SHA-3 Proposal: Lesamnta

Shoichi Hirose
University of Fukui
hrs_shch@u-fukui.ac.jp

Hidenori Kuwakado
Kobe University
kuwakado@kobe-u.ac.jp

Hirotaka Yoshida
Systems Development Laboratory, Hitachi, Ltd.
hirotaka.yoshida.qv@hitachi.com

Document version 1.0, Date: 30 October 2008

Table of Contents

1 Introduction

This document specifies a family of hash functions, **Lesamnta**[1], which consists of four algorithms: Lesamnta-224, Lesamnta-256, Lesamnta-384, and Lesamnta-512. The four algorithms differ in terms of the sizes of the blocks and words of data that are used during hashing. Figure 1 summarizes the basic properties of all four Lesamnta algorithms.

Algorithm	Message length (bits)	Block size (bits)	Word size (bits)	Message digest size (bits)	Security[2] (bits)
Lesamnta-224	$< 2^{64}$	256	32	224	112
Lesamnta-256	$< 2^{64}$	256	32	256	128
Lesamnta-384	$< 2^{128}$	512	64	384	192
Lesamnta-512	$< 2^{128}$	512	64	512	256

Figure 1: Lesamnta algorithm properties

2 Definitions

2.1 Glossary of Terms and Acronyms

The following definitions are used throughout this specification.

Bit	A binary digit having a value of 0 or 1.
Byte	A group of eight bits.
Block Cipher Key	A cryptographic key used by the Key Expansion routine to generate a set of Round Keys.
Compression function	A function mapping the $(i-1)^{th}$ hash value $H^{(i-1)}$ and the i^{th} message block $M^{(i)}$ to the i^{th} hash value $H^{(i)}$.
Key Expansion	A routine used to generate a series of Round Keys from the Block Cipher Key.
Output function	A function mapping the $(N-1)^{th}$ hash value $H^{(N-1)}$ and the N^{th} message block $M^{(N)}$ to the final hash value $H^{(N)}$.
Round Key	Values derived from the Block Cipher Key by the Key Expansion routine; they are applied to the SubState256 and SubState512 data in the Compression and Output functions.
State	An intermediate hash value.
SubState256	A 64-bit unit of data used in Lesamnta-256; it can be pictured as a rectangular array of bytes with two rows and four columns.

[1]Lesamnta is pronounced like ”Lezanta”

[2]In this context, “security” refers to the fact that a birthday attack on a message digest of size n produces a collision with a workfactor of approximately $2^{n/2}$.

SubState512	A 128-bit unit of data used in Lesamnta-512; it can be pictured as a rectangular array of bytes with four rows and four columns.
S-box	A non-linear substitution table used in several byte substitution transformations and in the Key Expansion routine to perform one-for-one substitution of a byte value.
Word	A group of either 32 bits (4 bytes) or 64 bits (8 bytes), depending on the Lesamnta algorithm.

2.2 Algorithm Parameters and Symbols

The specification uses the following parameters and symbols.

$C^{(round)}$	The $round^{th}$ round constant.
$H^{(i)}$	The i^{th} hash value. $H^{(0)}$ is the **initial** hash value; $H^{(N)}$ is the **final** hash value and is used to determine the message digest.
$H_j^{(i)}$	The j^{th} word of the i^{th} hash value, where $H_0^{(i)}$ is the leftmost word of hash value i.
$K^{(round)}$	The $round^{th}$ Round Key.
l	The length of the message M in bits.
m	The number of bits in a message block $M^{(i)}$.
M	The message to be hashed.
$M^{(i)}$	The message block i, with a size of m bits.
$M_j^{(i)}$	The j^{th} word of the i^{th} message block, where $M_0^{(i)}$ is the leftmost word of message block i.
N	The number of blocks in the padded message.
Nr_comp256	The number of rounds for the **`Compression256()`** function. For this document, Nr_comp256 is 32.
Nr_comp512	The number of rounds for the **`Compression512()`** function. For this document, Nr_comp512 is 32.
Nr_out256	The number of rounds for the **`Output256()`** function. For this document, Nr_out256 is 32.
Nr_out512	The number of rounds for the **`Output512()`** function. For this document, Nr_out512 is 32.
w	The number of bits in a word.
x_j	The w-bit word of the State.
XOR	The exclusive OR operation.
$\oplus$	The exclusive OR operation.
$\vee$	The OR operation.
$\bullet$	Finite field multiplication.
$\|\|$	Concatenation.

2.3 Functions

The specification uses the following functions.

`AddRoundKey256()`	A transformation used in `Compression256()` and `Output256()`, in which a Round Key is added to a SubState256 by using an XOR operation. The length of the Round Key equals the size of the SubState256.
`AddRoundKey512()`	A transformation used in `Compression512()` and `Output512()`, in which a Round Key is added to a SubState512 by using an XOR operation. The length of the Round Key equals the size of the SubState512.
`ByteTranspos256()`	A function used in the Key Expansion routines, which takes an 8-byte word and performs a bytewise transposition.
`ByteTranspos512()`	A function used in the Key Expansion routines, which takes a 16-byte word and performs a bytewise transposition.
`Compression256()`	The Compression function of Lesamnta-256.
`Compression512()`	The Compression function of Lesamnta-512.
$EncComp_{256}$	The encryption function of the block cipher used in the Compression function of Lesamnta-256.
$EncComp_{512}$	The encryption function of the block cipher used in the Compression function of Lesamnta-512.
$EncOut_{256}$	The encryption function of the block cipher used in the Output function of Lesamnta-256.
$EncOut_{512}$	The encryption function of the block cipher used in the Output function of Lesamnta-512.
F_{256}	A non-linear transformation used in a round, consisting of `AddRoundKey256()`, `SubBytes256()`, `ShiftRows256()`, and `MixColumns256()`.
F_{512}	A non-linear transformation used in a round, consisting of `AddRoundKey512()`, `SubBytes512()`, `ShiftRows512()`, and `MixColumns512()`.
F_K	The round function of the key scheduling function.
F_M	The round function of the mixing function.
`KeyExpComp256()`	The Key Expansion routine used in $EncComp_{256}$.
`KeyExpComp512()`	The Key Expansion routine used in $EncComp_{512}$.
`KeyExpOut256()`	The Key Expansion routine used in $EncOut_{256}$.
`KeyExpOut512()`	The Key Expansion routine used in $EncOut_{512}$.
`KeyLinear256()`	A linear function used in the Key Expansion routine `KeyExpComp256()`.
`KeyLinear512()`	A linear function used in the Key Expansion routine `KeyExpComp512()`.

MixColumns256()	A transformation used in **Compression256()** and **Output256()**, which takes all of the columns of a SubState256 and mixes their data (independently of one another) to produce new columns.
MixColumns512()	A transformation used in **Compression512()** and **Output512()**, which takes all of the columns of a SubState512 and mixes their data (independently of one another) to produce new columns.
Output256()	The Output function used in Lesamnta-256.
Output512()	The Output function used in Lesamnta-512.
ShiftRows256()	A transformation used in **Compression256()** and **Output256()**, which processes a SubState256 by cyclically shifting the second row of the SubState256 by one byte.
ShiftRows512()	A transformation used in **Compression512()** and **Output512()**, which processes a SubState512 by cyclically shifting the last three rows of the SubState512 by different offsets.
SubBytes256()	A transformation used in **Compression256()** and **Output256()**, which processes a SubState256 by using a non-linear byte substitution table (i.e., the S-box) that operates independently on each of the SubState256 bytes.
SubBytes512()	A transformation used in **Compression512()** and **Output512()**, which processes a SubState512 by using a non-linear byte substitution table (i.e., the S-box) that operates independently on each of the SubState512 bytes.
SubWords256()	A function used in the Key Expansion routines **KeyExpComp256()** and **KeyExpOut256()**, which takes 8 bytes from two input words and applies a non-linear byte substitution table (i.e., the S-box) to each of the 8 bytes to produce two output words.
SubWords512()	A function used in the Key Expansion routines **KeyExpComp512()** and **KeyExpOut512()**, which takes 16 bytes from two input words and applies a non-linear byte substitution table (i.e., the S-box) to each of the 16 bytes to produce two output words.
WordRotation256()	A function used in **Compression256()**, **Output256()**, and the Key Expansion routines, which takes eight 32-bit words and performs a cyclic permutation.
WordRotation512()	A function used in **Compression512()**, **Output512()**, and the Key Expansion routines, which takes eight 64-bit words and performs a cyclic permutation.

3 Notation and Conventions

3.1 Inputs and Outputs

Lesamnta takes a message with less than 2^{64} bits (for Lesamnta-224 and Lesamnta-256) or 2^{128} bits (for Lesamnta-384 and Lesamnta-512) and outputs a message digest. The message digest ranges in length from 224 to 512 bits, depending on the algorithm.

3.2 Bytes

All byte values in the Lesamnta algorithm are presented as a concatenation of the individual bit values (0 or 1) between braces, in the order $\{b_0, b_1, b_2, b_3, b_4, b_5, b_6, b_7\}$. These bytes are interpreted as finite field elements by using a polynomial representation:

$$b_0x^7 + b_1x^6 + b_2x^5 + b_3x^4 + b_4x^3 + b_5x^2 + b_6x + b_7 = \sum_{i=0}^{7} b_{7-i}x^i.$$

For example, {`01100011`} identifies the specific finite field element $x^6 + x^5 + x + 1$.

It is also convenient to denote byte values by hexadecimal notation, with each of two groups of four bits being denoted by a single character, as illustrated in Fig. 2.

Bit pattern	Character
0000	0
0001	1
0010	2
0011	3

Bit pattern	Character
0100	4
0101	5
0110	6
0111	7

Bit pattern	Character
1000	8
1001	9
1010	a
1011	b

Bit pattern	Character
1100	c
1101	d
1110	e
1111	f

Figure 2: Hexadecimal representations of bit patterns

Hence, the element {`01100011`} can be represented as {`63`}, where the character denoting the four-bit group containing the higher-numbered bits is to the left.

Some finite field operations involve one additional bit (b_{-1}) to the left of an 8-bit byte. Where this extra bit is present, it appears as '{`01`}' immediately preceding the 8-bit byte; for example, a 9-bit sequence is presented as {`01`}{`1b`}.

3.3 Arrays of Bytes

Arrays of bytes are represented in the following form:

$$a_0, a_1, \ldots, a_7.$$

The bytes and the bit ordering within bytes are derived from a 64-bit input sequence

$$input_0, input_1, \ldots, input_{63},$$

as follows:

$$\begin{aligned} a_0 &= \{input_0, input_1, \ldots, input_7\}, \\ a_1 &= \{input_8, input_9, \ldots, input_{15}\}, \\ &\vdots \\ a_7 &= \{input_{56}, input_{57}, \ldots, input_{63}\}. \end{aligned}$$

The pattern can be extended to longer sequences (i.e., for Lesamnta-384/512), so that, in general,

$$a_n = \{input_{8n}, input_{8n+1}, \ldots, input_{8n+7}\}.$$

Taking the notation of Secs. 3.2 and 3.3 together, Fig. 3 shows how the bits within each byte are numbered.

Input bit sequence	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
Byte number	0								1								...
Bit number in byte	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	...
Bit number in word	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...

Figure 3: Indices for bytes and bits

3.4 Endian

Throughout this document, the big-endian convention is followed in expressing both 32- and 64-bit words, so that within each word, the most significant bit is stored in the leftmost bit position.

3.5 Bit Strings

A word is a *w*-bit string that can be represented as a sequence of hexadecimal, or hex, digits. To convert a word to hex digits, each 4-bit string is converted to its hex digit equivalent, as shown in Fig. 2. For example, the 32-bit string

```
1010 0001 0000 0011 1111 1110 0010 0011
```

can be expressed as `a103fe23`, and the 64-bit string

```
1010 0001 0000 0011 1111 1110 0010 0011
0011 0010 1110 1111 0011 0000 0001 1010
```

can be expressed as `a103fe2332ef301a`.

3.6 Message Block

For the Lesamnta algorithms, the size of the **message block** - m bits - depends on the algorithm.

1. For **Lesamnta-224** and **Lesamnta-256**, each message block has **256 bits**, which are represented as a sequence of eight **32-bit words**.

2. For **Lesamnta-384** and **Lesamnta-512**, each message block has **512 bits**, which are represented as a sequence of eight **64-bit words**.

3.7 SubState256

For a 64-bit part of a state, the Lesamnta-224 and Lesamnta-256 algorithms' operations are performed on a two-dimensional array of bytes called a **SubState256**. The SubState256 consists of two rows of bytes, each containing four bytes. In a SubState256 array, denoted by the symbol s, each individual byte has two indices, with its row number r in the range $0 \le r < 2$ and its column number c in the range $0 \le c < 4$. This allows an individual byte of the SubState256 to be referred to as either $s_{r,c}$ or $s[r, c]$.

At the start of the F_{256} function in each round of **`Compression256()`** and **`Output256()`**, as described in Sec. 5.3, the input - the array of bytes $in_0, in_1, \dots, in_7$ - is copied into the SubState256 array, as illustrated in Fig. 4. The **`Compression256()`** or **`Output256()`** function is then executed on this SubState256 array, after which the array's final set of values is copied to the output: an array of bytes $out_0, out_1, \dots, out_7$.

Input bytes

in_0	in_2	in_4	in_6
in_1	in_3	in_5	in_7

SubState256

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$

Output bytes

out_0	out_2	out_4	out_6
out_1	out_3	out_5	out_7

Figure 4: SubState256 array input and output

Hence, at the beginning of the F_{256} function, the input array in is copied to the SubState256 array, according to this scheme:

$$s[r, c] = in[r + 2c], \quad \text{for } 0 \le r < 2 \text{ and } 0 \le c < 4,$$

and at the end of the F_{256} function, the SubState256 array is copied to the output array out as follows:

$$out[r + 2c] = s[r, c], \quad \text{for } 0 \le r < 2 \text{ and } 0 \le c < 4.$$

3.8 SubState512

For a 128-bit part of a state, the Lesamnta-384 and Lesamnta-512 algorithms' operations are performed on a two-dimensional array of bytes called a **SubState512**. The SubState512 consists of four rows of bytes, each containing four bytes. In a SubState512 array, denoted by the symbol s, each individual byte has two indices, with its row number r in the range $0 \leq r < 4$ and its column number c in the range $0 \leq c < 4$. This allows an individual byte of the SubState512 to be referred to as either $s_{r,c}$ or $s[r, c]$.

At the start of the F_{512} function in each round of **`Compression512()`** and **`Output512()`**, as described in Sec. 5.5, the input - the array of bytes $in_0, in_1, \ldots, in_{15}$ - is copied into the SubState512 array, as illustrated in Fig. 5. The **`Compression512()`** or **`Output512()`** function is then executed on this SubState512 array, after which the array's final set of values is copied to the output: an array of bytes $out_0, out_1, \ldots, out_{15}$.

Input bytes

in_0	in_4	in_8	in_{12}
in_1	in_5	in_9	in_{13}
in_2	in_6	in_{10}	in_{14}
in_3	in_7	in_{11}	in_{15}

$\rightarrow$

SubState512

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$

$\rightarrow$

Output bytes

out_0	out_4	out_8	out_{12}
out_1	out_5	out_9	out_{13}
out_2	out_6	out_{10}	out_{14}
out_3	out_7	out_{11}	out_{15}

Figure 5: SubState512 array input and output

Hence, at the beginning of the F_{512} function, the input array in is copied to the SubState512 array, according to this scheme:

$$s[r, c] = in[r + 4c], \quad \text{for } 0 \leq r < 4 \text{ and } 0 \leq c < 4,$$

and at the end of the F_{512} function, the SubState512 array is copied to the output array out as follows:

$$out[r + 4c] = s[r, c], \quad \text{for } 0 \leq r < 4 \text{ and } 0 \leq c < 4.$$

4 Mathematical Preliminaries

Lesamnta uses certain operations in the finite field GF(2^8). Such a finite field has many different representations. We fix a characteristic polynomial and represent an element of GF(2^8) by a polynomial.

First, we define the finite field GF(2^8) as GF(2^8) = GF(2)[x]/($\varphi(x)$), where the polynomial $\varphi(x)$ is given as follows:

$$\varphi(x) = x^8 + x^4 + x^3 + x + 1 = \{\texttt{01}\}\{\texttt{1b}\}.$$

4.1 Addition

The sum of two polynomials over GF(2^8) is a polynomial whose coefficients are given by the sums modulo 2 of the corresponding coefficients. In other words, addition is calculated by a bitwise XOR. For example, the sum of {57} and {a3} is calculated as follows:

$$\begin{aligned}\{\texttt{57}\} + \{\texttt{a3}\} &= (x^6 + x^4 + x^2 + x + 1) + (x^7 + x^5 + x + 1) \\ &= x^7 + x^6 + x^5 + x^4 + x^2 \\ &= \{\texttt{f4}\}.\end{aligned}$$

4.2 Multiplication

Multiplication in GF(2^8) (denoted by $\bullet$) can be divided into two steps. First, we define the multiplication of any element $f(x) = \sum_{i=0}^{7} a_{7-i}x^i$ and x by using $\varphi(x)$ as follows:

$$x \cdot f(x) = \sum_{i=0}^{7} a_{7-i}x^{i+1} \bmod \varphi(x).$$

For example, the multiplication of {02} and {87} is calculated as follows:

$$\begin{aligned}\{\texttt{02}\} \bullet \{\texttt{87}\} &= x \cdot (x^7 + x^2 + x + 1) \\ &= x^8 + x^3 + x^2 + x \\ &= (x^4 + x^3 + x + 1) + x^3 + x^2 + x \\ &= x^4 + x^2 + 1 \\ &= \{\texttt{15}\}.\end{aligned}$$

Second, we calculate $x^i \cdot f(x)$ for any i by iterative application of the above definition.

5 Specification

This chapter describes the Lesamnta algorithms.

