science*, pages 28–40. Springer-Verlag, 1997.

[JV04a] P. Junod and S. Vaudenay. FOX: a new family of block ciphers. In H. Handschuh and A. Hasan, editors, *Selected Areas in Cryptography: 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004. Revised Selected Papers*, volume 3357 of *Lecture Notes in Computer Science*, pages 114–129. Springer-Verlag, 2004.

[JV04b] P. Junod and S. Vaudenay. Perfect diffusion primitives for block ciphers - building efficient MDS matrices. In H. Handschuh and A. Hasan, editors, *Selected Areas in Cryptography: 11th International Workshop, SAC 2004, Waterloo, Canada, August 9-10, 2004. Revised Selected Papers*, volume 3357 of *Lecture Notes in Computer Science*, pages 84–99. Springer-Verlag, 2004.

[Knu95] L. Knudsen. Truncated and higher order differentials. In B. Preneel, editor, *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994. Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer-Verlag, 1995.

[KW02] L. Knudsen and D. Wagner. Integral cryptanalysis (extended abstract). In J. Daemen and V. Rijmen, editors, *Fast Software Encryption: 9th International Workshop, FSE 2002, Leuven, Belgium, February 4-6, 2002. Revised Papers*, volume 2365 of *Lecture Notes in Computer Science*, pages 112–127. Springer-Verlag, 2002.

[LH94] K. Langford and E. Hellman. Differential-linear cryptanalysis. In Y. Desmedt, editor, *Advances in Cryptology – CRYPTO '94: 14th Annual International Cryptology Conference, Santa Barbara, California, USA, August 21-25, 1994. Proceedings*, volume 839 of *Lecture Notes in Computer Science*, pages 17–25. Springer-Verlag, 1994.

[LR88] M. Luby and C. Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.

[MR03] S. Murphy and M. Robshaw. Comments on the security of the AES and the XSL technique. *Electronic Letters*, 39(1):36–38, 2003.

[SV95] C. Schnorr and S. Vaudenay. Black box cryptanalysis of hash networks based on multipermutations. In A. De Santis, editor, *Advances in Cryptology – EUROCRYPT '94: Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 1994. Proceedings*, volume 950 of *Lecture Notes in Computer Science*, pages 47–57. Springer-Verlag, 1995.

[Vau95] S. Vaudenay. On the need for multipermutations: cryptanalysis of MD4 and SAFER. In B. Preneel, editor, *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994. Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 286–297. Springer-Verlag, 1995.

[Vau00] S. Vaudenay. On the Lai-Massey scheme. In K. Lam , T. Okamoto, and C. Xing, editors, *Advances in Cryptology – ASIACRYPT '99: International Conference on the*

Theory and Application of Cryptology and Information Security, Singapore, November 14-18, 1999. Proceedings, volume 1716 of *Lecture Notes in Computer Science*, pages 8–19. Springer-Verlag, 2000.

[Vau03] S. Vaudenay. Decorrelation: a theory for block cipher security. *Journal of Cryptology*, 16(4):249–286, 2003.

[Wag99] D. Wagner. The boomerang attack. In L. Knudsen, editor, *Fast Software Encryption: 6th International Workshop, FSE'99, Rome, Italy, March 1999. Proceedings*, volume 1636 of *Lecture Notes in Computer Science*, pages 156–170. Springer-Verlag, 1999.

[Wu02] H. Wu. Related-cipher attacks. In R. Deng, S. Qing, F. Bao, and J. Zhou, editors, *Information and Communications Security: 4th International Conference, ICICS 2002, Singapore, December 9-12, 2002. Proceedings*, volume 2513 of *Lecture Notes in Computer Science*, pages 447–455. Springer-Verlag, 2002.

Test Vectors

```


FOX test vectors generator v1.2
-------------------------------



FOX64/16/64 key        : 00112233 44556677
FOX64/16/64 message    : 01234567 89ABCDEF
FOX64/16/64 ciphertext : 200E1F58 47D8A2CE
FOX64/16/64 message    : 01234567 89ABCDEF



FOX64/16/128 key        : 00112233 44556677 8899AABB CCDDEEFF
FOX64/16/128 message    : 01234567 89ABCDEF
FOX64/16/128 ciphertext : B85D6B76 6DCE952E
FOX64/16/128 message    : 01234567 89ABCDEF



FOX64/16/192 key        : 00112233 44556677 8899AABB CCDDEEFF FFEEDDCC BBAA9988
FOX64/16/192 message    : 01234567 89ABCDEF
FOX64/16/192 ciphertext : 2741D796 3406DACA
FOX64/16/192 message    : 01234567 89ABCDEF



FOX64/16/256 key        : 00112233 44556677 8899AABB CCDDEEFF FFEEDDCC BBAA9988 77665544 33221100
FOX64/16/256 message    : 01234567 89ABCDEF
FOX64/16/256 ciphertext : 8A4EDFBC 36BEF7F6
FOX64/16/256 message    : 01234567 89ABCDEF



FOX128/16/64 key        : 00112233 44556677
FOX128/16/64 message    : 01234567 89ABCDEF FEDCBA98 76543210
FOX128/16/64 ciphertext : 1EECBC7D EB66E7DA E1A7876D 90C0B239
FOX128/16/64 message    : 01234567 89ABCDEF FEDCBA98 76543210



FOX128/16/128 key        : 00112233 44556677 8899AABB CCDDEEFF
FOX128/16/128 message    : 01234567 89ABCDEF FEDCBA98 76543210
FOX128/16/128 ciphertext : 849E0F06 82F50CD5 88AE0730 06A10BEE
FOX128/16/128 message    : 01234567 89ABCDEF FEDCBA98 76543210
```

```
FOX128/16/192 key        : 00112233 44556677 8899AABB CCDDEEFF FFEEDDCC BBAA9988
FOX128/16/192 message    : 01234567 89ABCDEF FEDCBA98 76543210
FOX128/16/192 ciphertext : 5934214E CBA2D5FD 58C261B2 8261B1BC
FOX128/16/192 message    : 01234567 89ABCDEF FEDCBA98 76543210



FOX128/16/256 key        : 00112233 44556677 8899AABB CCDDEEFF FFEEDDCC BBAA9988 77665544 33221100
FOX128/16/256 message    : 01234567 89ABCDEF FEDCBA98 76543210
FOX128/16/256 ciphertext : 45CCB103 0F67B768 247F5302 66BC4996
FOX128/16/256 message    : 01234567 89ABCDEF FEDCBA98 76543210

```

Reference Implementations

File README

```
FOX / Reference implementation v1.2
Pascal Junod <pascal@junod.info>
--------------------------------------------------

The sole purpose of this reference code is to output
a set of test vectors and to help understanding the
structure of FOX. It is not fast, not portable, not
elegant and not secure. It implements a full
precomputed table-lookup strategy.

The code has been written for the IA32 architecture,
which is a little-endian architecture. It won't work
on a big-endian architecture.

Acknowledgments are due to Mounir Idrassi, Patrick Lattman,
Marco Macchetti, Emmanuel Prouff, and Chen Wenyu for their
help during the debugging process.
```

File Makefile

```
############################################################
## FOX project / Reference implementation v1.2            ##
## Pascal Junod <pascal@junod.info>                       ##
##                                                        ##
############################################################

EXEC_NAME =             fox_util

CFLAGS =                -W -Wall -pedantic -g

objects =               fox128.o fox64.o fox_ctx.o fox_cst.o fox_util.o

all:                    $(objects)
                        $(CC) -o $(EXEC_NAME) $(objects)

$(objects):             %.o: %.c %.h
                        $(CC) -c $(CFLAGS) $< -o $@

.PHONY:                 clean debug

clean:
                        -rm -f $(objects) *~ $(EXEC_NAME)

```

File fox_portable.h

```
/*****************************************************************************/
```

```c
/* FOX project / Reference implementation v1.2                            */
/* Pascal Junod <pascal@junod.info>                                        */
/*                                                                         */
/* Base file is "nessie.h"                                                 */
/***************************************************************************/

#ifndef _FOX_PORTABLE_H_
#define _FOX_PORTABLE_H_

#include <limits.h>

typedef signed char sint8;
typedef unsigned char uint8;

#if UINT_MAX >= 4294967295UL

typedef signed short sint16;
typedef signed int sint32;
typedef unsigned short uint16;
typedef unsigned int uint32;

#define ONE32   0xffffffffU

#else

typedef signed int sint16;
typedef signed long sint32;
typedef unsigned int uint16;
typedef unsigned long uint32;

#define ONE32   0xffffffffUL

#endif

#define ONE8    0xffU
#define ONE16   0xffffU

#define T08(x)   ((x) & ONE8)
#define T016(x)  ((x) & ONE16)
#define T032(x)  ((x) & ONE32)

#define EXTRACT8_BIT(d, b)      ((((uint8)(d))  & ((uint8)0x1  << (b))) >> (b))
#define EXTRACT16_BIT(d, b)     ((((uint16)(d)) & ((uint16)0x1 << (b))) >> (b))
#define EXTRACT32_BIT(d, b)     ((((uint32)(d)) & ((uint32)0x1 << (b))) >> (b))

#define ROTL8(v, n)     ((uint8)((v)  << (n)) | ((uint8)(v)  >> (8  - (n))))
#define ROTL16(v, n)    ((uint16)((v) << (n)) | ((uint16)(v) >> (16 - (n))))
#define ROTL32(v, n)    ((uint32)((v) << (n)) | ((uint32)(v) >> (32 - (n))))

/* U8T032_BIG(c) returns the 32-bit value stored in big-endian convention    */
/* in the unsigned char array pointed to by c.                               */

#define U8T032_BIG(c)  (((uint32)T08(*(c)) << 24) | ((uint32)T08(*((c) + 1)) << 16) | \
                       ((uint32)T08(*((c) + 2)) << 8) |\
                       ((uint32)T08(*((c) + 3))))

/* U8T032_LITTLE(c) returns the 32-bit value stored in little-endian         */
/* convention in the unsigned char array pointed to by c.                    */

#define U8T032_LITTLE(c)  (((uint32)T08(*(c))) | ((uint32)T08(*((c) + 1)) << 8) | \
                      ((uint32)T08(*((c) + 2)) << 16) | ((uint32)T08(*((c) + 3)) << 24))


/* U32T08_BIG(c, v) stores the 32-bit-value v in big-endian convention       */
/* into the unsigned char array pointed to by c.                             */

#define U32T08_BIG(c, v)    do { \
                uint32 x = (v); \
                uint8 *d = (c); \
                d[0] = T08(x >> 24); \
```

```
                d[1] = TO8(x >> 16); \
                d[2] = TO8(x >> 8); \
                d[3] = TO8(x); \
        } while (0)

/* U32TO8_LITTLE(c, v) stores the 32-bit-value v in little-endian          */
/* convention into the unsigned char array pointed to by c.                */


#define U32TO8_LITTLE(c, v)    do { \
                uint32 x = (v); \
                uint8 *d = (c); \
                d[0] = TO8(x); \
                d[1] = TO8(x >> 8); \
                d[2] = TO8(x >> 16); \
                d[3] = TO8(x >> 24); \
        } while (0)

#endif   /* _FOX_PORTABLE_H_                                               */
```