5.1 Round Constants

5.1.1 Lesamnta-224/256

Lesamnta-224 and Lesamnta-256 use the same sequence of Nr_comp256(=Nr_out256) constant 64-bit words, $C^{(round)}$. These words are defined by the following equation:

$$C^{(round)} = \texttt{000000XY000000ZW},$$

where $\texttt{XY}$ is $2*round+1$ in hex, and $\texttt{ZW}$ is $2*round$ in hex. The round constants $C^{(0)}, C^{(1)}, \ldots, C^{(31)}$ are the following (from left to right, in hex):

```
0000000100000000, 0000000300000002, 0000000500000004, 0000000700000006,
0000000900000008, 0000000b0000000a, 0000000d0000000c, 0000000f0000000e,
0000001100000010, 0000001300000012, 0000001500000014, 0000001700000016,
0000001900000018, 0000001b0000001a, 0000001d0000001c, 0000001f0000001e,
0000002100000020, 0000002300000022, 0000002500000024, 0000002700000026,
0000002900000028, 0000002b0000002a, 0000002d0000002c, 0000002f0000002e,
0000003100000030, 0000003300000032, 0000003500000034, 0000003700000036,
0000003900000038, 0000003b0000003a, 0000003d0000003c, 0000003f0000003e.
```

5.1.2 Lesamnta-384/512

Lesamnta-384 and Lesamnta-512 use the same sequence of Nr_comp512(=Nr_out512) constant 128-bit words, $C^{(round)}$. These words are defined by the following equation:

$$C^{(round)} = \texttt{00000000000000XY00000000000000ZW},$$

where $\texttt{XY}$ is $2*round+1$ in hex, and $\texttt{ZW}$ is $2*round$ in hex. The round constants $C^{(0)}, C^{(1)}, \ldots, C^{(31)}$ are the following (from left to right, in hex):

```
00000000000000010000000000000000, 00000000000000030000000000000002,
00000000000000050000000000000004, 00000000000000070000000000000006,
0000000000000009000000000000008, 000000000000000b000000000000000a,
000000000000000d000000000000000c, 000000000000000f000000000000000e,
00000000000000110000000000000010, 00000000000000130000000000000012,
00000000000000150000000000000014, 00000000000000170000000000000016,
00000000000000190000000000000018, 000000000000001b000000000000001a,
000000000000001d000000000000001c, 000000000000001f000000000000001e,
00000000000000210000000000000020, 00000000000000230000000000000022,
00000000000000250000000000000024, 00000000000000270000000000000026,
00000000000000290000000000000028, 000000000000002b000000000000002a,
000000000000002d000000000000002c, 000000000000002f000000000000002e,
00000000000000310000000000000030, 00000000000000330000000000000032,
00000000000000350000000000000034, 00000000000000370000000000000036,
00000000000000390000000000000038, 000000000000003b000000000000003a,
000000000000003d000000000000003c, 000000000000003f000000000000003e.
```

5.2 Preprocessing

Preprocessing takes place before hash computation begins. This preprocessing consists of three steps: padding the message M (Sec. 5.2.1), parsing the padded message into message blocks (Sec. 5.2.2), and setting the initial hash value $H^{(0)}$ (Sec. 5.2.3).

5.2.1 Padding the Message

The message M is padded before hash computation begins. The purpose of this padding is to ensure that the message consists of a multiple of 256 or 512 bits, depending on the algorithm.

5.2.1.1 Lesamnta-224/256

Suppose that the length of message M is l bits. Append the bit "1" to the end of the message, followed by $k + 191$ zero bits, where k is the minimum non-negative integer such that $l + 1 + k + 191 \equiv 192 \pmod{256}$. Then, append a 64-bit block equal to the number l as expressed in binary representation. The length of the padded message should now be a multiple of 256 bits.

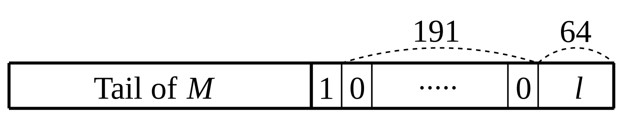

Figure 6: Last two blocks of a padded message for Lesamnta-224/256 ($l \equiv 0 \pmod{256}$)

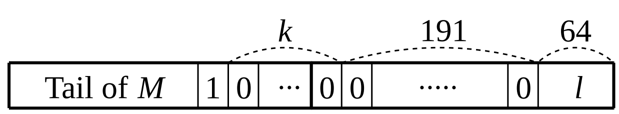

Figure 7: Last two blocks of a padded message for Lesamnta-224/256 ($l \not\equiv 0 \pmod{256}$)

5.2.1.2 Lesamnta-384/512

Suppose that the length of message M is l bits. Append the bit “1” to the end of the message, followed by $k + 383$ zero bits, where k is the minimum non-negative integer such that $l + 1 + k + 383 \equiv 384 \pmod{512}$. Then, append a 128-bit block equal to the number l as expressed in binary representation. The length of the padded message should now be a multiple of 512 bits.

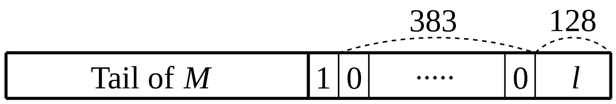

Figure 8: Last two blocks of a padded message for Lesamnta-384/512 ($l \equiv 0 \pmod{512}$)

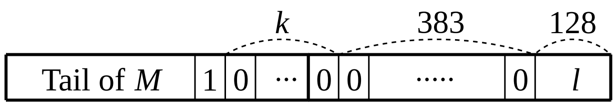

Figure 9: Last two blocks of a padded message for Lesamnta-384/512 ($l \not\equiv 0 \pmod{512}$)

5.2.2 Parsing the Padded Message

After a message has been padded, it must be parsed into N m-bit blocks before the hash computation can begin.

5.2.2.1 Lesamnta-224/256

For Lesamnta-224 and Lesamnta-256, the padded message is parsed into N 256-bit blocks: $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$. Since the 256 bits of the input block can be expressed as eight 32-bit words, the first 32 bits of message block $M^{(i)}$ are denoted as $M_0^{(i)}$; the next 32 bits, as $M_1^{(i)}$; and so on up to $M_7^{(i)}$.

5.2.2.2 Lesamnta-384/512

For Lesamnta-384 and Lesamnta-512, the padded message is parsed into N 512-bit blocks: $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$. Since the 512 bits of the input block can be expressed as eight 64-bit words, the first 64 bits of message block $M^{(i)}$ are denoted as $M_0^{(i)}$; the next 64 bits, as $M_1^{(i)}$; and so on up to $M_7^{(i)}$.

5.2.3 Setting the Initial Hash Value

Before hash computation begins for each of the Lesamnta algorithms, the initial hash value $H^{(0)}$ must be set. The size of the words in $H^{(0)}$ depends on the message digest size.

5.2.3.1 Lesamnta-224

For Lesamnta-224, the initial hash value $H^{(0)}$ consists of the following eight 32-bit words, in hex:

$$\begin{aligned} H_0^{(0)} &= \texttt{00000224}, \\ H_1^{(0)} &= \texttt{00000224}, \\ H_2^{(0)} &= \texttt{00000224}, \\ H_3^{(0)} &= \texttt{00000224}, \\ H_4^{(0)} &= \texttt{00000224}, \\ H_5^{(0)} &= \texttt{00000224}, \\ H_6^{(0)} &= \texttt{00000224}, \\ H_7^{(0)} &= \texttt{00000224}. \end{aligned}$$

5.2.3.2 Lesamnta-256

For Lesamnta-256, the initial hash value $H^{(0)}$ consists of the following eight 32-bit words, in hex:

$$\begin{aligned} H_0^{(0)} &= \texttt{00000256}, \\ H_1^{(0)} &= \texttt{00000256}, \\ H_2^{(0)} &= \texttt{00000256}, \\ H_3^{(0)} &= \texttt{00000256}, \\ H_4^{(0)} &= \texttt{00000256}, \\ H_5^{(0)} &= \texttt{00000256}, \\ H_6^{(0)} &= \texttt{00000256}, \\ H_7^{(0)} &= \texttt{00000256}. \end{aligned}$$

5.2.3.3 Lesamnta-384

For Lesamnta-384, the initial hash value $H^{(0)}$ consists of the following eight 64-bit words, in hex:

$$\begin{aligned} H_0^{(0)} &= \texttt{0000000000000384}, \\ H_1^{(0)} &= \texttt{0000000000000384}, \\ H_2^{(0)} &= \texttt{0000000000000384}, \\ H_3^{(0)} &= \texttt{0000000000000384}, \\ H_4^{(0)} &= \texttt{0000000000000384}, \\ H_5^{(0)} &= \texttt{0000000000000384}, \\ H_6^{(0)} &= \texttt{0000000000000384}, \\ H_7^{(0)} &= \texttt{0000000000000384}. \end{aligned}$$

5.2.3.4 Lesamnta-512

For Lesamnta-512, the initial hash value $H^{(0)}$ consists of the following eight 64-bit words, in hex:

$$
\begin{aligned}
H_0^{(0)} &= \texttt{0000000000000512},\\
H_1^{(0)} &= \texttt{0000000000000512},\\
H_2^{(0)} &= \texttt{0000000000000512},\\
H_3^{(0)} &= \texttt{0000000000000512},\\
H_4^{(0)} &= \texttt{0000000000000512},\\
H_5^{(0)} &= \texttt{0000000000000512},\\
H_6^{(0)} &= \texttt{0000000000000512},\\
H_7^{(0)} &= \texttt{0000000000000512}.
\end{aligned}
$$

5.3 Lesamnta-256 Algorithm

Lesamnta-256 can be used to hash a message M having a length of l bits, where $0 \le l < 2^{64}$. The final result of Lesamnta-256 is a 256-bit message digest.

5.3.1 Lesamnta-256 Preprocessing

1. Pad the message M, according to Sec. 5.2.1.1.
2. Parse the padded message into N 256-bit message blocks $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$, according to Sec. 5.2.2.1.
3. Set the initial hash value $H^{(0)}$, as specified in Sec. 5.2.3.2.

5.3.2 Lesamnta-256 Computation

The Lesamnta-256 hash computation uses the round constants defined in Sec. 5.1.1.

After preprocessing is completed, each message block $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$ is processed in order, as follows:

```
for i = 1 to N - 1
   Compression256(H^(i-1), M^(i))
end for

Output256(H^(N-1), M^(N))
```

Figure 10: Pseudocode for the Lesamnta-256 computation

The resulting 256-bit message digest of the message M is

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}.$$

The Compression function **`Compression256()`** is shown in the following pseudocode:

```
Compression256(word chain[8], word mb[8])
begin
   word K[Nr_comp256][2]
   word x[8]
   word substate256[2]

1. Prepare the key schedule of the block cipher EncComp256:

   KeyExpComp256(chain, K)

2. Compute the encryption function of the block cipher EncComp256:

   for j = 0 to 7
      x[j] = mb[j]
   end for

   for round = 0 to Nr_comp256 - 1
      substate256[0] = x[4]
      substate256[1] = x[5]

      AddRoundKey256(substate256, K[round])

      for iteration = 0 to 3
         SubBytes256(substate256)
         ShiftRows256(substate256)
         MixColumns256(substate256)
      end for

      x[6] = x[6] ⊕ substate256[0]
      x[7] = x[7] ⊕ substate256[1]

      WordRotation256(x)
   end for

3. Compute the intermediate hash value H^(i):

   for j = 0 to 7
      chain[j] = x[j] ⊕ mb[j]
   end for
end
```

Figure 11: Pseudocode for **`Compression256()`**

At the end of **`Compression256()`**, $H^{(i)}$ is given by `chain[0]`||`chain[1]`||...||`chain[7]`.

Figure 12 illustrates the round function of the block cipher $EncComp_{256}$.

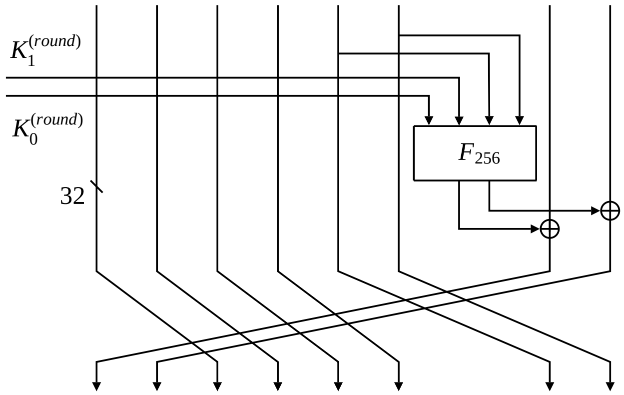

Figure 12: Round function in $EncComp_{256}$

The Output function **Output256()** is shown in the following pseudocode:

```
Output256(word chain[8], word mb[8])
begin
   word K[Nr_out256][2]
   word x[8]
   word substate256[2]

1. Prepare the key schedule of the block cipher EncOut256:

   KeyExpOut256(chain, K)

2. Compute the encryption function of the block cipher EncOut256:

   for j = 0 to 7
      x[j] = mb[j]
   end for

   for round = 0 to Nr_out256 - 1
      substate256[0] = x[4]
      substate256[1] = x[5]

      AddRoundKey256(substate256, K[round])

      for iteration = 0 to 3
         SubBytes256(substate256)
         ShiftRows256(substate256)
         MixColumns256(substate256)
      end for

      x[6] = x[6] ⊕ substate256[0]
      x[7] = x[7] ⊕ substate256[1]

      WordRotation256(x)
   end for

3. Compute the final hash value H^(N):

   for j = 0 to 7
      chain[j] = x[j] ⊕ mb[j]
   end for
end
```

Figure 13: Pseudocode for **Output256()**

At the end of **Output256()**, $H^{(N)}$ is given by chain[0]||chain[1]||...||chain[7].

Note that **Compression256()** and **Output256()** work in a similar manner. The differences between two functions are shown in bold.

5.3.2.1 `SubBytes256()` Transformation

The **`SubBytes256()`** transformation is a non-linear byte substitution that operates independently on each byte of the SubState256 by using the substitution table S-box, defined in Fig. 15. The **`SubBytes256()`** transformation proceeds as follows:

$$s'_{r,c} = \text{S-box}(s_{r,c}), \quad \text{for } 0 \le r < 2 \text{ and } 0 \le c < 4.$$

Figure 14 illustrates the **`SubBytes256()`** transformation.

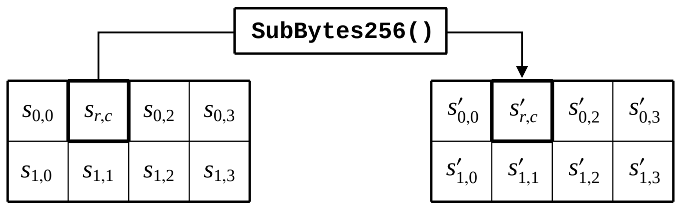

Figure 14: **`SubBytes256()`** applies the S-box to each byte of the SubState256

The S-box used in the **`SubBytes256()`** transformation is shown in hexadecimal form in Fig. 15. For example, if $s_{1,0}$ = {53}, then the substitution value is determined by the intersection of the row with index '5' and the column with index '3' in Fig. 15. This results in $s'_{1,0}$ having a value of {ed}.

y

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

Figure 15: S-box: substitution values for the byte **{xy}** (in hexadecimal format)

5.3.2.2 `ShiftRows256()` Transformation

In the **`ShiftRows256()`** transformation, the bytes in the second row of the SubState256 are cyclically shifted over different numbers of bytes (offsets). The first row is not shifted. Specifically, the **`ShiftRows256()`** transformation proceeds as follows:

$$S'_{1,c} = S_{1,(c+1) \bmod 4}, \quad \text{for } 0 \le c < 4.$$

Figure 16 illustrates the **`ShiftRows256()`** transformation.

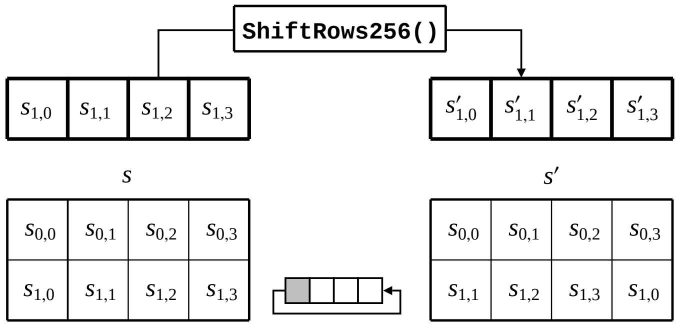

Figure 16: **`ShiftRows256()`** cyclically shifts the second row in the SubState256

5.3.2.3 `MixColumns256()` Transformation

The **`MixColumns256()`** transformation uses multiplication over a finite field, as defined in Sec. 4.2, in the following manner:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \end{bmatrix} = \begin{bmatrix} 02 & 01 \\ 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \end{bmatrix}, \quad \text{for } 0 \le c < 4.$$

As a result of this multiplication, the two bytes in a column are replaced by the following:

$$\begin{aligned} s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus s_{1,c}, \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}). \end{aligned}$$

Figure 17 illustrates the **`MixColumns256()`** transformation.

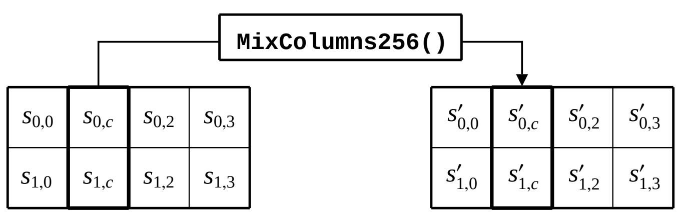

Figure 17: **`MixColumns256()`** operates on the SubState256 column by column

5.3.2.4 `AddRoundKey256()` Transformation

In the **`AddRoundKey256()`** transformation, the two-word Round Key $K^{(round)} = K_0^{(round)} \| K_1^{(round)}$ from the key schedule, as described in Secs. 5.3.2.6 and 5.3.2.7, is added to the SubState256 by a simple bitwise XOR operation. The two words are each added into the SubState256, such that

$$\begin{aligned}\left[s'_{0,0}, s'_{1,0}, s'_{0,1}, s'_{1,1}\right] &= [s_{0,0}, s_{1,0}, s_{0,1}, s_{1,1}] \oplus K_0^{(round)},\\ \left[s'_{0,2}, s'_{1,2}, s'_{0,3}, s'_{1,3}\right] &= [s_{0,2}, s_{1,2}, s_{0,3}, s_{1,3}] \oplus K_1^{(round)}.\end{aligned}$$

5.3.2.5 `WordRotation256()`

`WordRotation256()` takes eight 32-bit words $x_0, x_1, \ldots, x_7$ as input and performs a cyclic permutation. The function proceeds as follows:

$$x'_{j+2 \bmod 8} = x_j, \quad \text{for } 0 \le j < 8.$$

5.3.2.6 `KeyExpComp256()`

During the process of **`Compression256(`**$H^{(i-1)}, M^{(i)}$**`)`**, the $EncComp_{256}$ block cipher takes the intermediate hash value $H^{(i-1)}$ as the Block Cipher Key and performs the Key Expansion routine **`KeyExpComp256()`** to generate a key schedule.

`KeyExpComp256()` generates a total of $2 * Nr\_\text{comp256}$ words: the algorithm requires an initial set of eight words, and each of the Nr_comp256 rounds requires eight words of key data. The resulting key schedule consists of a linear array of words, with i in the range of $0 \le i < 2 * Nr\_\text{comp256}$. The round constant word array $C^{(round)} = C_0^{(round)} \| C_1^{(round)}$ is defined in Sec. 5.1.1. Expansion of the input key into the key schedule proceeds according to the pseudocode shown in Fig. 18.

`SubWords256()` is a function that takes 8-byte input words and applies the S-box (Fig. 15) to each of the 8 bytes to produce output words. **`WordRotation256()`** is defined in Sec. 5.3.2.5.

Each of the functions **`KeyLinear256()`** and **`ByteTranspos256()`** takes 8 bytes $a_0, a_1, \ldots, a_7$ as input and performs a bytewise permutation. **`KeyLinear256()`** is a bytewise operation given by the following equation, where multiplication over GF(2^8) is defined in Sec. 4.2:

$$\begin{bmatrix} a'_i \\ a'_{i+1} \\ a'_{i+2} \\ a'_{i+3} \end{bmatrix} = \begin{bmatrix} \mathtt{02} & \mathtt{03} & \mathtt{01} & \mathtt{01} \\ \mathtt{01} & \mathtt{02} & \mathtt{03} & \mathtt{01} \\ \mathtt{01} & \mathtt{01} & \mathtt{02} & \mathtt{03} \\ \mathtt{03} & \mathtt{01} & \mathtt{01} & \mathtt{02} \end{bmatrix} \begin{bmatrix} a_i \\ a_{i+1} \\ a_{i+2} \\ a_{i+3} \end{bmatrix}, \qquad i = 0, 4.$$

$$\begin{aligned}a'_i &= (\{\mathtt{02}\} \bullet a_i) \oplus (\{\mathtt{03}\} \bullet a_{i+1}) \oplus a_{i+2} \oplus a_{i+3},\\ a'_{i+1} &= a_i \oplus (\{\mathtt{02}\} \bullet a_{i+1}) \oplus (\{\mathtt{03}\} \bullet a_{i+2}) \oplus a_{i+3},\\ a'_{i+2} &= a_i \oplus a_{i+1} \oplus (\{\mathtt{02}\} \bullet a_{i+2}) \oplus (\{\mathtt{03}\} \bullet a_{i+3}),\\ a'_{i+3} &= (\{\mathtt{03}\} \bullet a_i) \oplus a_{i+1} \oplus a_{i+2} \oplus (\{\mathtt{02}\} \bullet a_{i+3}).\end{aligned}$$

```
KeyExpComp256(word chain[8], word K[Nr_comp256][2])
begin
   word t[2]  /* The structure is not a SubState256 */

   for round = 0 to Nr_comp256 - 1
      t[0] = chain[4] ⊕ C[round][0]
      t[1] = chain[5] ⊕ C[round][1]

      SubWords256(t)
      KeyLinear256(t)
      ByteTranspos256(t)

      chain[6] = chain[6] ⊕ t[0]
      chain[7] = chain[7] ⊕ t[1]

      WordRotation256(chain)
      K[round][0] = chain[2]
      K[round][1] = chain[3]
   end for
end
```

Figure 18: Pseudocode for **`KeyExpComp256()`**

`ByteTranspos256()` performs bytewise transposition in the following manner:

$$a'_0 = a_4, \quad a'_1 = a_5, \quad a'_2 = a_2, \quad a'_3 = a_3,$$
$$a'_4 = a_0, \quad a'_5 = a_1, \quad a'_6 = a_6, \quad a'_7 = a_7.$$

Figure 19 illustrates the **`ByteTranspos256()`** transformation.

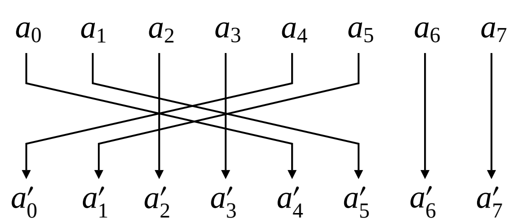

Figure 19: **`ByteTranspos256()`** transformation

5.3.2.7 `KeyExpOut256()`

During the process of **`Output256(`**$H^{(N-1)}, M^{(N)}$**`)`**, the $EncOut_{256}$ block cipher takes the intermediate hash value $H^{(N-1)}$ as the Block Cipher Key and performs the Key Expansion routine **`KeyExpOut256()`** to generate a key schedule.

`KeyExpOut256()` generates a total of $2 * Nr\_\text{out}256$ words: the algorithm requires an initial set of eight words, and each of the $Nr\_\text{out}256$ rounds requires eight words of key data. The resulting key schedule consists of a linear array of words, with i in the range of $0 \leq i < 2 * Nr\_\text{out}256$. The round constant word array $C^{(round)} = C_0^{(round)} \| C_1^{(round)}$ is defined in Sec. 5.1.1. Expansion of the input key into the key schedule proceeds according to the pseudocode shown in Fig. 20.

The functions **`SubBytes256()`**, **`ShiftRows256()`**, **`MixColumns256()`**, and **`WordRotation256()`** are defined in Secs. 5.3.2.1, 5.3.2.2, 5.3.2.3, and 5.3.2.5, respectively.

```
KeyExpOut256(word chain[8], word K[Nr_out256][2])
begin
   word substate256[2]

   for round = 0 to Nr_out256 - 1
      substate256[0] = chain[4] ⊕ C[round][0]
      substate256[1] = chain[5] ⊕ C[round][1]

      for iteration = 0 to 3
         SubBytes256(substate256)
         ShiftRows256(substate256)
         MixColumns256(substate256)
      end for

      chain[6] = chain[6] ⊕ substate256[0]
      chain[7] = chain[7] ⊕ substate256[1]

      WordRotation256(chain)
      K[round][0] = chain[2]
      K[round][1] = chain[3]
   end for
end
```

Figure 20: Pseudocode for **`KeyExpOut256()`**

5.4 Lesamnta-224 Algorithm

Lesamnta-224 can be used to hash a message M having a length of l bits, where $0 \leq l < 2^{64}$. The algorithm is defined in exactly the same manner as for Lesamnta-256 (Sec. 5.3), with the following two exceptions:

1. The initial hash value $H^{(0)}$ is set as specified in Sec. 5.2.3.1.

2. The 224-bit message digest is obtained by truncating the final hash value $H^{(N)}$ to its leftmost 224 bits:
$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)}.$$

5.5 Lesamnta-512 Algorithm

Lesamnta-512 can be used to hash a message M having a length of l bits, where $0 \le l < 2^{128}$. The final result of Lesamnta-512 is a 512-bit message digest.

5.5.1 Lesamnta-512 Preprocessing

1. Pad the message M, according to Sec. 5.2.1.2.
2. Parse the padded message into N 512-bit message blocks $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$, according to Sec. 5.2.2.2.
3. Set the initial hash value $H^{(0)}$, as specified in Sec. 5.2.3.4.

5.5.2 Lesamnta-512 Computation

The Lesamnta-512 hash computation uses the round constants defined in Sec. 5.1.2.

After preprocessing is completed, each message block $M^{(1)}, M^{(2)}, \ldots, M^{(N)}$ is processed in order, as follows:

```
for i = 1 to N - 1
   Compression512(H^(i-1), M^(i))
end for

Output512(H^(N-1), M^(N))
```

Figure 21: Pseudocode for the Lesamnta-512 computation

The resulting 512-bit message digest of the message M is

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}.$$

The Compression function **`Compression512()`** is shown in the following pseudocode:

```
Compression512(word chain[8], word mb[8])
begin
   word K[Nr_comp512][2]
   word x[8]
   word substate512[2]

1. Prepare the key schedule of the block cipher EncComp512:

   KeyExpComp512(chain, K)

2. Compute the encryption function of the block cipher EncComp512:

   for j = 0 to 7
      x[j] = mb[j]
   end for

   for round = 0 to Nr_comp512 - 1
      substate512[0] = x[4]
      substate512[1] = x[5]

      AddRoundKey512(substate512, K[round])

      for iteration = 0 to 3
         SubBytes512(substate512)
         ShiftRows512(substate512)
         MixColumns512(substate512)
      end for

      x[6] = x[6] ⊕ substate512[0]
      x[7] = x[7] ⊕ substate512[1]

      WordRotation512(x)
   end for

3. Compute the intermediate hash value H^(i):

   for j = 0 to 7
      chain[j] = x[j] ⊕ mb[j]
   end for
end
```

Figure 22: Pseudocode for **`Compression512()`**

At the end of **`Compression512()`**, $H^{(i)}$ is given by `chain[0]`$\|$`chain[1]`$\|\ldots\|$`chain[7]`.

Figure 23 illustrates the round function of the block cipher $EncComp_{512}$.

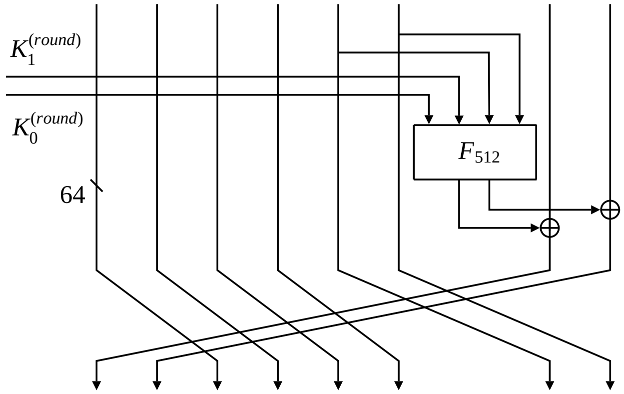

Figure 23: Round function in $EncComp_{512}$

The Output function **`Output512()`** is shown in the following pseudocode:

```
Output512(word chain[8], word mb[8])
begin
   word K[Nr_out512][2]
   word x[8]
   word substate512[2]

1. Prepare the key schedule of the block cipher EncOut512:

   KeyExpOut512(chain, K)

2. Compute the encryption function of the block cipher EncOut512:

   for j = 0 to 7
      x[j] = mb[j]
   end for

   for round = 0 to Nr_out512 - 1
      substate512[0] = x[4]
      substate512[1] = x[5]

      AddRoundKey512(substate512, K[round])

      for iteration = 0 to 3
         SubBytes512(substate512)
         ShiftRows512(substate512)
         MixColumns512(substate512)
      end for

      x[6] = x[6] ⊕ substate512[0]
      x[7] = x[7] ⊕ substate512[1]

      WordRotation512(x)
   end for

3. Compute the final hash value H^(N):

   for j = 0 to 7
      chain[j] = x[j] ⊕ mb[j]
   end for
end
```

Figure 24: Pseudocode for **`Output512()`**

At the end of **`Output512()`**, $H^{(N)}$ is given by `chain[0]`||`chain[1]`||...||`chain[7]`.

Note that **`Compression512()`** and **`Output512()`** work in a similar manner. The differences between the two functions are shown in bold.

5.5.2.1 **`SubBytes512()`** Transformation

The **`SubBytes512()`** transformation is a non-linear byte substitution that operates independently on each byte of the SubState512 by using the substitution table S-box, defined in Fig. 15. The **`SubBytes512()`** transformation proceeds as follows:

$$s'_{r,c} = \text{S-box}(s_{r,c}), \quad \text{for } 0 \le r < 4 \text{ and } 0 \le c < 4.$$

Figure 25 illustrates the **`SubBytes512()`** transformation.

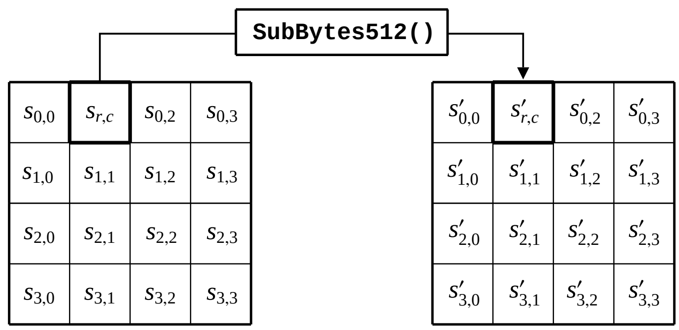

Figure 25: **`SubBytes512()`** applies the S-box to each byte of the SubState512

5.5.2.2 **`ShiftRows512()`** Transformation

In the **`ShiftRows512()`** transformation, the bytes in the last three rows of the SubState512 are cyclically shifted over different numbers of bytes (offsets). The first row is not shifted. Specifically, the **`ShiftRows512()`** transformation proceeds as follows:

$$S'_{r,c} = S_{r,(c+r) \bmod 4}, \quad \text{for } 0 < r < 4 \text{ and } 0 \le c < 4,$$

Figure 26 illustrates the **`ShiftRows512()`** transformation.

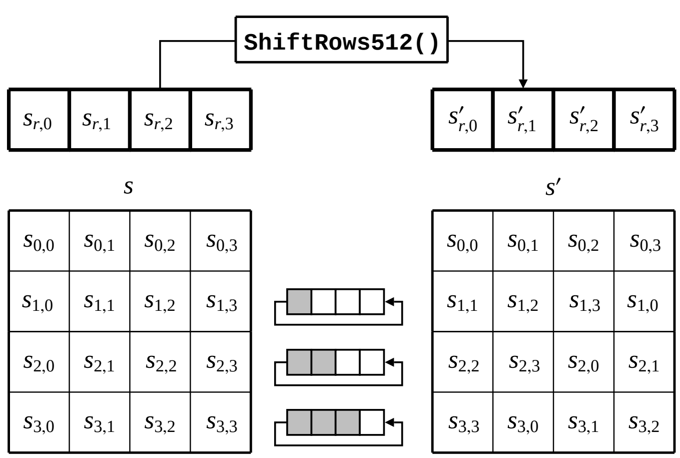

Figure 26: **`ShiftRows512()`** cyclically shifts the last three rows in the SubState512

5.5.2.3 **`MixColumns512()`** Transformation

The **`MixColumns512()`** transformation uses multiplication over a finite field, as defined in Sec. 4.2, in the following manner:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}, \quad \text{for } 0 \le c < 4.$$

As a result of this multiplication, the two bytes in a column are replaced by the following:

$$\begin{aligned} s'_{0,c} &= (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}, \\ s'_{1,c} &= s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}, \\ s'_{2,c} &= s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c}), \\ s'_{3,c} &= (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}). \end{aligned}$$

Figure 27 illustrates the **`MixColumns512()`** transformation.

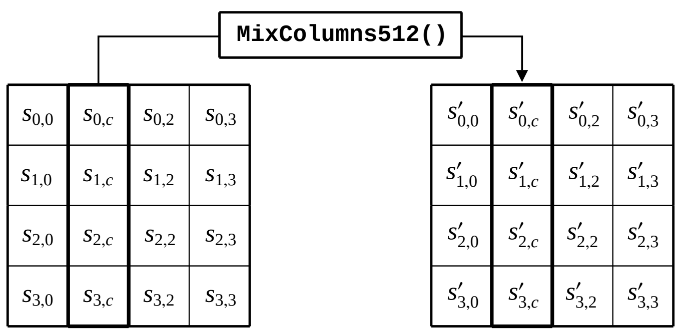

Figure 27: **`MixColumns512()`** operates on the SubState512 column by column

5.5.2.4 **`AddRoundKey512()`** Transformation

In the **`AddRoundKey512()`** transformation, the two-word Round Key $K^{(round)} = K_0^{(round)} \| K_1^{(round)}$ from the key schedule, as described in Secs. 5.5.2.6 and 5.5.2.7, is added to the SubState512 by a simple bitwise XOR operation. The two words are each added into the SubState512, such that

$$\begin{aligned} \left[s'_{0,0}, s'_{1,0}, s'_{2,0}, s'_{3,0}, s'_{0,1}, s'_{1,1}, s'_{2,1}, s'_{3,1}\right] &= [s_{0,0}, s_{1,0}, s_{2,0}, s_{3,0}, s_{0,1}, s_{1,1}, s_{2,1}, s_{3,1}] \oplus K_0^{(round)}, \\ \left[s'_{0,2}, s'_{1,2}, s'_{2,2}, s'_{3,2}, s'_{0,3}, s'_{1,3}, s'_{2,3}, s'_{3,3}\right] &= [s_{0,2}, s_{1,2}, s_{2,2}, s_{3,2}, s_{0,3}, s_{1,3}, s_{2,3}, s_{3,3}] \oplus K_1^{(round)}. \end{aligned}$$

5.5.2.5 **`WordRotation512()`**

`WordRotation512()` takes eight 64-bit words $x_0, x_1, \ldots, x_7$ as input and performs a cyclic permutation. The function proceeds as follows:

$$x'_{j+2 \bmod 8} = x_j, \quad \text{for } 0 \le j < 8.$$

5.5.2.6 **KeyExpComp512()**

During the process of **Compression512(**$H^{(i-1)}, M^{(i)}$**)**, the $EncComp_{512}$ block cipher takes the intermediate hash value $H^{(i-1)}$ as the Block Cipher Key and performs the Key Expansion routine **KeyExpComp512()** to generate a key schedule.

KeyExpComp512() generates a total of $2 * Nr\_\text{comp512}$ words: the algorithm requires an initial set of eight words, and each of the Nr_comp512 rounds requires eight words of key data. The resulting key schedule consists of a linear array of words, with i in the range of $0 \le i < 2 * Nr\_\text{comp512}$. The round constant word array $C^{(round)} = C_0^{(round)} || C_1^{(round)}$ is defined in Sec. 5.1.2. Expansion of the input key into the key schedule proceeds according to the pseudocode shown in Fig. 28.