File fox_error.h

```
/*****************************************************************************/
/* FOX project / Reference implementation v1.2                              */
/* Pascal Junod <pascal@junod.info>                                          */
/*                                                                           */
/*****************************************************************************/

#ifndef _FOX_ERROR_H_
#define _FOX_ERROR_H_

#define FOX_ERROR_MEMORY_ALLOC        "\nError: memory allocation"
#define FOX_ERROR_CONTEXT_INIT        "\nError: context initialization"
#define FOX_ERROR_TABLE_INIT          "\nError: table initialization"
#define FOX_ERROR_KEY_INIT            "\nError: key initialization"
#define FOX_ERROR_UNKNOWN_TABLE_ID    "\nError: unknown table ID"
#define FOX_ERROR_UNKNOWN_MODE        "\nError: unknown mode"
#define FOX_BUG                       "\nError: bug"

#endif /* _FOX_ERROR_H_ */
```

File fox_cst.h

```
/*****************************************************************************/
/* FOX project / Reference implementation v1.2                              */
/* Pascal Junod <pascal@junod.info>                                          */
/*                                                                           */
/*****************************************************************************/

#ifndef _FOX_CST_H_
#define _FOX_CST_H_

#include "fox_portable.h"

/* Constants                                                                 */

#define FOX_NUMBER_ROUNDS    FOX_NUMBER_ROUNDS_MIN

#define FOX64_TABLE_SIGMA4_MU4_ID0            0x00
#define FOX64_TABLE_SIGMA4_MU4_ID1            0x01
#define FOX64_TABLE_SIGMA4_MU4_ID2            0x02
#define FOX64_TABLE_SIGMA4_MU4_ID3            0x03

#define FOX64_TABLE_SIGMA4_ID0                0x04
#define FOX64_TABLE_SIGMA4_ID1                0x05
#define FOX64_TABLE_SIGMA4_ID2                0x06
#define FOX64_TABLE_SIGMA4_ID3                0x07

```

```
#define FOX128_TABLE_SIGMA8_ID0              0x08
#define FOX128_TABLE_SIGMA8_ID1              0x09
#define FOX128_TABLE_SIGMA8_ID2              0x0A
#define FOX128_TABLE_SIGMA8_ID3              0x0B

#define FOX128_TABLE_SIGMA8_MU8_ID0          0x10
#define FOX128_TABLE_SIGMA8_MU8_ID1          0x11
#define FOX128_TABLE_SIGMA8_MU8_ID2          0x12
#define FOX128_TABLE_SIGMA8_MU8_ID3          0x13
#define FOX128_TABLE_SIGMA8_MU8_ID4          0x14
#define FOX128_TABLE_SIGMA8_MU8_ID5          0x15
#define FOX128_TABLE_SIGMA8_MU8_ID6          0x16
#define FOX128_TABLE_SIGMA8_MU8_ID7          0x17

/* FOX_IRRPOLY = x^8+x^7+x^6+x^5+x^4+x^3+1                               */

#define FOX_IRRPOLY                         0x1F9

/* Constants used in the key-schedule algorithm                          */

#define FOX_MKEYM2                           0x6A
#define FOX_MKEYM1                           0x76

#define FOX_LFSR_C                   0x006A0000UL
#define FOX_LFSR_FP                  0x0100001BUL


/* These are the first decimal of e-2                                    */

extern const uint8 FOX_KEY_PAD[32];

/* The three "small" S-boxes                                             */

extern const uint8 FOX_S1[16];
extern const uint8 FOX_S2[16];
extern const uint8 FOX_S3[16];

#endif /* _FOX_CST_H_                                                    */
```

File fox_cst.c

```
/*************************************************************************/
/* FOX project / Reference implementation v1.2                           */
/* Pascal Junod <pascal@junod.info>                                      */
/*                                                                       */
/*************************************************************************/

#include "fox_portable.h"
#include "fox_cst.h"

/* These are the first decimal of e-2                                    */

const uint8 FOX_KEY_PAD[32] = { 0xB7, 0xE1, 0x51, 0x62,
                                0x8A, 0xED, 0x2A, 0x6A,
                                0xBF, 0x71, 0x58, 0x80,
                                0x9C, 0xF4, 0xF3, 0xC7,
                                0x62, 0xE7, 0x16, 0x0F,
                                0x38, 0xB4, 0xDA, 0x56,
                                0xA7, 0x84, 0xD9, 0x04,
                                0x51, 0x90, 0xCF, 0xEF };

/* The three "small" S-boxes                                             */

const uint8 FOX_S1[16] = { 0x2, 0x5, 0x1, 0x9,
                           0xE, 0xA, 0xC, 0x8,
                           0x6, 0x4, 0x7, 0xF,
                           0xD, 0xB, 0x0, 0x3 };

const uint8 FOX_S2[16] = { 0xB, 0x4, 0x1, 0xF,
```

```
                             0x0, 0x3, 0xE, 0xD,
                             0xA, 0x8, 0x7, 0x5,
                             0xC, 0x2, 0x9, 0x6 };

const uint8 FOX_S3[16] = { 0xD, 0xA, 0xB, 0x1,
                           0x4, 0x3, 0x8, 0x9,
                           0x5, 0x7, 0x2, 0xC,
                           0xF, 0x0, 0x6, 0xE };


```

File fox_ctx.h

```
/******************************************************************************/
/* FOX project / Reference implementation v1.2                                */
/* Pascal Junod <pascal@junod.info>                                           */
/*                                                                            */
/******************************************************************************/

#ifndef _FOX_CTX_H_
#define _FOX_CTX_H_

#include "fox_portable.h"
#include "fox_cst.h"

/* Types                                                                      */

typedef uint8 FOX_mode;


typedef struct  {
    uint32 *exp_key;
    uint8   raw_key[32];
    uint8   key_length;
    uint8   rounds;
} FOX_key_;

typedef FOX_key_ *FOX_key;

typedef struct {
    uint32 *val;
    uint32  size_bytes;
    uint8   id;
} FOX_table_;

typedef FOX_table_ *FOX_table;

typedef struct {
    FOX_table sigma4_mu4_0;
    FOX_table sigma4_mu4_1;
    FOX_table sigma4_mu4_2;
    FOX_table sigma4_mu4_3;

    FOX_table sigma4_0;
    FOX_table sigma4_1;
    FOX_table sigma4_2;
    FOX_table sigma4_3;

} FOX64_ctx_;

typedef FOX64_ctx_ *FOX64_ctx;

typedef struct {
    FOX_table sigma8_mu8_0;
    FOX_table sigma8_mu8_1;
    FOX_table sigma8_mu8_2;
    FOX_table sigma8_mu8_3;
    FOX_table sigma8_mu8_4;
    FOX_table sigma8_mu8_5;
```

```
    FOX_table sigma8_mu8_6;
    FOX_table sigma8_mu8_7;

    FOX_table sigma8_0;
    FOX_table sigma8_1;
    FOX_table sigma8_2;
    FOX_table sigma8_3;

} FOX128_ctx_;

typedef FOX128_ctx_ *FOX128_ctx;

/* Exportable routines                                                       */

extern int FOX64_init_ctx (FOX64_ctx *);
extern void FOX64_clean_ctx (FOX64_ctx);

extern int FOX128_init_ctx (FOX128_ctx *);
extern void FOX128_clean_ctx (FOX128_ctx);


extern int FOX64_init_key (FOX_key *,
                           const FOX64_ctx,
                           const uint8 *,
                           const uint32,
                           const uint8);

extern void FOX64_clean_key (FOX_key);

extern int FOX128_init_key (FOX_key *,
                            const FOX128_ctx,
                            const uint8 *,
                            const uint32,
                            const uint8);

extern void FOX128_clean_key (FOX_key);

extern void FOX_io (uint32 *);
extern void FOX_or (uint32 *);


/* Internal routines                                                         */

int FOX_init_table (FOX_table *, const uint8);
void FOX_clean_table (FOX_table);

uint32 FOX_times_alpha (const uint32);
uint32 FOX_div_alpha  (const uint32);

uint32 FOX_eval_sbox (const uint32 x, const uint8 *s1,
                      const uint8 *s2, const uint8 *s3);


#endif /* _FOX_CTX_H_                                                        */
```