SubWords512() is a function that takes 16-byte input words and applies the S-box (Fig. 15) to each of the 16 bytes to produce output words. **WordRotation512()** is defined in Sec. 5.5.2.5.

```
KeyExpComp512(word chain[8], word K[Nr_comp512][2])
begin
   word t[2]  /* The structure is not a SubState512 */

   for round = 0 to Nr_comp512 - 1
      t[0] = chain[4] ⊕ C[round][0]
      t[1] = chain[5] ⊕ C[round][1]

      SubWords512(t)
      KeyLinear512(t)
      ByteTranspos512(t)

      chain[6] = chain[6] ⊕ t[0]
      chain[7] = chain[7] ⊕ t[1]

      WordRotation512(chain)
      K[round][0] = chain[2]
      K[round][1] = chain[3]
   end for
end
```

Figure 28: Pseudocode for **KeyExpComp512()**

Each of the The functions **KeyLinear512()** and **ByteTranspos512()** takes 16 bytes $a_0, a_1, \ldots, a_{15}$ as input and performs a bytewise permutation. **KeyLinear512()** is a bytewise operation given by the following equation, where multiplication over GF(2^8) is defined in Sec. 4.2:

$$\begin{bmatrix} a'_i \\ a'_{i+1} \\ a'_{i+2} \\ a'_{i+3} \\ a'_{i+4} \\ a'_{i+5} \\ a'_{i+6} \\ a'_{i+7} \end{bmatrix} = \begin{bmatrix} \texttt{01} & \texttt{01} & \texttt{02} & \texttt{0a} & \texttt{09} & \texttt{08} & \texttt{01} & \texttt{04} \\ \texttt{04} & \texttt{01} & \texttt{01} & \texttt{02} & \texttt{0a} & \texttt{09} & \texttt{08} & \texttt{01} \\ \texttt{01} & \texttt{04} & \texttt{01} & \texttt{01} & \texttt{02} & \texttt{0a} & \texttt{09} & \texttt{08} \\ \texttt{08} & \texttt{01} & \texttt{04} & \texttt{01} & \texttt{01} & \texttt{02} & \texttt{0a} & \texttt{09} \\ \texttt{09} & \texttt{08} & \texttt{01} & \texttt{04} & \texttt{01} & \texttt{01} & \texttt{02} & \texttt{0a} \\ \texttt{0a} & \texttt{09} & \texttt{08} & \texttt{01} & \texttt{04} & \texttt{01} & \texttt{01} & \texttt{02} \\ \texttt{02} & \texttt{0a} & \texttt{09} & \texttt{08} & \texttt{01} & \texttt{04} & \texttt{01} & \texttt{01} \\ \texttt{01} & \texttt{02} & \texttt{0a} & \texttt{09} & \texttt{08} & \texttt{01} & \texttt{04} & \texttt{01} \end{bmatrix} \begin{bmatrix} a_i \\ a_{i+1} \\ a_{i+2} \\ a_{i+3} \\ a_{i+4} \\ a_{i+5} \\ a_{i+6} \\ a_{i+7} \end{bmatrix}, \quad i = 0, 8.$$

$$\begin{aligned}
a'_i &= a_i \oplus a_{i+1} \oplus (\{\texttt{02}\} \bullet a_{i+2}) \oplus (\{\texttt{0a}\} \bullet a_{i+3}) \oplus (\{\texttt{09}\} \bullet a_{i+4}) \oplus (\{\texttt{08}\} \bullet a_{i+5}) \oplus a_{i+6} \oplus (\{\texttt{04}\} \bullet a_{i+7}), \\
a'_{i+1} &= (\{\texttt{04}\} \bullet a_i) \oplus a_{i+1} \oplus a_{i+2} \oplus (\{\texttt{02}\} \bullet a_{i+3}) \oplus (\{\texttt{0a}\} \bullet a_{i+4}) \oplus (\{\texttt{09}\} \bullet a_{i+5}) \oplus (\{\texttt{08}\} \bullet a_{i+6}) \oplus a_{i+7}, \\
a'_{i+2} &= a_i \oplus (\{\texttt{04}\} \bullet a_{i+1}) \oplus a_{i+2} \oplus a_{i+3} \oplus (\{\texttt{02}\} \bullet a_{i+4}) \oplus (\{\texttt{0a}\} \bullet a_{i+5}) \oplus (\{\texttt{09}\} \bullet a_{i+6}) \oplus (\{\texttt{08}\} \bullet a_{i+7}), \\
a'_{i+3} &= (\{\texttt{08}\} \bullet a_i) \oplus a_{i+1} \oplus (\{\texttt{04}\} \bullet a_{i+2}) \oplus a_{i+3} \oplus a_{i+4} \oplus (\{\texttt{02}\} \bullet a_{i+5}) \oplus (\{\texttt{0a}\} \bullet a_{i+6}) \oplus (\{\texttt{09}\} \bullet a_{i+7}), \\
a'_{i+4} &= (\{\texttt{09}\} \bullet a_i) \oplus (\{\texttt{08}\} \bullet a_{i+1}) \oplus a_{i+2} \oplus (\{\texttt{04}\} \bullet a_{i+3}) \oplus a_{i+4} \oplus a_{i+5} \oplus (\{\texttt{02}\} \bullet a_{i+6}) \oplus (\{\texttt{0a}\} \bullet a_{i+7}), \\
a'_{i+5} &= (\{\texttt{0a}\} \bullet a_i) \oplus (\{\texttt{09}\} \bullet a_{i+1}) \oplus (\{\texttt{08}\} \bullet a_{i+2}) \oplus a_{i+3} \oplus (\{\texttt{04}\} \bullet a_{i+4}) \oplus a_{i+5} \oplus a_{i+6} \oplus (\{\texttt{02}\} \bullet a_{i+7}), \\
a'_{i+6} &= (\{\texttt{02}\} \bullet a_i) \oplus (\{\texttt{0a}\} \bullet a_{i+1}) \oplus (\{\texttt{09}\} \bullet a_{i+2}) \oplus (\{\texttt{08}\} \bullet a_{i+3}) \oplus a_{i+4} \oplus (\{\texttt{04}\} \bullet a_{i+5}) \oplus a_{i+6} \oplus a_{i+7}, \\
a'_{i+7} &= a_i \oplus (\{\texttt{02}\} \bullet a_{i+1}) \oplus (\{\texttt{0a}\} \bullet a_{i+2}) \oplus (\{\texttt{09}\} \bullet a_{i+3}) \oplus (\{\texttt{08}\} \bullet a_{i+4}) \oplus a_{i+5} \oplus (\{\texttt{04}\} \bullet a_{i+6}) \oplus a_{i+7}.
\end{aligned}$$

`ByteTranspos512()` performs bytewise transposition in the following manner:

$$\begin{array}{llllllll}
a'_0 = a_8, & a'_1 = a_9, & a'_2 = a_{10}, & a'_3 = a_{11}, & a'_4 = a_4, & a'_5 = a_5, & a'_6 = a_6, & a'_7 = a_7, \\
a'_8 = a_0, & a'_9 = a_1, & a'_{10} = a_2, & a'_{11} = a_3, & a'_{12} = a_{12}, & a'_{13} = a_{13}, & a'_{14} = a_{14}, & a'_{15} = a_{15}.
\end{array}$$

Figure 29 illustrates the **`ByteTranspos512()`** transformation.

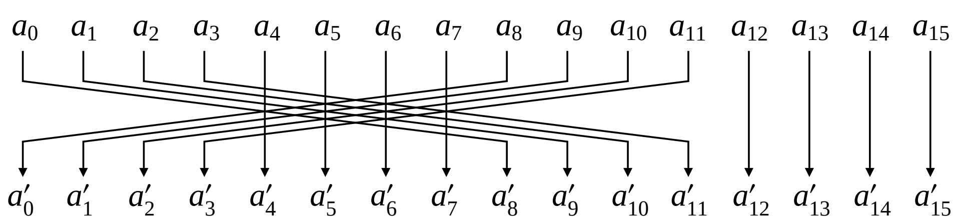

Figure 29: **`ByteTranspos512() transformation`**

5.5.2.7 **`KeyExpOut512()`**

During the process of **`Output512(`**$H^{(N-1)}, M^{(N)}$**`)`**, the $EncOut_{512}$ block cipher takes the intermediate hash value $H^{(N-1)}$ as the Block Cipher Key and performs the Key Expansion routine **`KeyExpOut512()`** to generate a key schedule.

`KeyExpOut512()` generates a total of $2 * Nr\_\mathrm{out512}$ words: the algorithm requires an initial set of eight words, and each of the Nr_out512 rounds requires eight words of key data. The resulting key schedule consists of a linear array of words, with i in the range of $0 \le i < 2 * Nr\_\mathrm{out512}$. The round constant word array $C^{(round)} = C_0^{(round)} \| C_1^{(round)}$ is defined in Sec. 5.1.2.

Expansion of the input key into the key schedule proceeds according to the pseudocode shown in Fig. 30.

The functions **SubBytes512()**, **ShiftRows512()**, **MixColumns512()**, and **WordRotation512()** are defined in Secs. 5.5.2.1, 5.5.2.2, 5.5.2.3, and 5.5.2.5, respectively.

```
KeyExpOut512(word chain[8], word K[Nr_out512][2])
begin
   word substate512[2]

   for round = 0 to Nr_out512 - 1
      substate512[0] = chain[4] ⊕ C[round][0]
      substate512[1] = chain[5] ⊕ C[round][1]

      for iteration = 0 to 3
         SubBytes512(substate512)
         ShiftRows512(substate512)
         MixColumns512(substate512)
      end for

      chain[6] = chain[6] ⊕ substate512[0]
      chain[7] = chain[7] ⊕ substate512[1]

      WordRotation512(chain)
      K[round][0] = chain[2]
      K[round][1] = chain[3]
   end for
end
```

Figure 30: Pseudocode for **KeyExpOut512()**

5.6 Lesamnta-384 Algorithm

Lesamnta-384 can be used to hash a message M having a length of l bits, where $0 \leq l < 2^{128}$. The algorithm is defined in exactly the same manner as for Lesamnta-512 (Sec. 5.5), with the following two exceptions:

1. The initial hash value $H^{(0)}$ is set as specified in Sec. 5.2.3.3.
2. The 384-bit message digest is obtained by truncating the final hash value $H^{(N)}$ to its leftmost 384 bits:
$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)}.$$

5.7 Lesamnta Examples

5.7.1 Lesamnta-256 Example

Let the message M, be the 24-bit ($l = 24$) ASCII string "**abc**", which is equivalent to the following binary string:

$$\texttt{01100001 01100010 01100011}.$$

The message is padded by appending a "1" bit, followed by 423 "0" bits, and ending with the hex value 00000000 00000018 (the two 32-bit word representation of length 24). Thus, the final padded message consists of two blocks ($N = 2$).

For Lesamnta-256, the initial hash value $H^{(0)}$ is

$$
\begin{aligned}
H_0^{(0)} &= \texttt{00000256},\\
H_1^{(0)} &= \texttt{00000256},\\
H_2^{(0)} &= \texttt{00000256},\\
H_3^{(0)} &= \texttt{00000256},\\
H_4^{(0)} &= \texttt{00000256},\\
H_5^{(0)} &= \texttt{00000256},\\
H_6^{(0)} &= \texttt{00000256},\\
H_7^{(0)} &= \texttt{00000256}.
\end{aligned}
$$

The words of the padded message block $M^{(1)}$ are then assigned to the words $x_0, ..., x_7$ of the block cipher $EncComp_{256}$:

$$
\begin{aligned}
x_0 &= \texttt{61626380},\\
x_1 &= \texttt{00000000},\\
x_2 &= \texttt{00000000},\\
x_3 &= \texttt{00000000},\\
x_4 &= \texttt{00000000},\\
x_5 &= \texttt{00000000},\\
x_6 &= \texttt{00000000},\\
x_7 &= \texttt{00000000}.
\end{aligned}
$$

The following schedule shows the hex values for $x_0, ..., x_7$, **after** round r of the "for $r = 0$ to 31" loop described in Sec. 5.3.2, Figure 11, step 2.

	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$r = 0$:	924bde4c	924bde4c	61626380	00000000	00000000	00000000	00000000	00000000
$r = 1$:	271b6be7	2b583bdb	924bde4c	924bde4c	61626380	00000000	00000000	00000000
$r = 2$:	9a5f8551	08e5acca	271b6be7	2b583bdb	924bde4c	924bde4c	61626380	00000000
$r = 3$:	318ce5af	b7a8215b	9a5f8551	08e5acca	271b6be7	2b583bdb	924bde4c	924bde4c
$r = 4$:	15e5553b	e26a5218	318ce5af	b7a8215b	9a5f8551	08e5acca	271b6be7	2b583bdb
$r = 5$:	a7932650	8835a31c	15e5553b	e26a5218	318ce5af	b7a8215b	9a5f8551	08e5acca
$r = 6$:	64926b7a	1af443fc	a7932650	8835a31c	15e5553b	e26a5218	318ce5af	b7a8215b
$r = 7$:	f58103a1	c4a7b9f7	64926b7a	1af443fc	a7932650	8835a31c	15e5553b	e26a5218
$r = 8$:	d6e2e3c3	5efe05de	f58103a1	c4a7b9f7	64926b7a	1af443fc	a7932650	8835a31c
$r = 9$:	e93f5fcc	c44e4e6e	d6e2e3c3	5efe05de	f58103a1	c4a7b9f7	64926b7a	1af443fc
$r = 10$:	62e5737e	a701ecd7	e93f5fcc	c44e4e6e	d6e2e3c3	5efe05de	f58103a1	c4a7b9f7
$r = 11$:	7efb3e71	14433399	62e5737e	a701ecd7	e93f5fcc	c44e4e6e	d6e2e3c3	5efe05de
$r = 12$:	584202c0	871a2fd7	7efb3e71	14433399	62e5737e	a701ecd7	e93f5fcc	c44e4e6e
$r = 13$:	09e5d4b9	7f476927	584202c0	871a2fd7	7efb3e71	14433399	62e5737e	a701ecd7
$r = 14$:	3f75d6b1	82df6e25	09e5d4b9	7f476927	584202c0	871a2fd7	7efb3e71	14433399
$r = 15$:	167f4af9	36ec1fdc	3f75d6b1	82df6e25	09e5d4b9	7f476927	584202c0	871a2fd7
$r = 16$:	0b6d0af1	d8a4ed39	167f4af9	36ec1fdc	3f75d6b1	82df6e25	09e5d4b9	7f476927
$r = 17$:	bbc87f9b	33e64080	0b6d0af1	d8a4ed39	167f4af9	36ec1fdc	3f75d6b1	82df6e25
$r = 18$:	344a8de9	1122a932	bbc87f9b	33e64080	0b6d0af1	d8a4ed39	167f4af9	36ec1fdc
$r = 19$:	4cfba3a0	519dbe2b	344a8de9	1122a932	bbc87f9b	33e64080	0b6d0af1	d8a4ed39
$r = 20$:	40b51e54	df911e26	4cfba3a0	519dbe2b	344a8de9	1122a932	bbc87f9b	33e64080
$r = 21$:	e45b2b33	dfb34ce6	40b51e54	df911e26	4cfba3a0	519dbe2b	344a8de9	1122a932
$r = 22$:	859cd55a	080884eb	e45b2b33	dfb34ce6	40b51e54	df911e26	4cfba3a0	519dbe2b
$r = 23$:	cafc90b6	ef086cdc	859cd55a	080884eb	e45b2b33	dfb34ce6	40b51e54	df911e26
$r = 24$:	4c31690a	3c726b86	cafc90b6	ef086cdc	859cd55a	080884eb	e45b2b33	dfb34ce6
$r = 25$:	340b67eb	7cb138bd	4c31690a	3c726b86	cafc90b6	ef086cdc	859cd55a	080884eb
$r = 26$:	a3dac1c1	f7fa6162	340b67eb	7cb138bd	4c31690a	3c726b86	cafc90b6	ef086cdc
$r = 27$:	a8cfafa7	3d5d14b1	a3dac1c1	f7fa6162	340b67eb	7cb138bd	4c31690a	3c726b86
$r = 28$:	d3de8d3d	133083c0	a8cfafa7	3d5d14b1	a3dac1c1	f7fa6162	340b67eb	7cb138bd
$r = 29$:	a8321805	e1b21118	d3de8d3d	133083c0	a8cfafa7	3d5d14b1	a3dac1c1	f7fa6162
$r = 30$:	0b9e1b3f	68db00ac	a8321805	e1b21118	d3de8d3d	133083c0	a8cfafa7	3d5d14b1
$r = 31$:	a5fced96	897331ee	0b9e1b3f	68db00ac	a8321805	e1b21118	d3de8d3d	133083c0

That completes the processing of the **first** message block $M^{(1)}$. The intermediate hash value $H^{(1)}$ is calculated to be

$$
\begin{aligned}
H_0^{(1)} &= \texttt{a5fced96} \oplus \texttt{61626380} = \texttt{c49e8e16}, \\
H_1^{(1)} &= \texttt{897331ee} \oplus \texttt{00000000} = \texttt{897331ee}, \\
H_2^{(1)} &= \texttt{0b9e1b3f} \oplus \texttt{00000000} = \texttt{0b9e1b3f}, \\
H_3^{(1)} &= \texttt{68db00ac} \oplus \texttt{00000000} = \texttt{68db00ac}, \\
H_4^{(1)} &= \texttt{a8321805} \oplus \texttt{00000000} = \texttt{a8321805}, \\
H_5^{(1)} &= \texttt{e1b21118} \oplus \texttt{00000000} = \texttt{e1b21118}, \\
H_6^{(1)} &= \texttt{d3de8d3d} \oplus \texttt{00000000} = \texttt{d3de8d3d}, \\
H_7^{(1)} &= \texttt{133083c0} \oplus \texttt{00000000} = \texttt{133083c0}.
\end{aligned}
$$

The words of the **second** padded message block $M^{(2)}$ are then assigned to the words $x_0, ..., x_7$ of the block cipher $EncOut_{256}$:

$$\begin{aligned} x_0 &= \texttt{00000000},\\ x_1 &= \texttt{00000000},\\ x_2 &= \texttt{00000000},\\ x_3 &= \texttt{00000000},\\ x_4 &= \texttt{00000000},\\ x_5 &= \texttt{00000000},\\ x_6 &= \texttt{00000000},\\ x_7 &= \texttt{00000018}. \end{aligned}$$

The following schedule shows the hex values for $x_0, ..., x_7$, **after** round r of the "for $r = 0$ to 31" loop described in Sec. 5.3.2, Figure 13, step 2.

	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
$r = 0$:	7db22819	7b84aff3	00000000	00000000	00000000	00000000	00000000	00000000
$r = 1$:	2cb35079	2f2327fe	7db22819	7b84aff3	00000000	00000000	00000000	00000000
$r = 2$:	0886491b	bdf6a9bd	2cb35079	2f2327fe	7db22819	7b84aff3	00000000	00000000
$r = 3$:	21bfbf59	b854bc30	0886491b	bdf6a9bd	2cb35079	2f2327fe	7db22819	7b84aff3
$r = 4$:	f1c77947	40b67b9e	21bfbf59	b854bc30	0886491b	bdf6a9bd	2cb35079	2f2327fe
$r = 5$:	23a05bc2	4c0b325e	f1c77947	40b67b9e	21bfbf59	b854bc30	0886491b	bdf6a9bd
$r = 6$:	8a7c7c87	c8461974	23a05bc2	4c0b325e	f1c77947	40b67b9e	21bfbf59	b854bc30
$r = 7$:	2e8e1d78	b05f0c02	8a7c7c87	c8461974	23a05bc2	4c0b325e	f1c77947	40b67b9e
$r = 8$:	b391c5ee	aa7d210b	2e8e1d78	b05f0c02	8a7c7c87	c8461974	23a05bc2	4c0b325e
$r = 9$:	08b40481	ff1e4869	b391c5ee	aa7d210b	2e8e1d78	b05f0c02	8a7c7c87	c8461974
$r = 10$:	a420e8ec	80c14ce5	08b40481	ff1e4869	b391c5ee	aa7d210b	2e8e1d78	b05f0c02
$r = 11$:	406ac0a0	8a0e1380	a420e8ec	80c14ce5	08b40481	ff1e4869	b391c5ee	aa7d210b
$r = 12$:	5f625ef3	6a58a031	406ac0a0	8a0e1380	a420e8ec	80c14ce5	08b40481	ff1e4869
$r = 13$:	634a9d62	9ef7610d	5f625ef3	6a58a031	406ac0a0	8a0e1380	a420e8ec	80c14ce5
$r = 14$:	415dd8a0	35c1dac8	634a9d62	9ef7610d	5f625ef3	6a58a031	406ac0a0	8a0e1380
$r = 15$:	27e6d188	7c2c5b8f	415dd8a0	35c1dac8	634a9d62	9ef7610d	5f625ef3	6a58a031
$r = 16$:	86badf0b	b654454a	27e6d188	7c2c5b8f	415dd8a0	35c1dac8	634a9d62	9ef7610d
$r = 17$:	bfa35647	a9015eb9	86badf0b	b654454a	27e6d188	7c2c5b8f	415dd8a0	35c1dac8
$r = 18$:	9c7c8895	1aef2bc9	bfa35647	a9015eb9	86badf0b	b654454a	27e6d188	7c2c5b8f
$r = 19$:	42c06cc6	8907bb96	9c7c8895	1aef2bc9	bfa35647	a9015eb9	86badf0b	b654454a
$r = 20$:	45f14bf9	18051660	42c06cc6	8907bb96	9c7c8895	1aef2bc9	bfa35647	a9015eb9
$r = 21$:	1ce7ffb4	a9a9e70d	45f14bf9	18051660	42c06cc6	8907bb96	9c7c8895	1aef2bc9
$r = 22$:	8414fcd9	51b7246c	1ce7ffb4	a9a9e70d	45f14bf9	18051660	42c06cc6	8907bb96
$r = 23$:	75f94fc0	d2589717	8414fcd9	51b7246c	1ce7ffb4	a9a9e70d	45f14bf9	18051660
$r = 24$:	c8e89f1b	8bf7ebf6	75f94fc0	d2589717	8414fcd9	51b7246c	1ce7ffb4	a9a9e70d
$r = 25$:	a1d7681e	3cbe9910	c8e89f1b	8bf7ebf6	75f94fc0	d2589717	8414fcd9	51b7246c
$r = 26$:	5fd41059	a4d991ee	a1d7681e	3cbe9910	c8e89f1b	8bf7ebf6	75f94fc0	d2589717
$r = 27$:	8373c6c6	8ba99026	5fd41059	a4d991ee	a1d7681e	3cbe9910	c8e89f1b	8bf7ebf6
$r = 28$:	d366ec57	4407852b	8373c6c6	8ba99026	5fd41059	a4d991ee	a1d7681e	3cbe9910
$r = 29$:	ae6cf0c9	47d9aeff	d366ec57	4407852b	8373c6c6	8ba99026	5fd41059	a4d991ee
$r = 30$:	ca26c0c9	ac23a7af	ae6cf0c9	47d9aeff	d366ec57	4407852b	8373c6c6	8ba99026
$r = 31$:	36936338	78299c69	ca26c0c9	ac23a7af	ae6cf0c9	47d9aeff	d366ec57	4407852b

That completes the processing of the second and final message block $M^{(2)}$. The final hash value $H^{(2)}$ is calculated to be

$$\begin{aligned}
H_0^{(2)} &= \texttt{36936338} \oplus \texttt{00000000} = \texttt{36936338},\\
H_1^{(2)} &= \texttt{78299c69} \oplus \texttt{00000000} = \texttt{78299c69},\\
H_2^{(2)} &= \texttt{ca26c0c9} \oplus \texttt{00000000} = \texttt{ca26c0c9},\\
H_3^{(2)} &= \texttt{ac23a7af} \oplus \texttt{00000000} = \texttt{ac23a7af},\\
H_4^{(2)} &= \texttt{ae6cf0c9} \oplus \texttt{00000000} = \texttt{ae6cf0c9},\\
H_5^{(2)} &= \texttt{47d9aeff} \oplus \texttt{00000000} = \texttt{47d9aeff},\\
H_6^{(2)} &= \texttt{d366ec57} \oplus \texttt{00000000} = \texttt{d366ec57},\\
H_7^{(2)} &= \texttt{4407852b} \oplus \texttt{00000018} = \texttt{44078533}.
\end{aligned}$$

The resulting 256-bit message digest is

`36936338 78299c69 ca26c0c9 ac23a7af ae6cf0c9 47d9aeff d366ec57 44078533`.

5.7.2 Lesamnta-512 Example

Let the message M be the 24-bit ($l = 24$) ASCII string "**abc**", which is equivalent to the following binary string:

`01100001 01100010 01100011`.

The message is padded by appending a "1" bit, followed by 871 "0" bits, and ending with the hex value `0000000000000000 0000000000000018` (the two 64-bit word representation of length 24). Thus, the final padded message consists of two blocks ($N = 2$).

For Lesamnta-512, the initial hash value $H^{(0)}$ is

$$\begin{aligned}
H_0^{(0)} &= \texttt{0000000000000512},\\
H_1^{(0)} &= \texttt{0000000000000512},\\
H_2^{(0)} &= \texttt{0000000000000512},\\
H_3^{(0)} &= \texttt{0000000000000512},\\
H_4^{(0)} &= \texttt{0000000000000512},\\
H_5^{(0)} &= \texttt{0000000000000512},\\
H_6^{(0)} &= \texttt{0000000000000512},\\
H_7^{(0)} &= \texttt{0000000000000512}.
\end{aligned}$$

The words of the padded message block $M^{(1)}$ are then assigned to the words $x_0, ..., x_7$ of the block cipher $EncComp_{512}$:

$$\begin{aligned}
x_0 &= \texttt{6162638000000000},\\
x_1 &= \texttt{0000000000000000},\\
x_2 &= \texttt{0000000000000000},\\
x_3 &= \texttt{0000000000000000},\\
x_4 &= \texttt{0000000000000000},\\
x_5 &= \texttt{0000000000000000},\\
x_6 &= \texttt{0000000000000000},\\
x_7 &= \texttt{0000000000000000}.
\end{aligned}$$

The following schedule shows the hex values for $x_0, ..., x_7$, **after** round r of the "for $r = 0$ to 31" loop described in Sec. 5.5.2, Figure 22, step 2.

	x_0/x_4	x_1/x_5	x_2/x_6	x_3/x_7
$r = 0$:	230d5e40851cb824	230d5e40851cb824	6162638000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
$r = 1$:	bb27b99ec31efd17	648097e5093a10e8	230d5e40851cb824	230d5e40851cb824
	6162638000000000	0000000000000000	0000000000000000	0000000000000000
$r = 2$:	6612e1d8b6e40600	32851c3f32409f9f	bb27b99ec31efd17	648097e5093a10e8
	230d5e40851cb824	230d5e40851cb824	6162638000000000	0000000000000000
$r = 3$:	fb75bbde6c95c571	04131e4ec79b2add	6612e1d8b6e40600	32851c3f32409f9f
	bb27b99ec31efd17	648097e5093a10e8	230d5e40851cb824	230d5e40851cb824
$r = 4$:	cb0cfe8fae16735e	2b075e87a69cc50e	fb75bbde6c95c571	04131e4ec79b2add
	6612e1d8b6e40600	32851c3f32409f9f	bb27b99ec31efd17	648097e5093a10e8
$r = 5$:	6fcb2839c4c9a227	da92ab977e57abbc	cb0cfe8fae16735e	2b075e87a69cc50e
	fb75bbde6c95c571	04131e4ec79b2add	6612e1d8b6e40600	32851c3f32409f9f
$r = 6$:	a4f0de3f7d0c4336	8a64ab6504493a96	6fcb2839c4c9a227	da92ab977e57abbc
	cb0cfe8fae16735e	2b075e87a69cc50e	fb75bbde6c95c571	04131e4ec79b2add
$r = 7$:	2d375a2eabab1fb7	9d423a20138e5bfc	a4f0de3f7d0c4336	8a64ab6504493a96
	6fcb2839c4c9a227	da92ab977e57abbc	cb0cfe8fae16735e	2b075e87a69cc50e
$r = 8$:	91f43770e29ae13f	d11012d112c24993	2d375a2eabab1fb7	9d423a20138e5bfc
	a4f0de3f7d0c4336	8a64ab6504493a96	6fcb2839c4c9a227	da92ab977e57abbc
$r = 9$:	6f78095ab7e7710a	2b65442db2afafcf	91f43770e29ae13f	d11012d112c24993
	2d375a2eabab1fb7	9d423a20138e5bfc	a4f0de3f7d0c4336	8a64ab6504493a96
$r = 10$:	b015b34805866e5c	def53ced7729fc16	6f78095ab7e7710a	2b65442db2afafcf
	91f43770e29ae13f	d11012d112c24993	2d375a2eabab1fb7	9d423a20138e5bfc
$r = 11$:	352afb43790c6555	245a789c29dd333e	b015b34805866e5c	def53ced7729fc16
	6f78095ab7e7710a	2b65442db2afafcf	91f43770e29ae13f	d11012d112c24993
$r = 12$:	73ed27e5fa7e3a85	77d6013bfe2ab57c	352afb43790c6555	245a789c29dd333e
	b015b34805866e5c	def53ced7729fc16	6f78095ab7e7710a	2b65442db2afafcf
$r = 13$:	c050e54f26a2d76c	e6d6f285cac7a8b8	73ed27e5fa7e3a85	77d6013bfe2ab57c
	352afb43790c6555	245a789c29dd333e	b015b34805866e5c	def53ced7729fc16
$r = 14$:	8c23abef0c1f1892	2207010d00310d9e	c050e54f26a2d76c	e6d6f285cac7a8b8
	73ed27e5fa7e3a85	77d6013bfe2ab57c	352afb43790c6555	245a789c29dd333e
$r = 15$:	ab21c2e457cd9134	fd091afc000cb7ec	8c23abef0c1f1892	2207010d00310d9e
	c050e54f26a2d76c	e6d6f285cac7a8b8	73ed27e5fa7e3a85	77d6013bfe2ab57c

$r = 16$:	fff52589b44e3be5	c0160d12659abe10	ab21c2e457cd9134	fd091afc000cb7ec
	8c23abef0c1f1892	2207010d00310d9e	c050e54f26a2d76c	e6d6f285cac7a8b8
$r = 17$:	8c27f5ce9e2ce604	43b106446c171dd0	fff52589b44e3be5	c0160d12659abe10
	ab21c2e457cd9134	fd091afc000cb7ec	8c23abef0c1f1892	2207010d00310d9e
$r = 18$:	12b77e2e7cf6684d	ac5eb7afbd6a2bf7	8c27f5ce9e2ce604	43b106446c171dd0
	fff52589b44e3be5	c0160d12659abe10	ab21c2e457cd9134	fd091afc000cb7ec
$r = 19$:	bd88e91fbfb40826	c3ffdde8c288de20	12b77e2e7cf6684d	ac5eb7afbd6a2bf7
	8c27f5ce9e2ce604	43b106446c171dd0	fff52589b44e3be5	c0160d12659abe10
$r = 20$:	e133d378b46baa78	373236579c0bebc7	bd88e91fbfb40826	c3ffdde8c288de20
	12b77e2e7cf6684d	ac5eb7afbd6a2bf7	8c27f5ce9e2ce604	43b106446c171dd0
$r = 21$:	a8c43cbd33bdd476	cd67e506633b8775	e133d378b46baa78	373236579c0bebc7
	bd88e91fbfb40826	c3ffdde8c288de20	12b77e2e7cf6684d	ac5eb7afbd6a2bf7
$r = 22$:	2881837893fb5d4c	e2cabe5977a080be	a8c43cbd33bdd476	cd67e506633b8775
	e133d378b46baa78	373236579c0bebc7	bd88e91fbfb40826	c3ffdde8c288de20
$r = 23$:	7409957b1ff2a49b	0d7ec50153a4c843	2881837893fb5d4c	e2cabe5977a080be
	a8c43cbd33bdd476	cd67e506633b8775	e133d378b46baa78	373236579c0bebc7
$r = 24$:	09dee13209daf22d	77c8a8106f844467	7409957b1ff2a49b	0d7ec50153a4c843
	2881837893fb5d4c	e2cabe5977a080be	a8c43cbd33bdd476	cd67e506633b8775
$r = 25$:	1e7a8da467fe41b2	cb9135c1f1e31e2b	09dee13209daf22d	77c8a8106f844467
	7409957b1ff2a49b	0d7ec50153a4c843	2881837893fb5d4c	e2cabe5977a080be
$r = 26$:	1a8bc5e7f3c751ba	1296cc83c92683ae	1e7a8da467fe41b2	cb9135c1f1e31e2b
	09dee13209daf22d	77c8a8106f844467	7409957b1ff2a49b	0d7ec50153a4c843
$r = 27$:	beb513de6ac4513e	4837fc7fe45b2fc3	1a8bc5e7f3c751ba	1296cc83c92683ae
	1e7a8da467fe41b2	cb9135c1f1e31e2b	09dee13209daf22d	77c8a8106f844467
$r = 28$:	515adc58554c68d2	08cd3bb067a2b546	beb513de6ac4513e	4837fc7fe45b2fc3
	1a8bc5e7f3c751ba	1296cc83c92683ae	1e7a8da467fe41b2	cb9135c1f1e31e2b
$r = 29$:	5cbd07b2788db208	12d63beeeafbed6c	515adc58554c68d2	08cd3bb067a2b546
	beb513de6ac4513e	4837fc7fe45b2fc3	1a8bc5e7f3c751ba	1296cc83c92683ae
$r = 30$:	3f8622891a4fda5e	4dee38cb466d4328	5cbd07b2788db208	12d63beeeafbed6c
	515adc58554c68d2	08cd3bb067a2b546	beb513de6ac4513e	4837fc7fe45b2fc3
$r = 31$:	5f1d8da5cf51d123	2edc631fd504b5c4	3f8622891a4fda5e	4dee38cb466d4328
	5cbd07b2788db208	12d63beeeafbed6c	515adc58554c68d2	08cd3bb067a2b546

That completes the processing of the **first** message block $M^{(1)}$. The intermediate hash value $H^{(1)}$ is calculated to be

$$H_0^{(0)} = \texttt{5f1d8da5cf51d123} \oplus 6162638000000000 = \texttt{3e7fee25cf51d123}$$
$$H_1^{(0)} = \texttt{2edc631fd504b5c4} \oplus 0000000000000000 = \texttt{2edc631fd504b5c4}$$
$$H_2^{(0)} = \texttt{3f8622891a4fda5e} \oplus 0000000000000000 = \texttt{3f8622891a4fda5e}$$
$$H_3^{(0)} = \texttt{4dee38cb466d4328} \oplus 0000000000000000 = \texttt{4dee38cb466d4328}$$
$$H_4^{(0)} = \texttt{5cbd07b2788db208} \oplus 0000000000000000 = \texttt{5cbd07b2788db208}$$
$$H_5^{(0)} = \texttt{12d63beeeafbed6c} \oplus 0000000000000000 = \texttt{12d63beeeafbed6c}$$
$$H_6^{(0)} = \texttt{515adc58554c68d2} \oplus 0000000000000000 = \texttt{515adc58554c68d2}$$
$$H_7^{(0)} = \texttt{08cd3bb067a2b546} \oplus 0000000000000000 = \texttt{08cd3bb067a2b546}$$

The words of the **second** padded message block $M^{(2)}$ are then assigned to the words $x_0, ..., x_7$ of the block cipher $EncOut_{512}$:

$$
\begin{aligned}
x_0 &= 0000000000000000,\\
x_1 &= 0000000000000000,\\
x_2 &= 0000000000000000,\\
x_3 &= 0000000000000000,\\
x_4 &= 0000000000000000,\\
x_5 &= 0000000000000000,\\
x_6 &= 0000000000000000,\\
x_7 &= 0000000000000018.
\end{aligned}
$$

The following schedule shows the hex values for $x_0, \ldots, x_7$, **after** round r of the "for $r = 0$ to 31" loop described in Sec. 5.5.2, Figure 24, step 2.