File fox_ctx.c

```
/*****************************************************************************/
/* FOX project / Reference implementation v1.2                               */
/* Pascal Junod <pascal@junod.info>                                          */
/*                                                                           */
/*****************************************************************************/

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <string.h>

#include "fox_portable.h"
```

```
#include "fox_error.h"
#include "fox_ctx.h"
#include "fox64.h"
#include "fox128.h"


void FOX_or (uint32 *data)
{
    uint32 l, r;

    assert (data != NULL);

    l = *data >> 16;
    r = *data & 0xFFFF;

    *data = (r << 16) | (l ^ r);
}

void FOX_io (uint32 *data)
{
    uint32 l, r;

    assert (data != NULL);

    l = *data >> 16;
    r = *data & 0xFFFF;

    *data = (( l ^ r) << 16) | l;
}

uint32 FOX_times_alpha (const uint32 input)
{

    if (input) {
        return  (input & 0x80) ? (input << 1) ^ FOX_IRRPOLY : input << 1;
    } else {
        return 0x00;
    }
}

uint32 FOX_div_alpha  (const uint32 input)
{

    if (input) {
         return (input & 0x01) ? (input ^ FOX_IRRPOLY) >> 1  : input >> 1;
    } else {
        return 0x00;
    }
}

int FOX64_init_ctx (FOX64_ctx *ptr)
{
    FOX64_ctx ctx;

    if ( (ctx = malloc (sizeof (FOX64_ctx_))) == NULL) {
        fprintf (stderr, FOX_ERROR_MEMORY_ALLOC);
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma4_mu4_0, FOX64_TABLE_SIGMA4_MU4_ID0)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma4_mu4_1, FOX64_TABLE_SIGMA4_MU4_ID1)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma4_mu4_2, FOX64_TABLE_SIGMA4_MU4_ID2)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma4_mu4_3, FOX64_TABLE_SIGMA4_MU4_ID3)) {
        goto error_label;
    }
```

```
    if (FOX_init_table (&ctx->sigma4_0, FOX64_TABLE_SIGMA4_ID0)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma4_1, FOX64_TABLE_SIGMA4_ID1)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma4_2, FOX64_TABLE_SIGMA4_ID2)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma4_3, FOX64_TABLE_SIGMA4_ID3)) {
        goto error_label;
    }

    *ptr = ctx;

    return 0;

 error_label:
    fprintf (stderr, FOX_ERROR_CONTEXT_INIT);
    FOX64_clean_ctx (ctx);

    return -1;
}


void FOX64_clean_ctx (FOX64_ctx ctx)
{
    if (ctx != NULL) {
        FOX_clean_table (ctx->sigma4_mu4_0);
        FOX_clean_table (ctx->sigma4_mu4_1);
        FOX_clean_table (ctx->sigma4_mu4_2);
        FOX_clean_table (ctx->sigma4_mu4_3);

        FOX_clean_table (ctx->sigma4_0);
        FOX_clean_table (ctx->sigma4_1);
        FOX_clean_table (ctx->sigma4_2);
        FOX_clean_table (ctx->sigma4_3);

        free (memset (ctx, 0x00, sizeof (FOX64_ctx_)));
    }
}

int FOX128_init_ctx (FOX128_ctx *ptr)
{
    FOX128_ctx ctx;

    if ( (ctx = malloc (sizeof (FOX128_ctx_))) == NULL) {
        fprintf (stderr, FOX_ERROR_MEMORY_ALLOC);
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma8_mu8_0, FOX128_TABLE_SIGMA8_MU8_ID0)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma8_mu8_1, FOX128_TABLE_SIGMA8_MU8_ID1)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma8_mu8_2, FOX128_TABLE_SIGMA8_MU8_ID2)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma8_mu8_3, FOX128_TABLE_SIGMA8_MU8_ID3)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma8_mu8_4, FOX128_TABLE_SIGMA8_MU8_ID4)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma8_mu8_5, FOX128_TABLE_SIGMA8_MU8_ID5)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma8_mu8_6, FOX128_TABLE_SIGMA8_MU8_ID6)) {
        goto error_label;
```

```
    }
    if (FOX_init_table (&ctx->sigma8_mu8_7, FOX128_TABLE_SIGMA8_MU8_ID7)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma8_0, FOX128_TABLE_SIGMA8_ID0)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma8_1, FOX128_TABLE_SIGMA8_ID1)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma8_2, FOX128_TABLE_SIGMA8_ID2)) {
        goto error_label;
    }
    if (FOX_init_table (&ctx->sigma8_3, FOX128_TABLE_SIGMA8_ID3)) {
        goto error_label;
    }

    *ptr = ctx;

    return 0;

 error_label:
    fprintf (stderr, FOX_ERROR_CONTEXT_INIT);
    FOX128_clean_ctx (ctx);

    return -1;
}

void FOX128_clean_ctx (FOX128_ctx ctx)
{
    if (ctx != NULL) {
        FOX_clean_table (ctx->sigma8_mu8_0);
        FOX_clean_table (ctx->sigma8_mu8_1);
        FOX_clean_table (ctx->sigma8_mu8_2);
        FOX_clean_table (ctx->sigma8_mu8_3);
        FOX_clean_table (ctx->sigma8_mu8_4);
        FOX_clean_table (ctx->sigma8_mu8_5);
        FOX_clean_table (ctx->sigma8_mu8_6);
        FOX_clean_table (ctx->sigma8_mu8_7);

        FOX_clean_table (ctx->sigma8_0);
        FOX_clean_table (ctx->sigma8_1);
        FOX_clean_table (ctx->sigma8_2);
        FOX_clean_table (ctx->sigma8_3);

        free (memset (ctx, 0x00, sizeof (FOX128_ctx_)));
    }
}

int FOX64_init_key (FOX_key *ptr,
                    const FOX64_ctx ctx,
                    const uint8 *bytes,
                    const uint32 length,
                    const uint8 rounds)
{
    FOX_key key;
    uint32 i, j;
    uint32 o;
    uint8 pkey[32], mkey[32], dkey[32];
    uint32 dkey32[8], temp32[8], reg32[8];
    uint32 b, ek;
    uint32 lfsr_state;
    uint8 lfsr[4];

    assert (ctx != NULL);

    assert (length <= 256);
    assert (length % 8 == 0);
    assert (rounds >= FOX64_NUMBER_ROUNDS_MIN);

```

```
    if ( (key = malloc (sizeof (FOX_key_))) == NULL) {
        fprintf (stderr, FOX_ERROR_MEMORY_ALLOC);
        goto error_label;
    }
    if ( (key->exp_key = malloc (sizeof (uint32) * 2 * rounds )) == NULL) {
        fprintf (stderr, FOX_ERROR_MEMORY_ALLOC);
        goto error_label;
    }

    memcpy (key->raw_key, bytes, (length >> 3));
    memcpy (pkey, bytes, (length >> 3));

    /* Size in bits                                                         */
    key->key_length = length;

    key->rounds = rounds;

    /* Computation of the state bit b and of ek                             */

    if ( (length == 128) || (length == 256) ) {
        b = 1;
    } else {
        b = 0;
    }

    if (length <= 128) {
        ek = 128;
    } else {
        ek = 256;
    }

    /* P-part                                                               */

    if (length < ek) {
        for (i = (length >> 3), j = 0; i < (ek >> 3); i++, j++) {
            pkey[i] = FOX_KEY_PAD[j];
        }
    }

    memcpy (mkey, pkey, (ek >> 3));

    /* M-part                                                               */

    if (length < ek) {
        mkey[0] ^= (FOX_MKEYM2 + FOX_MKEYM1);
        mkey[1] ^= (FOX_MKEYM1 + mkey[0]);
        for (i = 2; i < (ek >> 3); i++) {
            mkey[i] ^= (mkey[i - 2] + mkey[i - 1]);
        }
    }

    /* D-Part                                                               */

    /* Initialization of the LFSR                                           */
    lfsr_state = FOX_LFSR_C | ((uint32)rounds << 8) | (~rounds & 0xFF);

    /* We back-clock the LFSR once                                          */
    if (lfsr_state & 0x1) {
        lfsr_state ^= FOX_LFSR_FP;
    }
    lfsr_state >>= 1;

    for (i = 0; i < rounds; i++) {
        j = 0;
        while (j < (ek >> 3)) {
            if ( (j % 3) == 0 ) {
                /* We have to clock the LFSR                                */
                lfsr_state <<= 1;
                if (lfsr_state & 0x01000000) {
                    lfsr_state  ^= FOX_LFSR_FP;
```

```
                }
                /* Endianness issue here !                                    */
                U32TO8_BIG (lfsr, lfsr_state);
            }
            dkey[j] = mkey[j] ^ lfsr[(j % 3) + 1];
            j++;
        }

        for (j = 0; j < (ek >> 5); j++) {
            dkey32[j] = U8TO32_BIG (dkey + (j << 2));
        }

        /* NL-part : we feed the current DKEY to the NLx part                 */
        /* sigma4 - mu4 operation                                             */
        for (j = 0; j < (ek >> 5); j++) {
            o  = ctx->sigma4_mu4_0->val[(dkey32[j] & 0xFF000000) >> 24];
            o ^= ctx->sigma4_mu4_1->val[(dkey32[j] & 0x00FF0000) >> 16];
            o ^= ctx->sigma4_mu4_2->val[(dkey32[j] & 0x0000FF00) >>  8];
            o ^= ctx->sigma4_mu4_3->val[(dkey32[j] & 0x000000FF)];
            reg32[j] = o;
        }

        if (ek == 128) {
            /* mix64 operation                                                */
            temp32[0] = reg32[1] ^ reg32[2] ^ reg32[3];
            temp32[1] = reg32[0] ^ reg32[2] ^ reg32[3];
            temp32[2] = reg32[0] ^ reg32[1] ^ reg32[3];
            temp32[3] = reg32[0] ^ reg32[1] ^ reg32[2];
        } else {
            /* mix64h operation                                               */
            temp32[0] = reg32[2] ^ reg32[4] ^ reg32[6];
            temp32[1] = reg32[3] ^ reg32[5] ^ reg32[7];
            temp32[2] = reg32[0] ^ reg32[4] ^ reg32[6];
            temp32[3] = reg32[1] ^ reg32[5] ^ reg32[7];
            temp32[4] = reg32[0] ^ reg32[2] ^ reg32[6];
            temp32[5] = reg32[1] ^ reg32[3] ^ reg32[7];
            temp32[6] = reg32[0] ^ reg32[2] ^ reg32[4];
            temp32[7] = reg32[1] ^ reg32[3] ^ reg32[5];
        }
        /* Constant addition                                                  */
        /* Endianness issue here !                                            */
        for (j = 0; j < (ek >> 5); j++) {
            temp32[j] ^= U8TO32_BIG (FOX_KEY_PAD + (j << 2));
        }
        /* Conditional flip                                                   */
        if (b) {
            for (j = 0; j < (ek >> 5); j++) {
                temp32[j] = ~temp32[j];
            }
        }

        /* sigma4 operation                                                   */
        for (j = 0; j < (ek >> 5); j++) {
                o  = ctx->sigma4_0->val[(temp32[j] & 0xFF000000) >> 24];
                o ^= ctx->sigma4_1->val[(temp32[j] & 0x00FF0000) >> 16];
                o ^= ctx->sigma4_2->val[(temp32[j] & 0x0000FF00) >>  8];
                o ^= ctx->sigma4_3->val[(temp32[j] & 0x000000FF)];
                temp32[j] = o;
            }

        if (ek == 128) {
            /* Hashing                                                        */
            reg32[0] = temp32[0] ^ temp32[2];
            reg32[1] = temp32[1] ^ temp32[3];

            /* Encryption phase                                               */
            FOX_lmor64 (reg32, dkey32, ctx);
            FOX_lmid64 (reg32, dkey32 + 2, ctx);
            *(key->exp_key + 2*i) = reg32[0];
            *(key->exp_key + 2*i + 1) = reg32[1];
```

```
        } else {
            /* Hashing                                                     */
            reg32[0] = temp32[0] ^ temp32[1];
            reg32[1] = temp32[2] ^ temp32[3];
            reg32[2] = temp32[4] ^ temp32[5];
            reg32[3] = temp32[6] ^ temp32[7];

            temp32[0] = reg32[0] ^ reg32[2];
            temp32[1] = reg32[1] ^ reg32[3];

            /* Encryption phase                                            */
            FOX_lmor64 (temp32, dkey32, ctx);
            FOX_lmor64 (temp32, dkey32 + 2, ctx);
            FOX_lmor64 (temp32, dkey32 + 4, ctx);
            FOX_lmid64 (temp32, dkey32 + 6, ctx);
            *(key->exp_key + 2*i) = temp32[0];
            *(key->exp_key + 2*i + 1) = temp32[1];
        }
    }

    *ptr = key;

    return 0;

 error_label:
    FOX64_clean_key (key);

    return -1;
}

void FOX64_clean_key (FOX_key k)
{
    if (k != NULL) {
        if (k->exp_key != NULL) {
            free (memset (k->exp_key, 0x00, k->rounds * 2 * sizeof (uint32)));
        }
        free (memset (k, 0x00, sizeof (FOX_key_)));
    }
}

int FOX128_init_key (FOX_key *ptr,
                     const FOX128_ctx ctx,
                     const uint8 *bytes,
                     const uint32 length,
                     const uint8 rounds)
{
    FOX_key key;
    uint32 i, j;
    uint32 o[2];
    uint8 dkey[32], pkey[32], mkey[32];
    uint32 dkey32[8], temp32[8], reg32[8];
    uint32 b, ek;
    uint32 lfsr_state;
    uint8 lfsr[4];

    assert (length <= 256);
    assert (length % 8 == 0);
    assert (rounds >= FOX64_NUMBER_ROUNDS_MIN);

    if ( (key = malloc (sizeof (FOX_key_))) == NULL) {
        fprintf (stderr, FOX_ERROR_MEMORY_ALLOC);
        goto error_label;
    }
    if ( (key->exp_key = malloc (sizeof (uint32) * 4 * rounds )) == NULL) {
        fprintf (stderr, FOX_ERROR_MEMORY_ALLOC);
        goto error_label;
    }

    memcpy (key->raw_key, bytes, (length >> 3));
    memcpy (pkey, bytes, (length >> 3));
```

```
    /* Size in bits                                                          */

    key->key_length = length;
    key->rounds = rounds;

    /* Computation of the state bit b and of ek                              */

    if ( length == 256 ) {
        b = 1;
    } else {
        b = 0;
    }
    ek = 256;


    /* P-part                                                                */

    if (length < ek) {
        for (i = (length >> 3), j = 0; i < 32; i++, j++) {
            pkey[i] = FOX_KEY_PAD[j];
        }
    }

    memcpy (mkey, pkey, (ek >> 3));

    /* M-part                                                                */

    if (length < ek) {
        mkey[0] ^= (FOX_MKEYM2 + FOX_MKEYM1);
        mkey[1] ^= (FOX_MKEYM1 + mkey[0]);
        for (i = 2; i < (ek >> 3); i++) {
            mkey[i] ^= (mkey[i - 2] + mkey[i - 1]);
        }
    }

    /* D-Part                                                                */

    /* Initialization of the LFSR                                            */
    lfsr_state = FOX_LFSR_C | ((uint32)rounds << 8) | (~rounds & 0xFF);

    /* We back-clock the LFSR once                                           */
    if (lfsr_state & 0x1) {
        lfsr_state ^= FOX_LFSR_FP;
    }
    lfsr_state >>= 1;

    for (i = 0; i < rounds; i++) {
        j = 0;
        while (j < (ek >> 3)) {
            if ( (j % 3) == 0 ) {
                /* We have to clock the LFSR                                 */
                lfsr_state <<= 1;
                if (lfsr_state & 0x01000000) {
                    lfsr_state  ^= FOX_LFSR_FP;
                }
                /* Endianness issue here !                                   */
                U32TO8_BIG (lfsr, lfsr_state);
            }
            dkey[j] = mkey[j] ^ lfsr[(j % 3) + 1];
            j++;
        }

        for (j = 0; j < 8; j++) {
            dkey32[j] = U8TO32_BIG (dkey + (j << 2));
        }

        /* NL Part                                                           */

        /* sigma8 - mu8 operation                                            */
```

```
        for (j = 0; j < 4; j++) {
            o[0]  = ctx->sigma8_mu8_0->val[ (dkey32[2*j] & 0xFF000000) >> 23];
            o[1]  = ctx->sigma8_mu8_0->val[((dkey32[2*j] & 0xFF000000) >> 23) + 1];
            o[0] ^= ctx->sigma8_mu8_1->val[ (dkey32[2*j] & 0x00FF0000) >> 15];
            o[1] ^= ctx->sigma8_mu8_1->val[((dkey32[2*j] & 0x00FF0000) >> 15) + 1];
            o[0] ^= ctx->sigma8_mu8_2->val[ (dkey32[2*j] & 0x0000FF00) >> 7];
            o[1] ^= ctx->sigma8_mu8_2->val[((dkey32[2*j] & 0x0000FF00) >> 7) + 1];
            o[0] ^= ctx->sigma8_mu8_3->val[ (dkey32[2*j] & 0x000000FF) << 1];
            o[1] ^= ctx->sigma8_mu8_3->val[((dkey32[2*j] & 0x000000FF) << 1)+ 1];

            o[0] ^= ctx->sigma8_mu8_4->val[(dkey32[2*j+1] & 0xFF000000) >> 23];
            o[1] ^= ctx->sigma8_mu8_4->val[((dkey32[2*j+1] & 0xFF000000) >> 23) + 1];
            o[0] ^= ctx->sigma8_mu8_5->val[(dkey32[2*j+1] & 0x00FF0000) >> 15];
            o[1] ^= ctx->sigma8_mu8_5->val[((dkey32[2*j+1] & 0x00FF0000) >> 15) + 1];
            o[0] ^= ctx->sigma8_mu8_6->val[(dkey32[2*j+1] & 0x0000FF00) >> 7];
            o[1] ^= ctx->sigma8_mu8_6->val[((dkey32[2*j+1] & 0x0000FF00) >> 7) + 1];
            o[0] ^= ctx->sigma8_mu8_7->val[(dkey32[2*j+1] & 0x000000FF) << 1];
            o[1] ^= ctx->sigma8_mu8_7->val[((dkey32[2*j+1] & 0x000000FF) << 1) + 1];

            reg32[2*j] = o[0];
            reg32[2*j + 1] = o[1];
        }

        /* mix128 operation                                                   */

        temp32[0] = reg32[2] ^ reg32[4] ^ reg32[6];
        temp32[1] = reg32[3] ^ reg32[5] ^ reg32[7];
        temp32[2] = reg32[0] ^ reg32[4] ^ reg32[6];
        temp32[3] = reg32[1] ^ reg32[5] ^ reg32[7];
        temp32[4] = reg32[0] ^ reg32[2] ^ reg32[6];
        temp32[5] = reg32[1] ^ reg32[3] ^ reg32[7];
        temp32[6] = reg32[0] ^ reg32[2] ^ reg32[4];
        temp32[7] = reg32[1] ^ reg32[3] ^ reg32[5];

        /* Constant addition                                                  */
        /* Endianness issue here !                                            */
        for (j = 0; j < 8; j++) {
            temp32[j] ^= U8TO32_BIG (FOX_KEY_PAD + (j << 2));
        }
        /* Conditional flip                                                   */