	x_0/x_4	x_1/x_5	x_2/x_6	x_3/x_7
$r = 0$:	d97eb976b5cae7b2	f6e54f8f9f2f838c	0000000000000000	0000000000000000
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
$r = 1$:	1bb657b228019226	eeccd8d36781fe4a	d97eb976b5cae7b2	f6e54f8f9f2f838c
	0000000000000000	0000000000000000	0000000000000000	0000000000000000
$r = 2$:	fb6c651cb07f0756	a4eafa7e37812406	1bb657b228019226	eeccd8d36781fe4a
	d97eb976b5cae7b2	f6e54f8f9f2f838c	0000000000000000	0000000000000000
$r = 3$:	a54cc7495c328d80	11cd3d4dbfbd126f	fb6c651cb07f0756	a4eafa7e37812406
	1bb657b228019226	eeccd8d36781fe4a	d97eb976b5cae7b2	f6e54f8f9f2f838c
$r = 4$:	4c33a91a8f0df69d	42cfd1a98b14a699	a54cc7495c328d80	11cd3d4dbfbd126f
	fb6c651cb07f0756	a4eafa7e37812406	1bb657b228019226	eeccd8d36781fe4a
$r = 5$:	a7a8282b6e3c3bb3	87a6d999479b1222	4c33a91a8f0df69d	42cfd1a98b14a699
	a54cc7495c328d80	11cd3d4dbfbd126f	fb6c651cb07f0756	a4eafa7e37812406
$r = 6$:	07e6dcc7565cb26c	13201c3510519a92	a7a8282b6e3c3bb3	87a6d999479b1222
	4c33a91a8f0df69d	42cfd1a98b14a699	a54cc7495c328d80	11cd3d4dbfbd126f
$r = 7$:	20915656a888c4e2	abd14e2c830859b9	07e6dcc7565cb26c	13201c3510519a92
	a7a8282b6e3c3bb3	87a6d999479b1222	4c33a91a8f0df69d	42cfd1a98b14a699
$r = 8$:	6575618e1f64665c	29e8dc7ae201a791	20915656a888c4e2	abd14e2c830859b9
	07e6dcc7565cb26c	13201c3510519a92	a7a8282b6e3c3bb3	87a6d999479b1222
$r = 9$:	822c1e21e65471fd	de5bf43484a52d25	6575618e1f64665c	29e8dc7ae201a791
	20915656a888c4e2	abd14e2c830859b9	07e6dcc7565cb26c	13201c3510519a92
$r = 10$:	81c44e19575d610e	d312147aea845dac	822c1e21e65471fd	de5bf43484a52d25
	6575618e1f64665c	29e8dc7ae201a791	20915656a888c4e2	abd14e2c830859b9
$r = 11$:	6da03e2875c1eb8b	e007b149234c2039	81c44e19575d610e	d312147aea845dac
	822c1e21e65471fd	de5bf43484a52d25	6575618e1f64665c	29e8dc7ae201a791
$r = 12$:	9fe7019fcc3ac5ae	bf0eb2daf37379d8	6da03e2875c1eb8b	e007b149234c2039
	81c44e19575d610e	d312147aea845dac	822c1e21e65471fd	de5bf43484a52d25
$r = 13$:	1737980bb2b545bb	a9d4b5b23da13cce	9fe7019fcc3ac5ae	bf0eb2daf37379d8
	6da03e2875c1eb8b	e007b149234c2039	81c44e19575d610e	d312147aea845dac
$r = 14$:	dc84d51d1978f12c	e080e9dfb6ca8a13	1737980bb2b545bb	a9d4b5b23da13cce
	9fe7019fcc3ac5ae	bf0eb2daf37379d8	6da03e2875c1eb8b	e007b149234c2039
$r = 15$:	1a1297a192d1db02	35e7c35321f0b6bb	dc84d51d1978f12c	e080e9dfb6ca8a13
	1737980bb2b545bb	a9d4b5b23da13cce	9fe7019fcc3ac5ae	bf0eb2daf37379d8

$r = 16:$	3e41c264f01d726d	923b6d1e72db4bba	1a1297a192d1db02	35e7c35321f0b6bb
	dc84d51d1978f12c	e080e9dfb6ca8a13	1737980bb2b545bb	a9d4b5b23da13cce
$r = 17:$	0ad1d941331b1c98	79e2862b3e66fd09	3e41c264f01d726d	923b6d1e72db4bba
	1a1297a192d1db02	35e7c35321f0b6bb	dc84d51d1978f12c	e080e9dfb6ca8a13
$r = 18:$	aaca47e2b0fe3f5a	7156642dcda2eb29	0ad1d941331b1c98	79e2862b3e66fd09
	3e41c264f01d726d	923b6d1e72db4bba	1a1297a192d1db02	35e7c35321f0b6bb
$r = 19:$	6dafc52cc1d0d547	1e36608071dac5e3	aaca47e2b0fe3f5a	7156642dcda2eb29
	0ad1d941331b1c98	79e2862b3e66fd09	3e41c264f01d726d	923b6d1e72db4bba
$r = 20:$	ec35e37d43c01678	5496d9fe8035083f	6dafc52cc1d0d547	1e36608071dac5e3
	aaca47e2b0fe3f5a	7156642dcda2eb29	0ad1d941331b1c98	79e2862b3e66fd09
$r = 21:$	389f9e00f826d720	bbbf18bfc2e461d6	ec35e37d43c01678	5496d9fe8035083f
	6dafc52cc1d0d547	1e36608071dac5e3	aaca47e2b0fe3f5a	7156642dcda2eb29
$r = 22:$	ab6f2ad05f521c37	fe4bf32629570c7e	389f9e00f826d720	bbbf18bfc2e461d6
	ec35e37d43c01678	5496d9fe8035083f	6dafc52cc1d0d547	1e36608071dac5e3
$r = 23:$	389b51e96af17430	c05eab0119af37df	ab6f2ad05f521c37	fe4bf32629570c7e
	389f9e00f826d720	bbbf18bfc2e461d6	ec35e37d43c01678	5496d9fe8035083f
$r = 24:$	218aa1db06fb8b1e	c60ccf05c24eecde	389b51e96af17430	c05eab0119af37df
	ab6f2ad05f521c37	fe4bf32629570c7e	389f9e00f826d720	bbbf18bfc2e461d6
$r = 25:$	9690419f78d28e70	5d0062be2e88926e	218aa1db06fb8b1e	c60ccf05c24eecde
	389b51e96af17430	c05eab0119af37df	ab6f2ad05f521c37	fe4bf32629570c7e
$r = 26:$	f8090120f1560a5e	cc9fc6a753650358	9690419f78d28e70	5d0062be2e88926e
	218aa1db06fb8b1e	c60ccf05c24eecde	389b51e96af17430	c05eab0119af37df
$r = 27:$	43789bad36235573	f9f1a2385da67c35	f8090120f1560a5e	cc9fc6a753650358
	9690419f78d28e70	5d0062be2e88926e	218aa1db06fb8b1e	c60ccf05c24eecde
$r = 28:$	b7e7e0d12698f72f	bfae42089b2f3fbf	43789bad36235573	f9f1a2385da67c35
	f8090120f1560a5e	cc9fc6a753650358	9690419f78d28e70	5d0062be2e88926e
$r = 29:$	e9c341998ad40243	b6783342a6634059	b7e7e0d12698f72f	bfae42089b2f3fbf
	43789bad36235573	f9f1a2385da67c35	f8090120f1560a5e	cc9fc6a753650358
$r = 30:$	1efb9c25cbcfb52c	aab3b143bf427ceb	e9c341998ad40243	b6783342a6634059
	b7e7e0d12698f72f	bfae42089b2f3fbf	43789bad36235573	f9f1a2385da67c35
$r = 31:$	81a5e646a12c0381	b119c3d7aa83da41	1efb9c25cbcfb52c	aab3b143bf427ceb
	e9c341998ad40243	b6783342a6634059	b7e7e0d12698f72f	bfae42089b2f3fbf

That completes the processing of the second and final message block $M^{(2)}$. The final hash value $H^{(2)}$ is calculated to be

$$
\begin{aligned}
H_0^{(2)} &= \texttt{81a5e646a12c0381} \oplus 0000000000000000 = \texttt{81a5e646a12c0381},\\
H_1^{(2)} &= \texttt{b119c3d7aa83da41} \oplus 0000000000000000 = \texttt{b119c3d7aa83da41},\\
H_2^{(2)} &= \texttt{1efb9c25cbcfb52c} \oplus 0000000000000000 = \texttt{1efb9c25cbcfb52c},\\
H_3^{(2)} &= \texttt{aab3b143bf427ceb} \oplus 0000000000000000 = \texttt{aab3b143bf427ceb},\\
H_4^{(2)} &= \texttt{e9c341998ad40243} \oplus 0000000000000000 = \texttt{e9c341998ad40243},\\
H_5^{(2)} &= \texttt{b6783342a6634059} \oplus 0000000000000000 = \texttt{b6783342a6634059},\\
H_6^{(2)} &= \texttt{b7e7e0d12698f72f} \oplus 0000000000000000 = \texttt{b7e7e0d12698f72f},\\
H_7^{(2)} &= \texttt{bfae42089b2f3fbf} \oplus 0000000000000018 = \texttt{bfae42089b2f3fa7}.
\end{aligned}
$$

The resulting 512-bit message digest is

```
81a5e646a12c0381 b119c3d7aa83da41 1efb9c25cbcfb52c aab3b143bf427ceb
e9c341998ad40243 b6783342a6634059 b7e7e0d12698f72f bfae42089b2f3fa7.
```

6 Performance Figures

We present some performance figures for the Lesamnta algorithms here.

6.1 Software Implementation

6.1.1 8-bit Processors

Lesamnta has been implemented in C and assembly languages for 8-bit processors.

6.1.1.1 Implementation on Atmel® AVR® ATmega8515 Processor

Lesamnta was implemented on the Atmel® AVR® ATmega8515 processor in the assembly language, using Atmel®'s AVR studio® as a development environment and simulator. The performance results are shown in Table 1.

Table 1: Execution time and memory requirements for Lesamnta on the Atmel® AVR® ATmega8515 in assembly language

Message digest size	Execution time		Memory requirements		
	Bulk speed (cycles/byte)	One-block message (cycles/message)	Constant data (bytes)	Code length (bytes)	RAM (bytes)
224	631	47312	256	1118	66
	901	69678	256	456	68
256	631	47312	256	1118	66
	901	69678	256	456	68
384	783	114031	256	2604	132
	988	147088	256	928	135
512	783	114031	256	2604	132
	988	147088	256	928	135

The second and third columns list the execution time for hashing. The former corresponds to bulk speed, that is throughput speed when hashing a long message. The latter is for the execution time to hash a 256-bit message with Lesamnta-224 or Lesamnta-256 and a 512-bit message with Lesamnta-384 or Lesamnta-512. The fourth, fifth and sixth columns list memory requirements. The fourth lists the size of constant data and the fifth lists the code length of instructions. The sixth column lists the RAM size. Since Lesamnta does not have any other algorithm than the main algorithm, which processes messages and chaining values, the algorithm setup takes no time.

Time-Memory Trade-Off All the implementations above have only an S-box table of 256 bytes. The difference of code length between the implementations comes from whether internal functions are inlined or not. Then, the time-memory tradeoff can be seen on Table 1.

6.1.1.2 Renesas® H8®/300L Processor

Lesamnta was implemented on the Renesas® H8®/300L processor in assembly and C languages, using Renesas®'s High-performance Embedded Workshop as a development environment and simulator. The performance results are shown in Tables 2 and 3.

Table 2: Execution time and memory requirements for Lesamnta on the Renesas® H8®/300L processor in assembly language

Messge digest size	Execution time		Memory requirements		
	Bulk speed (cycles/byte)	One-block message (cycles/message)	Constant data (bytes)	Code length (bytes)	RAM (bytes)
224	1526	114660	512	904	80
256	1526	114660	512	904	80

Table 3: Execution time and memory requirements for Lesamnta on the Renesas® H8®/300L processor in C language

Messge digest size	Execution time		Memory requirements		
	Bulk speed (cycles/byte)	One-block message (cycles/message)	Constant data (bytes)	Code length (bytes)	RAM (bytes)
224	5442	429232	256	1140	62
256	5442	429232	256	1140	62
384	7551	1012408	256	1712	123
512	7551	1012408	256	1712	123

In the tables, the second and third columns list the execution time for hashing. The former corresponds to bulk speed, that is throughput speed when hashing a long message. The latter is for the execution time to hash a 256-bit message with Lesamnta-224 or Lesamnta-256 and a 512-bit message with Lesamnta-384 or Lesamnta-512. The fourth, fifth and sixth columns list memory requirements. The fourth lists the size of constant data and the fifth lists the code length of instructions. The sixth column lists the stack size. Since Lesamnta does not have any other algorithm than the main algorithm, which processes messages and chaining values, the algorithm setup takes no time.

6.1.2 32-bit Processors

Here, we show some performance figures for Lesamnta on 32-bit processors.

6.1.2.1 ANSI C Implementation on NIST Reference Platform

We implemented Lesamnta in ANSI C language on the NIST Reference Platform. The NIST Reference Platform contains the Intel® Core™ 2Duo E6600 processor, Microsoft®'s VisualStudio® 2005 C++ compiler and Windows Vista® Ultimate 32-bit Edition. The platform is shown at Table 4. This implementation follows the NIST API format.

Table 4: NIST Reference Platform

Language	CPU	Memory	OS	Compiler
ANSI C	Core™2 Duo E6600 (2.4GHz)	2 GBytes	Windows Vista® Ultimate 32-bit Edition	VisualStudio®2005

Table 5 shows performance figures of the implementation. The second column lists the execution time to hash a long message, which corresponds to bulk speed. The third column lists the execution time to hash a 256-bit message for Lesamnta-224 or Lesamnta-256 and a 512-bit message for Lesamnta-384 or Lesamnta-512. The fourth column shows the size of constant data which are look-up tables, round constants and initial vectors. The size of the look-up tables dominates the value. Since Lesamnta does not have any other algorithm than the main algorithm, which processes messages and chaining values, the algorithm setup takes no time.

Note that the result for the implementation includes overhead coming from the NIST API format.

Table 5: Performance figure of implementations in ANSI C language with NIST API on the NIST Reference Platform

Message digest size	Execution time		Memory requirement
	Bulk speed (cycles/byte)	One-block message (cycles/message)	Constant data (bytes)
224	68.9	5709	8288
256	68.9	5709	8288
384	97.7	14320	12416
512	97.7	14320	12416

6.1.2.2 Assembly Implementation on Intel® Core™2 Duo E6600 Processor

Here, we show performance figures of assembly implementations of Lesamnta on the Intel® Core™2 Duo processor. The used platform is shown at Table 6.

Table 6: NIST Reference Platform

Language	CPU	Memory	OS	Compiler
Assembly	Core™2 Duo E6600 (2.4GHz)	2 GBytes	Ubuntu® Linux® 8.04 32-bit distribution	gnu as

Table 7 shows performance figures of the implementations. The second column lists the execution time to hash a long message, which corresponds to bulk speed. The third column lists the execution time to hash a 256-bit message for Lesamnta-224 or Lesamnta-256 and a 512-bit message for Lesamnta-384 or Lesamnta-512. The fourth column shows the size of constant data which are look-up tables, round constants and initial vectors. The size of the look-up tables dominates the value. The fifth column lists the code length of the instructions. The sixth column lists the size of stack. Since Lesamnta does not have any other algorithm than the main algorithm, which processes messages and chaining values, the algorithm setup takes no time.

Table 7: Performance figure of implementations in assembly language on the Intel® Core™2 Duo processor

Message digest size	Execution time		Memory requirements		
	Bulk speed (cycles/byte)	One-block message (cycles/message)	Constant data (bytes)	Code length (bytes)	Stack (bytes)
224	59.2	4750	8288	5705	84
	100.2	8383	1632	7463	84
256	59.2	4750	8288	5705	84
	100.2	8383	1632	7463	84
384	54.5	8827	20608	10944	148
	71.5	10968	9344	13549	148
512	54.5	8827	20608	10944	148
	71.5	10968	9344	13549	148

Time-Memory Tradeoff As is seen from Table 7, there is tradeoff between the speed of hashing and the size of look-up tables.

6.1.2.3 ANSI C Implementation on ARM® ARM926EJ-S™ Processor

Lesamnta was implemented on the ARM® ARM926EJ-S™ processor in ANSI C language, using ARM®'s RealView® Development Suite as a development environment and simulator. The performance results are shown in Table 8.

Table 8: Performance figure of implementations in ANSI C language with NIST API on the ARM® ARM926EJ-S™ processor

Message digest size	Execution time		Memory requirement
	Bulk speed (cycles/byte)	One-block message (cycles/message)	Constant data (bytes)
224	204.1	15978	8288
256	204.1	15978	8288
384	244.0	34020	12416
512	244.0	34020	12416

Table 8 shows performance figures of the implementation. The second column lists the execution time to hash a long message, which corresponds to bulk speed. The third column lists the execution time to hash a 256-bit message for Lesamnta-224 or Lesamnta-256 and a 512-bit message for Lesamnta-384 or Lesamnta-512. The fourth column shows the size of constant data which are look-up tables, round constants and initial vectors. The size of the look-up tables dominates the value. Since Lesamnta does not have any other algorithm than the main algorithm, which processes messages and chaining values, the algorithm setup takes no time.

6.1.3 64-bit Processor

Here, we show some performance figures for Lesamnta on a 64-bit processor.

6.1.3.1 ANSI C Implementation on NIST Reference Platform

We implemented Lesamnta in ANSI C language on the NIST Reference Platform. The NIST Reference Platform contains the Intel® Core™2 Duo 2.4GHz processor, Microsoft®'s VisualStudio® 2005 C++ compiler and Windows Vista® Ultimate 64-bit Edition. The platform is shown at Table 9. Moreover, the implementation follows the NIST API format.

Table 9: NIST 64-bit Reference Platform

Language	CPU	Memory	OS	Compiler
ANSI C	Core™2 Duo E6600 (2.4GHz)	2 GBytes	Windows Vista® 64-bit Edition	VisualStudio® 2005

Table 10 shows performance figures of the implementation. The second column lists the execution time to hash a long message, which corresponds to bulk speed. The third column lists the execution time to hash a 256-bit message for Lesamnta-224 or Lesamnta-256 and a 512-bit message for Lesamnta-384 or Lesamnta-512. The fourth column shows the size of constant data which are look-up tables, round constants and initial vectors. The size of the look-up tables dominates the value. Since Lesamnta does not have any other algorithm than the main algorithm, which processes messages and chaining values, the algorithm setup takes no time.

Note that the result for the implementation includes overhead coming from the NIST API format.

Table 10: Performance figure of implementations in ANSI C language with NIST API on the NIST 64-bit Reference Platform

Message digest size	Execution time		Memory requirement
	Bulk speed (cycles/byte)	One-block message (cycles/message)	Constant data (bytes)
224	78.4	6581	8288
256	78.4	6581	8288
384	65.4	10962	24704
512	65.4	10962	24704

6.1.3.2 Assembly Implementation on Intel® Core™2 Duo Processor

Here, we show performance figures of assembly implementations of Lesamnta on the Intel® Core™2 Duo processor. The used platform is shown at Table 11.

Table 11: 64-bit Platform used for measurement of assembly codes

Language	CPU	Memory	OS	Compiler
Assembly	Core™2 Duo E6600 (2.4GHz)	2 GBytes	Ubuntu® Linux® 8.04 64-bit distribution	gnu as

Table 12 shows performance figures of the implementations. The second column lists the execution time to hash a long message, which corresponds to bulk speed. The third column lists the execution time to hash a 256-bit message for Lesamnta-224 or Lesamnta-256 and a 512-bit message for Lesamnta-384 or Lesamnta-512. The fourth, fifth and sixth columns list memory requirements. The fourth column shows the size of constant data which are look-up tables, round constants and initial vectors. The size of the look-up tables dominates the value. The fifth column lists the code length of the instructions. The sixth column lists the size of stack. Since Lesamnta does not have any other algorithm than the main algorithm, which processes messages and chaining values, the algorithm setup takes no time.