        if (b) {
            for (j = 0; j < 8; j++) {
                temp32[j] = ~temp32[j];
            }
        }

        /* sigma8 operation                                                   */
        for (j = 0; j < 4; j++) {
            o[0]  = ctx->sigma8_0->val[(temp32[2*j] & 0xFF000000) >> 24];
            o[0] ^= ctx->sigma8_1->val[(temp32[2*j] & 0x00FF0000) >> 16];
            o[0] ^= ctx->sigma8_2->val[(temp32[2*j] & 0x0000FF00) >> 8];
            o[0] ^= ctx->sigma8_3->val[(temp32[2*j] & 0x000000FF)];

            o[1]  = ctx->sigma8_0->val[(temp32[2*j+1] & 0xFF000000) >> 24];
            o[1] ^= ctx->sigma8_1->val[(temp32[2*j+1] & 0x00FF0000) >> 16];
            o[1] ^= ctx->sigma8_2->val[(temp32[2*j+1] & 0x0000FF00) >> 8];
            o[1] ^= ctx->sigma8_3->val[(temp32[2*j+1] & 0x000000FF)];

            temp32[2*j] = o[0];
            temp32[2*j + 1] = o[1];
        }

        reg32[0] = temp32[0] ^ temp32[4];
        reg32[1] = temp32[1] ^ temp32[5];
        reg32[2] = temp32[2] ^ temp32[6];
        reg32[3] = temp32[3] ^ temp32[7];

        /* Encryption phase                                                   */
        FOX_elmor128 (reg32, dkey32, ctx);
        FOX_elmid128 (reg32, dkey32 + 4, ctx);
```

```
        *(key->exp_key + 4*i)     = reg32[0];
        *(key->exp_key + 4*i + 1) = reg32[1];
        *(key->exp_key + 4*i + 2) = reg32[2];
        *(key->exp_key + 4*i + 3) = reg32[3];
    }

    *ptr = key;

    return 0;

 error_label:
    FOX128_clean_key (key);

    return -1;
}

void FOX128_clean_key (FOX_key k)
{
    if (k != NULL) {
        if (k->exp_key != NULL) {
            free (memset (k->exp_key, 0x00, k->rounds *
                          4 * sizeof (uint32)));
        }
        free (memset (k, 0x00, sizeof (FOX_key_)));
    }
}

int FOX_init_table (FOX_table *ptr, const uint8 id)
{
    uint32 i, size, tmp;
    FOX_table table;

    if ( (table = malloc (sizeof (FOX_table_))) == NULL) {
        fprintf (stderr, FOX_ERROR_MEMORY_ALLOC);
        goto error_label;
    }

    if (id >= 0x8) {
        size = 2;
    } else {
        size = 1;
    }

    if ( (table->val = malloc (256 * sizeof(uint32) * size)) == NULL) {
        fprintf (stderr, FOX_ERROR_MEMORY_ALLOC);
        goto error_label;
    }
    table->id = id;
    table->size_bytes = 256 * sizeof(uint32) * size;

    switch (id) {

        case FOX64_TABLE_SIGMA4_MU4_ID0:
            for (i = 0; i < 256; i++) {
                tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
                table->val[i]  = tmp << 24;
                table->val[i] |= tmp << 16;
                table->val[i] |= (FOX_div_alpha(tmp) ^ tmp) << 8;
                table->val[i] |= FOX_times_alpha (tmp);
            }
            break;

        case FOX64_TABLE_SIGMA4_MU4_ID1:
            for (i = 0; i < 256; i++) {
                tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
                table->val[i]  = tmp << 24;
                table->val[i] |= (FOX_div_alpha(tmp) ^ tmp) << 16;
                table->val[i] |= FOX_times_alpha (tmp) << 8;
                table->val[i] |= tmp;
```

```
            }
            break;

        case FOX64_TABLE_SIGMA4_MU4_ID2:
            for (i = 0; i < 256; i++) {
                tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
                table->val[i]  = tmp << 24;
                table->val[i] |= FOX_times_alpha (tmp) << 16;
                table->val[i] |= tmp << 8;
                table->val[i] |= (FOX_div_alpha(tmp) ^ tmp);
            }
            break;

        case FOX64_TABLE_SIGMA4_MU4_ID3:
            for (i = 0; i < 256; i++) {
                tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
                table->val[i]  = FOX_times_alpha (tmp) << 24;
                table->val[i] |= tmp << 16;
                table->val[i] |= tmp << 8;
                table->val[i] |= tmp;
            }
            break;

        case FOX128_TABLE_SIGMA8_MU8_ID0:
            for (i = 0; i < 256; i++) {
                tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
                table->val[2*i]  = tmp << 24;
                table->val[2*i] |= tmp << 16;
                table->val[2*i] |= (FOX_times_alpha (tmp) ^ tmp) << 8;
                table->val[2*i] |= (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp)));
                table->val[2*i+1]  = FOX_times_alpha (tmp) << 24;
                table->val[2*i+1] |= FOX_times_alpha (FOX_times_alpha (tmp)) << 16;
                table->val[2*i+1] |= FOX_div_alpha (tmp) << 8;
                table->val[2*i+1] |= FOX_div_alpha (FOX_div_alpha (tmp));
            }
            break;

        case FOX128_TABLE_SIGMA8_MU8_ID1:
            for (i = 0; i < 256; i++) {
                tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
                table->val[2*i]  = tmp << 24;
                table->val[2*i] |= (FOX_times_alpha (tmp) ^ tmp) << 16;
                table->val[2*i] |= (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp))) << 8;
                table->val[2*i] |= FOX_times_alpha (tmp);
                table->val[2*i+1]  = FOX_times_alpha (FOX_times_alpha (tmp)) << 24;
                table->val[2*i+1] |= FOX_div_alpha (tmp) << 16;
                table->val[2*i+1] |= FOX_div_alpha (FOX_div_alpha (tmp)) << 8;
                table->val[2*i+1] |= tmp;
            }
            break;

        case FOX128_TABLE_SIGMA8_MU8_ID2:
            for (i = 0; i < 256; i++) {
                tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
                table->val[2*i]  = tmp << 24;
                table->val[2*i] |= (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp))) << 16;
                table->val[2*i] |= FOX_times_alpha (tmp) << 8;
                table->val[2*i] |= FOX_times_alpha (FOX_times_alpha (tmp));
                table->val[2*i+1]  = FOX_div_alpha (tmp) << 24;
                table->val[2*i+1] |= FOX_div_alpha (FOX_div_alpha (tmp)) << 16;
                table->val[2*i+1] |= tmp << 8;
                table->val[2*i+1] |= (FOX_times_alpha (tmp) ^ tmp);
            }
            break;

        case FOX128_TABLE_SIGMA8_MU8_ID3:
            for (i = 0; i < 256; i++) {
                tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
                table->val[2*i]  = tmp << 24;
                table->val[2*i] |= FOX_times_alpha (tmp) << 16;
```

```
                table->val[2*i] |= FOX_times_alpha (FOX_times_alpha (tmp)) << 8;
                table->val[2*i] |= FOX_div_alpha (tmp);
                table->val[2*i+1]  = FOX_div_alpha (FOX_div_alpha (tmp)) << 24;
                table->val[2*i+1] |= tmp << 16;
                table->val[2*i+1] |= (FOX_times_alpha (tmp) ^ tmp) << 8;
                table->val[2*i+1] |= (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp)));
            }
            break;

        case FOX128_TABLE_SIGMA8_MU8_ID4:
            for (i = 0; i < 256; i++) {
                tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
                table->val[2*i]  = tmp << 24;
                table->val[2*i] |= FOX_times_alpha (FOX_times_alpha (tmp)) << 16;
                table->val[2*i] |= FOX_div_alpha (tmp) << 8;
                table->val[2*i] |= FOX_div_alpha (FOX_div_alpha (tmp));
                table->val[2*i+1] = tmp << 24;
                table->val[2*i+1] |= (FOX_times_alpha (tmp) ^ tmp) << 16;
                table->val[2*i+1] |= (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp))) << 8;
                table->val[2*i+1] |= FOX_times_alpha (tmp);
            }
            break;

        case FOX128_TABLE_SIGMA8_MU8_ID5:
            for (i = 0; i < 256; i++) {
                tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
                table->val[2*i]  = tmp << 24;
                table->val[2*i] |= FOX_div_alpha (tmp) << 16;
                table->val[2*i] |= FOX_div_alpha (FOX_div_alpha (tmp)) << 8;
                table->val[2*i] |= tmp;
                table->val[2*i+1]  = (FOX_times_alpha (tmp) ^ tmp) << 24;
                table->val[2*i+1] |= (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp))) << 16;
                table->val[2*i+1] |= FOX_times_alpha (tmp) << 8;
                table->val[2*i+1] |= FOX_times_alpha (FOX_times_alpha (tmp));
            }
            break;

        case FOX128_TABLE_SIGMA8_MU8_ID6:
            for (i = 0; i < 256; i++) {
                tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
                table->val[2*i]  = tmp << 24;
                table->val[2*i] |= FOX_div_alpha (FOX_div_alpha (tmp)) << 16;
                table->val[2*i] |= tmp << 8;
                table->val[2*i] |= (FOX_times_alpha (tmp) ^ tmp);
                table->val[2*i+1]  = (FOX_div_alpha (tmp ^ FOX_div_alpha (tmp))) << 24;
                table->val[2*i+1] |= FOX_times_alpha (tmp) << 16;
                table->val[2*i+1] |= FOX_times_alpha (FOX_times_alpha (tmp)) << 8;
                table->val[2*i+1] |= FOX_div_alpha (tmp);
            }
            break;

        case FOX128_TABLE_SIGMA8_MU8_ID7:
            for (i = 0; i < 256; i++) {
                tmp = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
                table->val[2*i]  = (FOX_times_alpha (tmp) ^ tmp) << 24;
                table->val[2*i] |= tmp << 16;
                table->val[2*i] |= tmp << 8;
                table->val[2*i] |= tmp;
                table->val[2*i+1]  = tmp << 24;
                table->val[2*i+1] |= tmp << 16;
                table->val[2*i+1] |= tmp << 8;
                table->val[2*i+1] |= tmp;
            }
            break;

        case FOX64_TABLE_SIGMA4_ID0:
        case FOX128_TABLE_SIGMA8_ID0:
            for (i = 0; i < 256; i++) {
                table->val[i] = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3) << 24;
            }
```

```
            break;

        case FOX64_TABLE_SIGMA4_ID1:
        case FOX128_TABLE_SIGMA8_ID1:
            for (i = 0; i < 256; i++) {
                table->val[i] = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3) << 16;
            }
            break;

        case FOX64_TABLE_SIGMA4_ID2:
        case FOX128_TABLE_SIGMA8_ID2:
            for (i = 0; i < 256; i++) {
                table->val[i] = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3) << 8;
            }
            break;

        case FOX64_TABLE_SIGMA4_ID3:
        case FOX128_TABLE_SIGMA8_ID3:
            for (i = 0; i < 256; i++) {
                table->val[i] = FOX_eval_sbox (i, FOX_S1, FOX_S2, FOX_S3);
            }
            break;

        default:
            fprintf (stderr, FOX_ERROR_UNKNOWN_TABLE_ID);
            goto error_label;
    }

    *ptr = table;

    return 0;

 error_label:
    fprintf (stderr, FOX_ERROR_TABLE_INIT);
    FOX_clean_table (table);

    return -1;
}

void FOX_clean_table (FOX_table table)
{
    if (table != NULL) {
        if (table->val != NULL) {
            free (memset (table->val, 0x00, table->size_bytes));
        }
        free (memset (table, 0x00, sizeof (FOX_table_)));
    }
}

uint32 FOX_eval_sbox (const uint32 x, const uint8 *s1,
                      const uint8 *s2, const uint8 *s3)
{
    uint8 l, r, ll, lr, state;

    assert ( (x <= 0xFF) && (s1 != NULL) && (s2 != NULL) && (s3 != NULL) );

    l = (x & 0xF0) >> 4;
    r = (x & 0x0F);

    /* Round 1                                                                */

    state = s1[l ^ r];
    l ^= state;
    r ^= state;

    ll = (l & 0xC) >> 2;
    lr = (l & 0x3);

    l = (lr << 2) | (ll ^ lr);

```

```
    /* Stage 2                                                           */

    state = s2[l ^ r];
    l ^= state;
    r ^= state;

    ll = (l & 0xC) >> 2;
    lr = (l & 0x3);

    l = (lr << 2) | (ll ^ lr);

    /* Stage 3 (without orthomorphism)                                   */

    state = s3[l ^ r];
    l ^= state;
    r ^= state;

    /* Saving of the value */

    return  (uint32)((l << 4) | r);
}
```

File fox64.h

```
/******************************************************************************/
/* FOX project / Reference implementation v1.2                               */
/* Pascal Junod <pascal@junod.info>                                          */
/*                                                                           */
/******************************************************************************/

#ifndef _FOX64_H_
#define _FOX64_H_

#include "fox_portable.h"
#include "fox_ctx.h"

#define FOX64_MODE_ENCRYPT         0x0
#define FOX64_MODE_DECRYPT         0x1

#define FOX64_NUMBER_ROUNDS_MIN         12
#define FOX64_NUMBER_ROUNDS_GENERIC     16

#define FOX64_encrypt(p, k, ctx)  FOX64_process((p), (k), (ctx), FOX64_MODE_ENCRYPT)
#define FOX64_decrypt(c, k, ctx)  FOX64_process((c), (k), (ctx), FOX64_MODE_DECRYPT)

extern int FOX64_process (uint32 *, const FOX_key, const FOX64_ctx, const FOX_mode);

void FOX_lmor64 (uint32 *, const uint32 *, const FOX64_ctx);
void FOX_lmid64 (uint32 *, const uint32 *, const FOX64_ctx);
void FOX_lmio64 (uint32 *, const uint32 *, const FOX64_ctx);
void FOX_f32 (uint32 *, const uint32 *, const FOX64_ctx);

#endif /* _FOX64_H_                                                          */
```

File fox64.c

```
/******************************************************************************/
/* FOX project / Reference implementation v1.2                               */
/* Pascal Junod <pascal@junod.info>                                          */
/*                                                                           */
/******************************************************************************/

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

#include "fox_portable.h"
#include "fox_error.h"
```

```
#include "fox_ctx.h"
#include "fox64.h"

int FOX64_process (uint32 *data,
                   const FOX_key k,
                   const FOX64_ctx ctx,
                   const FOX_mode mode)
{
    int r;
    uint32 input[2];

    assert (data != NULL);
    assert (k != NULL);
    assert (ctx != NULL);

    assert (k->rounds >= FOX64_NUMBER_ROUNDS_MIN);

    input[0] = data[0];
    input[1] = data[1];

    switch (mode) {

        case FOX64_MODE_ENCRYPT:
            for (r = 0; r < k->rounds - 1; r++) {
                FOX_lmor64 (input, k->exp_key + (r * 2), ctx);
            }
            FOX_lmid64 (input, k->exp_key + (k->rounds-1) * 2, ctx);
            break;

        case FOX64_MODE_DECRYPT:
            for (r = k->rounds - 1; r > 0; r--) {
                FOX_lmio64 (input, k->exp_key + (r * 2), ctx);
            }
            FOX_lmid64 (input, k->exp_key, ctx);
            break;

        default:
            fprintf (stderr, FOX_ERROR_UNKNOWN_MODE);
            return -1;
    }

    data[0] = input[0];
    data[1] = input[1];

    return 0;
}

void FOX_lmor64 (uint32 *data,
                 const uint32 *key,
                 const FOX64_ctx ctx)
{
    uint32 tmp[2], f;

    tmp[0] = data[0];
    tmp[1] = data[1];

    f = tmp[0] ^ tmp[1];
    FOX_f32 (&f, key, ctx);
    tmp[0] ^= f;
    tmp[1] ^= f;
    FOX_or (tmp);

    data[0] = tmp[0];
    data[1] = tmp[1];
}

void FOX_lmid64 (uint32 *data,
                 const uint32 *key,
                 const FOX64_ctx ctx)
{
```

```
    uint32 tmp[2], f;

    tmp[0] = data[0];
    tmp[1] = data[1];

    f = tmp[0] ^ tmp[1];
    FOX_f32 (&f, key, ctx);
    tmp[0] ^= f;
    tmp[1] ^= f;

    data[0] = tmp[0];
    data[1] = tmp[1];
}

void FOX_lmio64 (uint32 *data,
                 const uint32 *key,
                 const FOX64_ctx ctx)
{
    uint32 tmp[2], f;

    tmp[0] = data[0];
    tmp[1] = data[1];

    f = tmp[0] ^ tmp[1];
    FOX_f32 (&f, key, ctx);
    tmp[0] ^= f;
    tmp[1] ^= f;
    FOX_io (tmp);

    data[0] = tmp[0];
    data[1] = tmp[1];
}

void FOX_f32 (uint32 *data,
              const uint32 *key,
              const FOX64_ctx ctx)
{
    uint32 i, o;

    i = *data;

    i ^= key[0];

    o  = ctx->sigma4_mu4_0->val[(i & 0xFF000000) >> 24];
    o ^= ctx->sigma4_mu4_1->val[(i & 0x00FF0000) >> 16];
    o ^= ctx->sigma4_mu4_2->val[(i & 0x0000FF00) >>  8];
    o ^= ctx->sigma4_mu4_3->val[(i & 0x000000FF)];

    o ^= key[1];

    i  = ctx->sigma4_0->val[(o & 0xFF000000) >> 24];
    i ^= ctx->sigma4_1->val[(o & 0x00FF0000) >> 16];
    i ^= ctx->sigma4_2->val[(o & 0x0000FF00) >>  8];
    i ^= ctx->sigma4_3->val[(o & 0x000000FF)];

    *data = i ^ key[0];
}
```

File fox128.h

```
/*****************************************************************************/
/* FOX project / Reference implementation v1.2                               */
/* Pascal Junod <pascal@junod.info>                                          */
/*                                                                           */
/*****************************************************************************/

#ifndef _FOX128_H_
#define _FOX128_H_

```

```
#include "fox_portable.h"
#include "fox_ctx.h"

#define FOX128_MODE_ENCRYPT             0x0
#define FOX128_MODE_DECRYPT             0x1

#define FOX128_NUMBER_ROUNDS_MIN         12
#define FOX128_NUMBER_ROUNDS_GENERIC     16

#define FOX128_encrypt(p, k, ctx)  FOX128_process((p), (k), (ctx), FOX128_MODE_ENCRYPT)
#define FOX128_decrypt(c, k, ctx)  FOX128_process((c), (k), (ctx), FOX128_MODE_DECRYPT)

extern int FOX128_process (uint32 *, const FOX_key, const FOX128_ctx, const FOX_mode);

void FOX_elmor128 (uint32 *, const uint32 *, const FOX128_ctx);
void FOX_elmid128 (uint32 *, const uint32 *, const FOX128_ctx);
void FOX_elmio128 (uint32 *, const uint32 *, const FOX128_ctx);

void FOX_f64 (uint32 *, const uint32 *, const FOX128_ctx);

#endif /* _FOX128_H_                                                         */
```