Table 12: Performance figure of implementations in assembly language on the Intel® Core™2 Duo processor

Message digest size	Execution time		Memory requirements		
	Bulk speed (cycles/byte)	One-block message (cycles/message)	Constant data (bytes)	Code length (bytes)	Stack (bytes)
224	52.7	4318	16672	5921	88
	93.8	8151	1824	7817	80
256	52.7	4318	16672	5921	88
	93.8	8151	1824	7817	80
384	51.2	8373	24704	12326	200
	70.8	10752	9344	13948	208
512	51.2	8373	24704	12326	200
	70.8	10752	9344	13948	208

Time-Memory Tradeoff As is seen from Table 12, there is tradeoff between the speed of hashing and the size of look-up tables.

6.2 Hardware

6.2.1 ASIC Implementation

We made estimations for speed and gate count of several different hardware architectures of Lesamnta. These estimates are based on existing 90 nm CMOS standard cell library. A gate is a two-input NAND equivalent. The results are shown in Table 13.

Table 13: ASIC implementation estimates of Lesamnta

Message digest size	Architecture	Gate count (k gates)	Max. frequency (MHz)	Throughput (Mbps)
256	Speed Optimized	190.1	282.5	6026.4
	Balance Optimized	68.0	636.9	3623.5
	Area Optimized	20.7	169.8	336.9
512	Speed Optimized	393.0	234.2	9992.2
	Balance Optimized	144.9	571.4	6501.6
	Area Optimized	44.3	144.1	571.9

7 Tunable Security Parameters

Lesamnta provides the following tunable security parameters.

1. The number of rounds for $EncComp_{256}$: Nr_comp256;

2. The number of rounds for $EncOut_{256}$: Nr_out256;

3. The number of rounds for $EncComp_{512}$: Nr_comp512; and

4. The number of rounds for $EncOut_{512}$: Nr_out512.

Choosing the values for these parameters enables selection of a range of possible security/performance tradeoffs. Considering the security analysis results described in Sec. 12, however, we recommend a value of 32 for each of these parameters, as specified in Sec. 5. Hereafter, we denote this recommended value of 32 by n_R.

8 Design Rationale

8.1 Block-Cipher-Based Hash Functions

The design rationale of Lesamnta is based on achieving the following goals:

- To provide the same application program interface as that of the SHA-2 family;

- To ensure both attack-based security and proof-based security; and

- To be efficient on a wide range of platforms.

To achieve these goals, we adopted an iterative hash function based on the block cipher as the basic design. Since the idea of building hash functions from block ciphers goes back more than 30 years, the enormous volume of research on this idea helped us to design Lesamnta.

Hence, Lesamnta basically follows a traditional design but incorporates new methods to resist recent attacks and provide security proof.

8.2 Domain Extension

The domain extension scheme of Lesamnta is designed to achieve the following goals: efficiency comparable to that of the Merkle-Damgård iteration, and security against the length-extension attack. The scheme consists of the Merkle-Damgård iteration of the compression function, enveloped with the output function. We call this MDO, and it is illustrated in Figure 31. Unlike the NMAC-like domain extension in [9], the output function g has the last block of a padded message input as a part of the input. The output function avoids the length-extension attack. The overhead of the output function is small, since it shares components with the compression function.

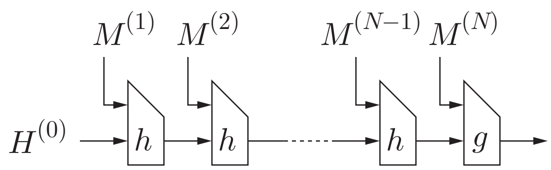

Figure 31: Domain extension scheme MDO. h is the compression function, and g is the output function. $\mathsf{pad}(M) = M^{(1)}\|M^{(2)}\|\cdots\|M^{(N-1)}\|M^{(N)}$, where pad is the padding function and M is a message input.

8.3 Compression Function

8.3.1 PGV Mode

The criteria taken into account in designing the compression function are the following:

- Efficiency equal to that of the underlying block cipher;
- Provable security in theoretical models; and
- Security evaluation using attacks against block ciphers.

The first criterion implies that the compression function should be as efficient as the underlying block cipher in terms of any computational resource. The second and third criteria imply that the security aspects of the compression function can be reduced to those of the block cipher.

The PGV modes [7] meet the first criterion, because they use the block cipher exactly one time. Not all PGV modes, however, meet the second criterion. It has been shown that the twelve PGV modes are secure in the ideal cipher model in terms of collision resistance and preimage resistance [7].

Lesamnta uses the Matyas-Meyer-Oseas (MMO) mode, which is one of the secure PGV modes in terms of collision resistance and preimage resistance. The MMO mode is defined as follows:

$$h(H^{(i-1)}, M^{(i)}) = E(H^{(i-1)}, M^{(i)}) \oplus M^{(i)} \;\; ,$$

where E is an encryption function and $H^{(i-1)}$ works as a key, as illustrated in Figure 32 [24].

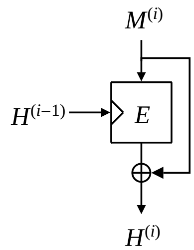

Figure 32: Matyas-Meyer-Oseas (MMO) mode

The MMO mode has no feedforward of the key, but only feedforward of the message. Compared with the other eleven secure PGV modes, it is easier to analyze the security of the MMO mode with block-cipher attacks. Thus, the security of the MMO mode can be reduced to the security of an underlying block cipher, in the senses of both proof-based security and attack-based security.

8.4 Output Function

To increase the security margin in terms of pseudo-randomness and to offer a tradeoff between security and efficiency, Lesamnta uses an output function g, constructed from an encryption function L in the following manner:

$$g(H^{(N-1)}, M^{(N)}) = L(H^{(N-1)}, M^{(N)}) \oplus M^{(N)} \quad .$$

8.5 Block Ciphers

Each of the four Lesamnta algorithms uses two block ciphers, E and L. We set the following requirements as goals for our design of these underlying block ciphers.

- 256-bit block ciphers for Lesamnta-224/256 and 512-bit block ciphers for Lesamnta-384/512.
- Key lengths of 256 bits for the 256-bit block ciphers and 512 bits for the 512-bit block ciphers
- Resistance against known attacks.
- Design simplicity:

  To facilitate ease of security analysis:

  To facilitate ease of implementation.
- Speed on processors for general purposes, on processors for servers, on future processors, and on various hardware platforms.
- Capable of implementation on an 8-bit processor with a small amount of RAM.
- Capable of implementation on hardware with a small gate count.

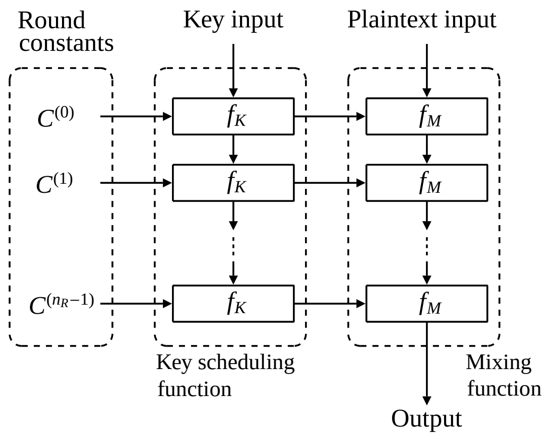

Figure 33: Structure of the encryption function for the hash function, E

Figure 33 shows an overview of the encryption function E.

The encryption function E is broken into two parts to process data: the key scheduling function and the mixing function. Each of these iteratively uses a sub-function. Therefore, we denote the corresponding sub-functions for the key scheduling function and mixing function by f_K and f_M, respectively.

Figure 34 shows an overview of the encryption function L.

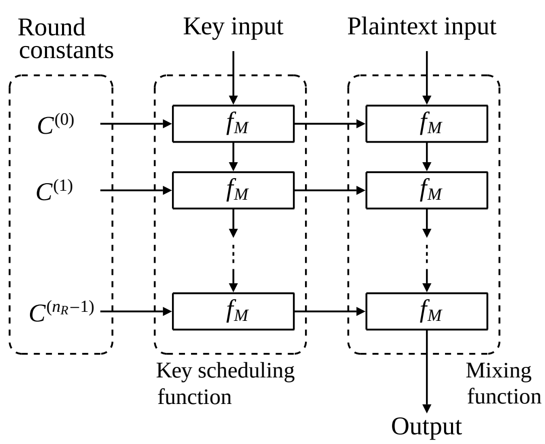

Figure 34: Structure of the encryption function for the output function, L

The structure of L is similar to that of E. In L, both the key scheduling function and the mixing function use f_M as the round function.

9 Motivation for Design Choices

9.1 Padding Method

The padding method of Lesamnta adopts Merkle-Damgård strengthening. Thus, the last block of a padded message includes the binary representation of the length of the message input.

For the padding method of Lesamnta, the last block does not contain any part of the message input. It only contains the length of the message input. As shown in Figs. 6 and 7 or Figs. 8 and 9, there are at most two possibilities for the last block corresponding to the remaining blocks. This property is necessary to prove that Lesamnta is indifferentiable from a random oracle in the ideal cipher model.

9.2 MMO Mode

We have four motivations for choosing the MMO mode.

1. Attack-based security
   From the viewpoint of attacks on a block cipher, recent collision-finding attacks use the fact that an attacker can directly control the key of a block cipher. This is because popular hash functions such as the SHA-2 family use the Davies-Meyer (DM) mode with a poor key scheduling function. In contrast, the MMO mode does not allow the attacker to control the key of a block cipher. Rather, since the key corresponds to the previous chaining values, the attack must control the chaining values by varying the message block. When we assume that the key (i.e., the previous chaining values) is fixed for the attacker, the attack model is similar to the attack model of block-cipher cryptanalysis. Then, known countermeasures against block-cipher cryptanalysis can be applied to design a secure MMO mode.

2. Proof-based security
   The MMO mode enables us to reduce the security of Lesamnta to that of the underlying block ciphers to a greater extent than with the DM mode used by the SHA family. In particular, the PRF property of HMAC is almost reduced to the PRP property of the underlying block ciphers. Furthermore, Lesamnta can be shown indifferentiable from a random oracle in the ideal cipher model.

3. Efficiency of implementation
   The computational resources required by the MMO mode are almost the same as those required by the block cipher. In particular, the following properties contribute to performance:

   - The number of invocations of the block cipher is exactly one.
   - The size of the internal buffer is less than that of other secure PGV modes such as the Miyaguchi-Preneel mode.
   - The output length is equal to that of the block cipher.

4. Resistance against side-channel attacks
Side-channel attacks should be taken into account in hardware implementation. It has been pointed out that one can perform side-channel attacks on HMAC with hash functions using the DM mode, such as the SHA family [27]. We thus adopt the MMO mode, with which HMACs remains secure against side-channel attacks.

9.3 Output Function

The primary purpose of the output function is to make length-extension attacks impossible. Resisting length-extension attacks requires that the following tasks be infeasible, where h and g are the compression function and the output function, respectively.

- To find $H^{(k-1)}, M^{(k)}$ satisfying $h(H^{(k-1)}, M^{(k)}) = g(H^{(k-1)}, M^{(k)})$; and
- To find $H^{(N-1)}$ satisfying $y = g(H^{(N-1)}, M^{(N)})$ for given y and $M^{(N)}$.

In Lesamnta, h and g are in the MMO mode, but the underlying block ciphers are different. The use of different block ciphers is effective in making the first task infeasible. To make the second task infeasible, Lesamnta uses a well-designed underlying block cipher for g. Additionally, to keep the implementation cost low, the block cipher of g consists of only the mixing function of h.

9.4 Block Cipher

Each algorithm of Lesamnta uses two block ciphers E and L. E is used in the compression function and the other is used in the output function. For reducing the hardware complexity, E shares the mixing function with L. In addition, the mixing function is identical to the key scheduling function in L except that the additional input parameter changes from the round key to the round constant.

The block size and key size of the block ciphers are both 256 (512) bits for Lesamnta-256 (Lesamnta-512). The block cipher plays an important role in both ensuring resistance against cryptanalytic attacks and achieving high performance. To meet these requirements, for the round function, we adopt a well-studied Feistel network and apply the design approach of AES in designing the F function, which is the most significant component in the underlying block ciphers. As a result, we can show that 12 rounds are secure against differential cryptanalysis in the sense that the maximum differential characteristic probability is less than 2^{-256} (2^{-512}).

9.4.1 Mixing Function

The plaintext is denoted by $P = (p_0, p_1, \ldots, p_7)$, and the ciphertext by $C = (c_0, c_1, \ldots, c_7)$. The mixing function is defined as follows:

$$\begin{aligned}
(x_0^{(0)}, x_1^{(0)}, \ldots, x_7^{(0)}) &= (p_0, p_1, \ldots, p_7) \ , \\
(x_0^{(r)}, x_1^{(r)}, \ldots, x_7^{(r)}) &= f_M(x_0^{(r-1)}, x_1^{(r-1)}, \ldots, x_7^{(r-1)}) \quad 1 \le r \le n_R \ , \\
(c_0, c_1, \ldots, c_7) &= (x_0^{(n_R)}, x_1^{(n_R)}, \ldots, x_7^{(n_R)}) \ .
\end{aligned}$$

9.4.1.1 Network in the Round Function

Our strategy to design the mixing function of Lesamnta is to construct it from block cipher components whose security and efficiency have been well-studied. This is because techniques to design and analyze block ciphers have been well understood through the AES competition. For now, we know a lot about both how to design 64-bit or 128-bit block ciphers and how to evaluate these ciphers.

Our design approach is to construct a 256-bit (512-bit) hash function from a 64-bit (128-bit) block-cipher like permutation. In this respect, the Feistel network is more suitable than the SP network since using the SP network would require to design 256-bit and 512-bit block ciphers which we think are less mature in terms of design, analysis, and implementation.

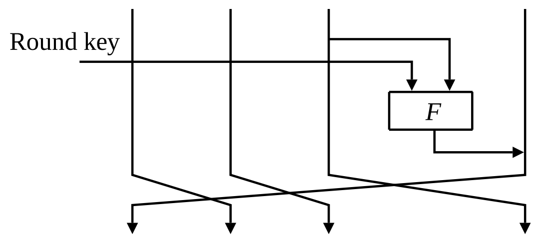

Figure 35: Type 1 4-branch generalized Feistel network

The mixing function of the block cipher of Lesamnta uses a type 1 4-branch generalized Feistel network (GFN) [36] for simplicity and hardware flexibility. It is illustrated in Fig. 35. For implementation reasons, each of the branches is stored in two 32-bit (64-bit) words for Lesamnta-256 (Lesamnta-512).

The round function f_M consists of XOR operations, a nonlinear function F, and a wordwise permutation. The F function is a non-linear transformation with a two-word input and a two-word round key input $K^{(r)}$ taken from the key schedule, and a two-word output. The round function f_M is defined as follows:

$$
\begin{aligned}
x_0^{(r)} \| x_1^{(r)} &= (x_6^{(r-1)} \| x_7^{(r-1)}) \oplus F(K^{(r)}, x_4^{(r-1)} \| x_5^{(r-1)}) \ , \\
x_2^{(r)} = x_0^{(r-1)}, &\quad x_3^{(r)} = x_1^{(r-1)}, \quad x_4^{(r)} = x_2^{(r-1)} \ , \\
x_5^{(r)} = x_3^{(r-1)}, &\quad x_6^{(r)} = x_4^{(r-1)}, \quad x_7^{(r)} = x_5^{(r-1)} \ .
\end{aligned}
$$

9.4.1.2 F Function

The functions F_{256} and F_{512} are the most significant components in the underlying block ciphers. Note that we denote F_{256} and F_{512} by F when the message digest size is not relevant. Our requirement on the F functions is both efficiency and resistance against known attacks such as differential cryptanalysis. Another requirement on the F functions is inversibility for a given round key to make the analysis of collision attacks easy. To design the F functions, we applied one of the most successful approaches known as the wide trail strategy [10] which is used in the design of AES. We can show that the maximum differential characteristic probability for Lesamnta-256

(Lesamnta-512) is less than 2^{-54} (2^{-150}) by applying the Four-Round Propagation Theorem in the wide trail strategy to the F functions:

Hereafter, we explain each step used in the F functions. In Lesamnta-224/256 and Lesamnta-384/512, operations are performed on SubState256 and SubState512.

The functions F_{256} and F_{512} are the composite mappings which are parameterized by the round key:

$F_{256} = \widetilde{F_{256}} \circ$ **`AddRoundKey256()`**,
where $\widetilde{F_{256}}$ = (**`ShiftRows256()`** $\circ$ **`ByteTranspos256()`** $\circ$ **`SubBytes256()`**)^4.
$F_{512} = \widetilde{F_{512}} \circ$ **`AddRoundKey512()`**,
where $\widetilde{F_{512}}$ = (**`ShiftRows512()`** $\circ$ **`ByteTranspos512()`** $\circ$ **`SubBytes512()`**)^4.

The function F is a sequence of transformations called *steps* like AES. The steps used in the full Lesamnta are the round key addition step, the non-linear step, the byte transposition step, and the linear diffusion step. For Lesamnta-384/512, each step in F_{512} is the same as the corresponding step in AES.

9.4.1.3 Round Key Addition Step

The round key addition steps **`AddRoundKey256()`** and **`AddRoundKey512()`** simply combine the SubState with the round key by means of bitwise XOR operation to facilitate ease of security analysis and of implementation.

9.4.1.4 Non-Linear Step

The non-linear steps **`SubBytes256()`** and **`SubBytes512()`** consist of parallel applications of a non-linear substitution box. As for the S-box, we apply the S-box used in AES, for security reasons and implementation reasons. This S-box has the following properties:

- The maximum differential probabilities are 2^{-6}.
- The S-box has no fixed points.

9.4.1.5 Byte Transposition Step

The byte transposition steps **`ByteTranspos256()`** and **`ByteTranspos512()`** cyclically shift rows over different numbers of bytes (offsets). These offsets are selected in a way that **`ByteTranspos256()`** and **`ByteTranspos512()`** are *diffusion optimal* [10], which means that the different bytes in each column are distributed over all different columns.

9.4.1.6 Linear Diffusion Step

The linear diffusion steps **`ShiftRows256()`** and **`ShiftRows512()`** are linear mappings based on the MDS code. An important diffusion measure introduced in [10] is the *branch number*. The branch numbers for **`ShiftRows256()`** and **`ShiftRows512()`** are 3 and 5, respectively.

`ShiftRows256()` and **`ShiftRows512()`** have an effect to mix the bytes in each SubState256 column and in each SubState512 column, respectively.

9.4.2 Key Scheduling Function

Since the structure of the key scheduling function is similar to that of the mixing function, strong non-linearity is ensured as compared with key scheduling functions of the SHA-2 family.