File fox128.c

```
/*****************************************************************************/
/* FOX project / Reference implementation v1.2                              */
/* Pascal Junod <pascal@junod.info>                                          */
/*                                                                           */
/*****************************************************************************/

#include <assert.h>
#include <stdlib.h>
#include <stdio.h>

#include "fox_portable.h"
#include "fox_error.h"
#include "fox_ctx.h"
#include "fox128.h"

int FOX128_process (uint32 *data,
                    const FOX_key k,
                    const FOX128_ctx ctx,
                    const FOX_mode mode)
{
    int r;
    uint32 input[4];

    assert (data != NULL);
    assert (k != NULL);
    assert (ctx != NULL);

    assert (k->rounds >= FOX128_NUMBER_ROUNDS_MIN);

    input[0] = data[0];
    input[1] = data[1];
    input[2] = data[2];
    input[3] = data[3];

    switch (mode) {

        case FOX128_MODE_ENCRYPT:
            for (r = 0; r < k->rounds - 1; r++) {
                FOX_elmor128 (input, k->exp_key + (r * 4), ctx);
            }
            FOX_elmid128 (input, k->exp_key + (k->rounds - 1) * 4, ctx);
                break;

        case FOX128_MODE_DECRYPT:
            for (r = k->rounds - 1; r > 0; r--) {
```

```
                FOX_elmio128 (input, k->exp_key + (r * 4), ctx);
            }
            FOX_elmid128 (input, k->exp_key, ctx);
            break;

        default:
            fprintf (stderr, FOX_ERROR_UNKNOWN_MODE);
            return -1;
    }

    data[0] = input[0];
    data[1] = input[1];
    data[2] = input[2];
    data[3] = input[3];

    return 0;
}

void FOX_elmor128 (uint32 *data,
                   const uint32 *key,
                   const FOX128_ctx ctx)
{
    uint32 tmp[4], f[2];

    tmp[0] = data[0];
    tmp[1] = data[1];
    tmp[2] = data[2];
    tmp[3] = data[3];

    f[0] = tmp[0] ^ tmp[1];
    f[1] = tmp[2] ^ tmp[3];

    FOX_f64 (f, key, ctx);

    tmp[0] ^= f[0];
    tmp[1] ^= f[0];
    tmp[2] ^= f[1];
    tmp[3] ^= f[1];

    FOX_or (tmp);
    FOX_or (tmp + 2);

    data[0] = tmp[0];
    data[1] = tmp[1];
    data[2] = tmp[2];
    data[3] = tmp[3];
}

void FOX_elmid128 (uint32 *data,
                   const uint32 *key,
                   const FOX128_ctx ctx)
{
    uint32 tmp[4], f[2];

    tmp[0] = data[0];
    tmp[1] = data[1];
    tmp[2] = data[2];
    tmp[3] = data[3];

    f[0] = tmp[0] ^ tmp[1];
    f[1] = tmp[2] ^ tmp[3];

    FOX_f64 (f, key, ctx);

    tmp[0] ^= f[0];
    tmp[1] ^= f[0];
    tmp[2] ^= f[1];
    tmp[3] ^= f[1];

    data[0] = tmp[0];
```

```
    data[1] = tmp[1];
    data[2] = tmp[2];
    data[3] = tmp[3];
}

void FOX_elmio128 (uint32 *data,
                   const uint32 *key,
                   const FOX128_ctx ctx)
{
    uint32 tmp[4], f[2];

    tmp[0] = data[0];
    tmp[1] = data[1];
    tmp[2] = data[2];
    tmp[3] = data[3];

    f[0] = tmp[0] ^ tmp[1];
    f[1] = tmp[2] ^ tmp[3];

    FOX_f64 (f, key, ctx);

    tmp[0] ^= f[0];
    tmp[1] ^= f[0];
    tmp[2] ^= f[1];
    tmp[3] ^= f[1];

    FOX_io (tmp);
    FOX_io (tmp + 2);

    data[0] = tmp[0];
    data[1] = tmp[1];
    data[2] = tmp[2];
    data[3] = tmp[3];
}

void FOX_f64 (uint32 *data,
              const uint32 *key,
              const FOX128_ctx ctx)
{
    uint32 i[2], o[2];

    i[0]  = data[0];
    i[1]  = data[1];

    i[0] ^= key[0];
    i[1] ^= key[1];

    o[0]  = ctx->sigma8_mu8_0->val[(i[0] & 0xFF000000) >> 23];
    o[1]  = ctx->sigma8_mu8_0->val[((i[0] & 0xFF000000) >> 23) + 1];
    o[0] ^= ctx->sigma8_mu8_1->val[(i[0] & 0x00FF0000) >> 15];
    o[1] ^= ctx->sigma8_mu8_1->val[((i[0] & 0x00FF0000) >> 15) + 1];
    o[0] ^= ctx->sigma8_mu8_2->val[(i[0] & 0x0000FF00) >> 7];
    o[1] ^= ctx->sigma8_mu8_2->val[((i[0] & 0x0000FF00) >> 7) + 1];
    o[0] ^= ctx->sigma8_mu8_3->val[(i[0] & 0x000000FF) << 1];
    o[1] ^= ctx->sigma8_mu8_3->val[((i[0] & 0x000000FF) << 1)+ 1];

    o[0] ^= ctx->sigma8_mu8_4->val[(i[1] & 0xFF000000) >> 23];
    o[1] ^= ctx->sigma8_mu8_4->val[((i[1] & 0xFF000000) >> 23) + 1];
    o[0] ^= ctx->sigma8_mu8_5->val[(i[1] & 0x00FF0000) >> 15];
    o[1] ^= ctx->sigma8_mu8_5->val[((i[1] & 0x00FF0000) >> 15) + 1];
    o[0] ^= ctx->sigma8_mu8_6->val[(i[1] & 0x0000FF00) >> 7];
    o[1] ^= ctx->sigma8_mu8_6->val[((i[1] & 0x0000FF00) >> 7) + 1];
    o[0] ^= ctx->sigma8_mu8_7->val[(i[1] & 0x000000FF) << 1];
    o[1] ^= ctx->sigma8_mu8_7->val[((i[1] & 0x000000FF) << 1) + 1];

    o[0] ^= key[2];
    o[1] ^= key[3];

    i[0]  = ctx->sigma8_0->val[(o[0] & 0xFF000000) >> 24];
    i[0] ^= ctx->sigma8_1->val[(o[0] & 0x00FF0000) >> 16];
```

```
    i[0] ^= ctx->sigma8_2->val[(o[0] & 0x0000FF00) >> 8];
    i[0] ^= ctx->sigma8_3->val[(o[0] & 0x000000FF)];

    i[1]  = ctx->sigma8_0->val[(o[1] & 0xFF000000) >> 24];
    i[1] ^= ctx->sigma8_1->val[(o[1] & 0x00FF0000) >> 16];
    i[1] ^= ctx->sigma8_2->val[(o[1] & 0x0000FF00) >> 8];
    i[1] ^= ctx->sigma8_3->val[(o[1] & 0x000000FF)];

    data[0] = i[0] ^ key[0];
    data[1] = i[1] ^ key[1];
}
```

File fox_util.h

```
/******************************************************************************/
/* FOX project / Reference implementation v1.2                               */
/* Pascal Junod <pascal@junod.info>                                          */
/*                                                                           */
/******************************************************************************/

#ifndef _FOX_UTIL_H_
#define _FOX_UTIL_H_

#include "fox_portable.h"

int fox64_64_16_test (const uint8 *p, const uint8 *k);
int fox64_128_16_test (const uint8 *p, const uint8 *k);
int fox64_192_16_test (const uint8 *p, const uint8 *k);
int fox64_256_16_test (const uint8 *p, const uint8 *k);

int fox128_64_16_test (const uint8 *p, const uint8 *k);
int fox128_128_16_test (const uint8 *p, const uint8 *k);
int fox128_192_16_test (const uint8 *p, const uint8 *k);
int fox128_256_16_test (const uint8 *p, const uint8 *k);

#endif /* _FOX_UTIL_H_                                                       */
```

File fox_util.c

```
/******************************************************************************/
/* FOX project / Reference implementation v1.2                               */
/* Pascal Junod <pascal@junod.info>                                          */
/*                                                                           */
/******************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "fox_portable.h"
#include "fox_error.h"
#include "fox64.h"
#include "fox128.h"
#include "fox_ctx.h"
#include "fox_util.h"

const uint8 p64[8]   = {0x01, 0x23, 0x45, 0x67,
                        0x89, 0xAB, 0xCD, 0xEF };

const uint8 p128[16] = {0x01, 0x23, 0x45, 0x67,
                        0x89, 0xAB, 0xCD, 0xEF,
                        0xFE, 0xDC, 0xBA, 0x98,
                        0x76, 0x54, 0x32, 0x10 };

const uint8 k[32] =    {0x00, 0x11, 0x22, 0x33,
                        0x44, 0x55, 0x66, 0x77,
                        0x88, 0x99, 0xAA, 0xBB,
                        0xCC, 0xDD, 0xEE, 0xFF,
```

```
                        0xFF, 0xEE, 0xDD, 0xCC,
                        0xBB, 0xAA, 0x99, 0x88,
                        0x77, 0x66, 0x55, 0x44,
                        0x33, 0x22, 0x11, 0x00 };

int main ()
{
    int return_value = EXIT_SUCCESS;

    fprintf (stdout, "\n\nFOX test vectors generator v1.2");
    fprintf (stdout,   "\n------------------------------\n\n");

    /* FOX64 test vectors                                                    */
    if (fox64_64_16_test (p64, k)) {
        fprintf (stdout, "\nFatal error_exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    if (fox64_128_16_test (p64, k)) {
        fprintf (stdout, "\nFatal error_exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    if (fox64_192_16_test (p64, k)) {
        fprintf (stdout, "\nFatal error_exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    if (fox64_256_16_test (p64, k)) {
        fprintf (stdout, "\nFatal error_exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }

    /* FOX128 test vectors                                                   */
    if (fox128_64_16_test (p128, k)) {
        fprintf (stdout, "\nFatal error_exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    if (fox128_128_16_test (p128, k)) {
        fprintf (stdout, "\nFatal error_exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    if (fox128_192_16_test (p128, k)) {
        fprintf (stdout, "\nFatal error_exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    if (fox128_256_16_test (p128, k)) {
        fprintf (stdout, "\nFatal error_exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }


 error_label:

    return return_value;
}

int fox64_64_16_test (const uint8 *p, const uint8 *k)
{
    FOX64_ctx ctx;
    FOX_key key;
    uint8 c[8];
    uint32 c32[2];
    int i, return_value = EXIT_SUCCESS;

```

```
    if (FOX64_init_ctx (&ctx)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    fprintf (stdout, "\n\nFOX64/16/64 key        : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (k + 4*i));
    }
    fprintf (stdout, "\nFOX64/16/64 message    : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (p + 4*i));
    }
    if (FOX64_init_key (&key, ctx, k, 64, 16)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    memcpy (c, p, 8);
    c32[0] = U8TO32_BIG (c);
    c32[1] = U8TO32_BIG (c + 4);
    FOX64_encrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX64/16/64 ciphertext : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    FOX64_decrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX64/16/64 message    : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    fprintf (stdout, "\n\n");

 error_label:
    FOX64_clean_ctx (ctx);
    FOX64_clean_key (key);

    return return_value;
}

int fox64_128_16_test (const uint8 *p, const uint8 *k)
{
    FOX64_ctx ctx;
    FOX_key key;
    uint8 c[8];
    uint32 c32[2];
    int i, return_value = EXIT_SUCCESS;

    if (FOX64_init_ctx (&ctx)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    fprintf (stdout, "\n\nFOX64/16/128 key        : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (k + 4*i));
    }
    fprintf (stdout, "\nFOX64/16/128 message    : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (p + 4*i));
    }
    if (FOX64_init_key (&key, ctx, k, 128, 16)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    memcpy (c, p, 8);
    c32[0] = U8TO32_BIG (c);
    c32[1] = U8TO32_BIG (c + 4);
    FOX64_encrypt (c32, key, ctx);
```

```
    fprintf (stdout, "\nFOX64/16/128 ciphertext : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    FOX64_decrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX64/16/128 message    : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    fprintf (stdout, "\n\n");

 error_label:
    FOX64_clean_ctx (ctx);
    FOX64_clean_key (key);

    return return_value;
}

int fox64_192_16_test (const uint8 *p, const uint8 *k)
{
    FOX64_ctx ctx;
    FOX_key key;
    uint8 c[8];
    uint32 c32[2];
    int i, return_value = EXIT_SUCCESS;

    if (FOX64_init_ctx (&ctx)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    fprintf (stdout, "\n\nFOX64/16/192 key        : ");
    for (i = 0; i < 6; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (k + 4*i));
    }
    fprintf (stdout, "\nFOX64/16/192 message    : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (p + 4*i));
    }
    if (FOX64_init_key (&key, ctx, k, 192, 16)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    memcpy (c, p, 8);
    c32[0] = U8TO32_BIG (c);
    c32[1] = U8TO32_BIG (c + 4);
    FOX64_encrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX64/16/192 ciphertext : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    FOX64_decrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX64/16/192 message    : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    fprintf (stdout, "\n\n");

 error_label:
    FOX64_clean_ctx (ctx);
    FOX64_clean_key (key);

    return return_value;
}
int fox64_256_16_test (const uint8 *p, const uint8 *k)
{
    FOX64_ctx ctx;
    FOX_key key;
    uint8 c[8];
```

```
    uint32 c32[2];
    int i, return_value = EXIT_SUCCESS;

    if (FOX64_init_ctx (&ctx)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    fprintf (stdout, "\n\nFOX64/16/256 key        : ");
    for (i = 0; i < 8; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (k + 4*i));
    }
    fprintf (stdout, "\nFOX64/16/256 message    : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (p + 4*i));
    }
    if (FOX64_init_key (&key, ctx, k, 256, 16)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    memcpy (c, p, 8);
    c32[0] = U8TO32_BIG (c);
    c32[1] = U8TO32_BIG (c + 4);
    FOX64_encrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX64/16/256 ciphertext : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    FOX64_decrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX64/16/256 message    : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    fprintf (stdout, "\n\n");

 error_label:
    FOX64_clean_ctx (ctx);
    FOX64_clean_key (key);

    return return_value;
}

int fox128_64_16_test (const uint8 *p, const uint8 *k)
{
    FOX128_ctx ctx;
    FOX_key key;
    uint8 c[16];
    uint32 c32[4];
    int i, return_value = EXIT_SUCCESS;

    if (FOX128_init_ctx (&ctx)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    fprintf (stdout, "\n\nFOX128/16/64 key        : ");
    for (i = 0; i < 2; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (k + 4*i));
    }
    fprintf (stdout, "\nFOX128/16/64 message    : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (p + 4*i));
    }
    if (FOX128_init_key (&key, ctx, k, 64, 16)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    memcpy (c, p, 16);
```

```
    c32[0] = U8TO32_BIG (c);
    c32[1] = U8TO32_BIG (c + 4);
    c32[2] = U8TO32_BIG (c + 8);
    c32[3] = U8TO32_BIG (c + 12);
    FOX128_encrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX128/16/64 ciphertext : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    FOX128_decrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX128/16/64 message    : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    fprintf (stdout, "\n\n");

 error_label:
    FOX128_clean_ctx (ctx);
    FOX128_clean_key (key);

    return return_value;
}

int fox128_128_16_test (const uint8 *p, const uint8 *k)
{
    FOX128_ctx ctx;
    FOX_key key;
    uint8 c[16];
    uint32 c32[4];
    int i, return_value = EXIT_SUCCESS;

    if (FOX128_init_ctx (&ctx)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    fprintf (stdout, "\n\nFOX128/16/128 key        : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (k + 4*i));
    }
    fprintf (stdout, "\nFOX128/16/128 message    : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (p + 4*i));
    }
    if (FOX128_init_key (&key, ctx, k, 128, 16)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    memcpy (c, p, 16);
    c32[0] = U8TO32_BIG (c);
    c32[1] = U8TO32_BIG (c + 4);
    c32[2] = U8TO32_BIG (c + 8);
    c32[3] = U8TO32_BIG (c + 12);
    FOX128_encrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX128/16/128 ciphertext : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    FOX128_decrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX128/16/128 message    : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    fprintf (stdout, "\n\n");

 error_label:
    FOX128_clean_ctx (ctx);
    FOX128_clean_key (key);

```

```
    return return_value;
}

int fox128_192_16_test (const uint8 *p, const uint8 *k)
{
    FOX128_ctx ctx;
    FOX_key key;
    uint8 c[16];
    uint32 c32[4];
    int i, return_value = EXIT_SUCCESS;

    if (FOX128_init_ctx (&ctx)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    fprintf (stdout, "\n\nFOX128/16/192 key        : ");
    for (i = 0; i < 6; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (k + 4*i));
    }
    fprintf (stdout, "\nFOX128/16/192 message    : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (p + 4*i));
    }
    if (FOX128_init_key (&key, ctx, k, 192, 16)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    memcpy (c, p, 16);
    c32[0] = U8TO32_BIG (c);
    c32[1] = U8TO32_BIG (c + 4);
    c32[2] = U8TO32_BIG (c + 8);
    c32[3] = U8TO32_BIG (c + 12);
    FOX128_encrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX128/16/192 ciphertext : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    FOX128_decrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX128/16/192 message    : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    fprintf (stdout, "\n\n");

 error_label:
    FOX128_clean_ctx (ctx);
    FOX128_clean_key (key);

    return return_value;
}

int fox128_256_16_test (const uint8 *p, const uint8 *k)
{
    FOX128_ctx ctx;
    FOX_key key;
    uint8 c[16];
    uint32 c32[4];
    int i, return_value = EXIT_SUCCESS;

    if (FOX128_init_ctx (&ctx)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    fprintf (stdout, "\n\nFOX128/16/256 key        : ");
    for (i = 0; i < 8; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (k + 4*i));
    }
```

```
    fprintf (stdout, "\nFOX128/16/256 message    : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", U8TO32_BIG (p + 4*i));
    }
    if (FOX128_init_key (&key, ctx, k, 256, 16)) {
        fprintf (stderr, "\nFatal error...exiting!\n");
        return_value = EXIT_FAILURE;
        goto error_label;
    }
    memcpy (c, p, 16);
    c32[0] = U8TO32_BIG (c);
    c32[1] = U8TO32_BIG (c + 4);
    c32[2] = U8TO32_BIG (c + 8);
    c32[3] = U8TO32_BIG (c + 12);
    FOX128_encrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX128/16/256 ciphertext : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    FOX128_decrypt (c32, key, ctx);
    fprintf (stdout, "\nFOX128/16/256 message    : ");
    for (i = 0; i < 4; i++) {
        fprintf (stdout, "%08X ", c32[i]);
    }
    fprintf (stdout, "\n\n");

 error_label:
    FOX128_clean_ctx (ctx);
    FOX128_clean_key (key);

    return return_value;
}
```