We designed the key scheduling function in E for the following purposes:

1. It introduces asymmetry which prevents symmetry between rounds leading to attacks such as slide attacks.

2. It provides the resistance against pseudo-collision attacks.

   Note that in the collision attack model, the attacker cannot control the input to the key scheduling function in a direct way due to the MMO mode while in the pseudo-collision attack model, he can.

3. It should be efficient on a wide range of platforms.

For the security purposes, the key scheduling function uses the type 1 general Feistel network where the non-linear function uses the composition of a non-linear step and the linear diffusion step as is commonly done in block ciphers. For the performance purposes, the linear diffusion step is composed of a linear mapping based on a MDS code and a bytewise permutation because linear diffusion steps consisting of a single linear mapping based on a MDS code would be expensive. The branch numbers of the linear mappings for E_{256} and E_{512} are 5 and 9, respectively. Since the key scheduling function shares most of its components with the mixing function, an efficient hardware implementation is possible.

9.4.3 Round Constants

The round constants introduce randomness, non-regularity, and asymmetry into the key scheduling function. The round constants of Lesamnta are generated by a counter-like function (Sec. 5.1). Each of two words of a round constant changes its value over rounds. This is because the linear mapping used in the key schedule operates on one word rather than two.

In contrast, the round constants of popular hash functions are often generated from real numbers such as $\sqrt{2}$. Hence, they are usually implemented via a large lookup table. Round constant generation by a counter-like function is more suitable for a hardware efficient implementation on resource-poor devices such as RFID tags than is generation by a large lookup table.

10 Expected Strength and Security Goals

Table 14 shows the expected strength of Lesamnta for each of the security requirements (i.e., the expected complexity of attacks). What values in Table 14 mean is explained below. The row indicated by "HMAC" lists the approximate number of queries required by any distinguishing attack against HMAC using Lesamnta. The row indicated by "PRF" lists the approximate number of queries required by any distinguishing attack against the additional PRF modes described in Sec. 13.1. The row indicated by "Randomized hashing" lists the approximate complexity to find another pair of a message and a random value for a given pair of a 2^k-bit message and a random value. The fourth row lists the approximate complexity of any collision attack. The fifth row lists the approximate complexity of any preimage attack. The sixth row lists the approximate complexity of the Kelsey-Schneier second-preimage attack with any first preimage shorter than 2^k bits. The seventh row lists the approximate number of queries required by any length-extension attack against Lesamnta. A cryptanalytic attack may be a profound threat to Lesamnta if its complexity is much less than the complexity in Table 14.

Table 14: Expected strength of Lesamnta

Requirement	Lesamnta			
	224	256	384	512
HMAC	2^{112}	2^{128}	2^{192}	2^{256}
PRF	2^{112}	2^{128}	2^{192}	2^{256}
Randomized hashing	2^{256-k}	2^{256-k}	2^{512-k}	2^{512-k}
Collision resistance	2^{112}	2^{128}	2^{192}	2^{256}
Preimage resistance	2^{224}	2^{256}	2^{384}	2^{512}
Second-preimage resistance	2^{256-k}	2^{256-k}	2^{512-k}	2^{512-k}
Length-extension attacks	2^{112}	2^{128}	2^{192}	2^{256}

Table 14 includes proof-based strength and attack-based strength. The security proof of Lesamnta is given as follows:

Proved security 1: Lesamnta is indifferentiable from a random oracle under the assumption that block ciphers E, L are independent ideal ciphers.

This proof partially ensures the security of randomized hashing, collision resistance, preimage resistance, second-preimage resistance, and length-extension attacks.

Proved security 2: Lesamnta is collision resistant under the assumption that the compression function h and the output function g are collision resistant.

This proof ensures the security of collision resistance, and in part, preimage resistance and second-preimage resistance.

Proved security 3: Lesamnta is a pseudorandom function under the assumption that block ciphers E, L are independent pseudorandom permutations.

This proof ensures the security of HMAC and PRF.

The attack-based strength is estimated in security analysis against known attacks described in Sec. 12.

11 Security Reduction Proof

11.1 MMO Mode

11.1.1 Collision Resistance

The collision resistance of the MMO mode is proved in the ideal cipher model. The MMO mode is given by $h(H, M) = E(H, M) \oplus M$, where E is an ideal cipher. Consider an infinitely powerful adversary A that makes q queries to E and E^{-1}. Then, the col-advantage of A is defined as

$$\begin{aligned}\mathbf{Adv}_h^{\mathrm{col}}(A) = \Pr\Big[&((H, M) \neq (H', M') \wedge h(H, M) = h(H', M')) \\ &\vee h(H, M) = H^{(-1)} | A^{E,E^{-1}} = ((H, M), (H', M'))\Big] ,\end{aligned}$$

where n is the block length of E. According to Black et al.'s analysis [7], the col-advantage is given by

$$\frac{0.039(q-1)(q-2)}{2^n} \leq \mathbf{Adv}_h^{\mathrm{col}}(A) \leq \frac{q(q+1)}{2^n} .$$

The above inequality means that any adversary must make about $2^{n/2}$ queries to find a collision.

In Lesamnta, the dedicated block cipher is in place of the ideal cipher E. Although it is not the ideal cipher, the above inequality suggests that the MMO mode is a good choice for designing a compression function.

11.1.2 Preimage Resistance

The preimage resistance of the MMO mode is proved in the ideal cipher model. Then, the pre-advantage of A is defined as, for any public constant K,

$$\mathbf{Adv}_h^{\mathrm{pre}}(A) = \Pr\left[M \notin Q \wedge h(K, M) = H | A^{E,E^{-1}} = (M, H)\right]$$

where Q is the set of messages that A sends to E and A receives from E^{-1} [7]. Since $h(K, M) = E(K, M) \oplus M$, the pre-advantage is transformed into

$$\mathbf{Adv}_h^{\mathrm{pre}}(A) = \Pr\left[M \notin Q \wedge E(K, M) = H \oplus M | A^{E,E^{-1}} = (M, H)\right].$$

Denoting by q the number of queries, we have

$$\mathbf{Adv}_h^{\mathrm{pre}}(A) = \frac{1}{2^n - q}.$$

In Lesamnta, the dedicated block cipher is in place of the ideal cipher E. Although it is not the ideal cipher, the preimage resistance of the MMO mode is reduced to the correlation between a plaintext and a ciphertext for a known key.

11.1.3 Pseudorandom Function

Consider an adversary A that outputs a bit after making queries to an oracle. Suppose that K is randomly chosen from a key space, ρ is a random function, and π is a random permutation. Then, the prf-advantage and the prp-advantage of A is defined as

$$\begin{aligned}\mathbf{Adv}_E^{\mathrm{prf}}(A) &= \left|\Pr\left[A^{E(K,\cdot)} = 1\right] - \Pr\left[A^{\rho} = 1\right]\right|, \\ \mathbf{Adv}_E^{\mathrm{prp}}(A) &= \left|\Pr\left[A^{E(K,\cdot)} = 1\right] - \Pr\left[A^{\pi} = 1\right]\right|,\end{aligned}$$

where E is an underlying block cipher of the MMO mode. For any adversary A that makes q queries to the oracle where $q < 2^{n/2}$, the PRP/PRF switching lemma yields

$$\mathbf{Adv}_E^{\mathrm{prp}}(A) - \frac{q(q-1)}{2^{n+1}} \leq \mathbf{Adv}_E^{\mathrm{prf}}(A) \leq \mathbf{Adv}_E^{\mathrm{prp}}(A) + \frac{q(q-1)}{2^{n+1}}.$$

Since the MMO mode h is given by $h(K, M) = E(K, M) \oplus M$, there is an adversary B that makes queries the same times as A and has the same prf-advantage.

$$\mathbf{Adv}_h^{\mathrm{prf}}(B) = \mathbf{Adv}_E^{\mathrm{prf}}(A)$$

Hence, we have

$$\mathbf{Adv}_E^{\mathrm{prp}}(A) - \frac{q(q-1)}{2^{n+1}} \leq \mathbf{Adv}_h^{\mathrm{prf}}(B) \leq \mathbf{Adv}_E^{\mathrm{prp}}(A) + \frac{q(q-1)}{2^{n+1}}.$$

The above inequality roughly means that if E is a secure block cipher, then h is a pseudorandom function.

11.2 MDO Domain Extension with MMO Functions

11.2.1 Collision Resistance

It is easy to see that Lesamnta is collision-resistant (CR) if its compression function and output function are CR, that is, it is difficult to compute a pair of distinct (S, X) and (S', X') such that

$$E_S(X) \oplus X = E_{S'}(X') \oplus X' \quad \text{or} \quad L_S(X) \oplus X = L_{S'}(X') \oplus X'$$

for the underlying block ciphers E and L. Unfortunately, the pseudorandomness of a block cipher cannot imply the property. It is easy to find a counterexample. However, it is still reasonable to assume that well-designed block ciphers have this property.

The CR of Lesamnta can also be proved in the ideal cipher model using the technique by Black et al. in [7].

11.2.2 HMAC

Lesamnta supports HMAC specified in FIPS 198:

$$\mathsf{HMAC}(K, M) = H((K \oplus \mathtt{opad}) \| H((K \oplus \mathtt{ipad}) \| M)) ,$$

where H represents Lesamnta and K is a secret key. A diagram of HMAC using Lesamnta is given in Figure 36.

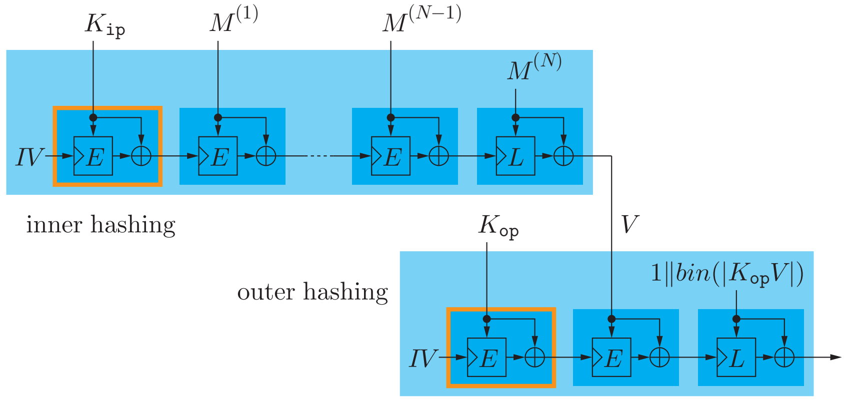

Figure 36: Diagram of HMAC using Lesamnta. E and L are underlying (n, n) block ciphers. $K_{\mathtt{ip}} = K \oplus \mathtt{ipad}$ and $K_{\mathtt{op}} = K \oplus \mathtt{opad}$. For a massage input M, $\mathsf{pad}(K_{\mathtt{ip}} \| M) = K_{\mathtt{ip}} M^{(1)} \cdots M^{(N)}$, where pad is the padding function. $bin(|K_{\mathtt{op}}V|)$ represents the $(n-1)$-bit binary representation of the length of $K_{\mathtt{op}} \| V$.

The security of HMAC using Lesamnta is reduced to the security of the underlying block ciphers. HMAC using Lesamnta resists any distinguishing attack that requires much fewer than $2^{n/2}$ queries if the underlying block ciphers are independent pseudorandom permutations and the following function is a pseudorandom bit generator:

$$\mu_E(K) = (E_{IV}(K_{\mathtt{op}}) \oplus K_{\mathtt{op}}) \| (E_{IV}(K_{\mathtt{ip}}) \oplus K_{\mathtt{ip}}) ,$$

where $K_{\mathtt{op}} = K \oplus \mathtt{opad}$ and $K_{\mathtt{ip}} = K \oplus \mathtt{ipad}$. More precise statements and proofs are given in Annex A.

11.2.3 Indifferentiability from the Random Oracle

Many cryptographic protocols are proved to be secure on the assumption that the underlying hash functions are random oracles. Thus, it is important to support this kind of results by validating the ideal assumption in such a way as in [9].

Lesamnta is shown to resist any attack to differentiate it from the random oracle with much fewer than $2^{n/2}$ queries in the ideal cipher model. More precise statements are given in Annex B.

12 Preliminary Analysis

In our preliminary analysis, we analyzed resistance of Lesamnta against various kinds of known attacks such as attacks collision-finding, first-preimage-finding, second-preimage-finding, length-extension attack, multicollision attack. The best results on attacks on Lesamnta-256 are a collision finding attack on 16 rounds with a complexity 2^{97}, a first preimage finding attack on 16 rounds with a complexity 2^{193}, and a second preimage finding attack on 16 rounds with a complexity 2^{193}. These attacks are easily repeated in the case of Lesamnta-512. The best results on attacks on Lesamnta-512 are a collision finding attack on 16 rounds with a complexity 2^{193}, a first preimage finding attack on 16 rounds with a complexity 2^{385}, and a second preimage finding attack on 16 rounds with a complexity 2^{385}.

In this section, we view the 256-bit internal state in Lesamnta-256 as four 64 bit words, instead of eight 32-bit words, in order to make the analysis easier. Similarly, we view the 512-bit internal state in Lesamnta-512 as four 128 bit words, instead of eight 64-bit words. We denote F_{256} and F_{512} by F. Furthermore, we decompose F as $F = \tilde{F} \circ$ **AddRoundKey**. Note that $\tilde{F}$ is a permutation.

Figure 37 and 38 illustrate another representation of F_M and $\tilde{F}$ permutation, respectively.

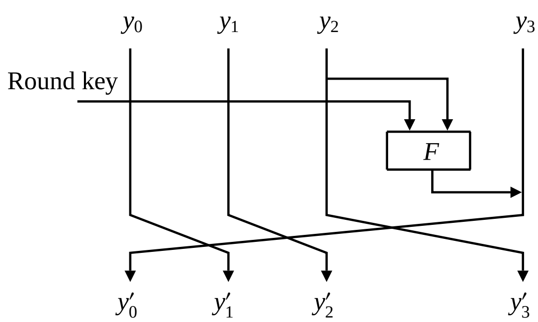

Figure 37: Another representation of F_M

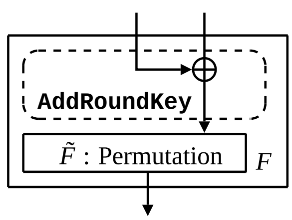

Figure 38: $\tilde{F}$ permutation

12.1 Length-Extension Attack

As an actual method for making the length-extension attack impossible, Lesamnta uses the output function different from the compression function. Furthermore, Lesamnta is proved to be indifferentiable from the random oracle in the ideal cipher model. Security against the length-extension attack is a necessary condition to be indifferentiable from the random oracle.

12.2 Multicollision Attack

Joux's multicollision attack [17] can be applied to Lesamnta. It is easy to see that the complexity to find 2^t collisions of Lesamnta is $O(t\,2^{n/2})$ if the birthday attack is used to find collisions of its compression function or output function.

12.3 Kelsey-Schneier Attack for Second-Preimage-Finding

The Kelsey-Schneier second-preimage attack [18] can be applied to Lesamnta. Against the attack, it has second-preimage resistance of approximately $n - k$ bits for any message shorter than 2^k bits.

12.4 Randomized Hashing Mode

The randomized hashing mode in NIST SP 800-106 [12] can be applied to Lesamnta. However, the more general mode called RMX [14] is suitable for iterated hash functions. The following function rmx specifies a version of RMX optimized for Lesamnta: It maximizes the number of random bits applied to the padded message. rmx takes two inputs: a message M and a random salt r. For simplicity, the length of r is assumed to be n, the output length of Lesamnta.

1. Let t be the minimum non-negative integer such that $|M| + t + 16 \equiv 0 \pmod{n}$.
2. $\tilde{M} = M\|0^t\|$(16-bit binary representation of t)
3. $R = \overbrace{r\|r\|\cdots\|r}^{|\tilde{M}|/n}$
4. $\mathsf{rmx}(M, r) \stackrel{\text{def}}{=} r\|(\tilde{M} \oplus R)$

The Kelsey-Schneier second-preimage attack can be applied to Lesamnta with rmx. Thus, it provides approximately $n - k$ bits of security against the following attack:

> The attacker chooses a message M with 2^k bits. Then, given random r, the attacker attempts to find a second message M' and a randomization value r' that yield the same randomized hash value.

12.5 Attacks for Collision-Finding, First (Second)-Preimage-Finding

In this section, we present a collision and second preimage attack for 16 rounds of Lesamnta-256. The analysis can easily be repeated for the case of 16 rounds of Lesamnta-512. This attack is based on our preliminary analysis and the analysis of a previous version of Lesamnta by Florian Mendel.

First, we show how to construct collisions for the compression function. Let $H = H_0\|H_1\|H_2\|H_3$ denote the output of the compression function. Now assume that we can find 2^{96} message blocks m^*, such that all message blocks produce the same value H_3. Then we know that due to the birthday paradox two of these message blocks also lead to the same values H_0, H_1, and H_2. In other words, we have constructed a collision for the compression function. Based on this short description, we will show now how to construct message blocks m^*, which all produce the same value H_3. We get the following characteristic:

Table 15: Characteristic for the collision attack

Round	Inputs (64-bit words)			
`message block`	Δ_0	Δ_1	Δ_2	$\Delta_3 \oplus \delta$
0	Δ_3	Δ_0	Δ_1	Δ_2
1	–	Δ_3	Δ_0	Δ_1
2	–	–	Δ_3	Δ_0
3	–	–	–	Δ_3
4	Δ_3	–	–	–
5	–	Δ_3	–	–
6	–	–	Δ_3	–
7	?	–	–	Δ_3
8	Δ_3	?	–	–
9	–	Δ_3	?	–
10	?	–	Δ_3	?
11	?	?	–	Δ_3
12	Δ_3	?	?	–
13	?	Δ_3	?	?
14	?	?	Δ_3	?
15	?	?	?	Δ_3
`feedforward`	?	?	?	δ

where the symbol ? denotes an arbitrary difference. and Δ denotes a message block difference The differences have to be selected such that they can be transformed by $\tilde{F}^{-1}$ in the following way:

$$\begin{aligned}\delta &\rightarrow \Delta_2\\ \Delta_2 &\rightarrow \Delta_1\\ \Delta_1 &\rightarrow \Delta_0\\ \Delta_0 &\rightarrow \Delta_3.\end{aligned}$$

It is easy to see that this characteristic for 16 rounds can be used to fix 64 bits of the output of the compression function. It can be summarized as follows.

1. Choose a random message block $m = M_0||M_1||M_2||M_3$ and compute $H = H_0||H_1||H_2||H_3$ and check if $H_3 = d$ for a predefined value d.

2. If $H_3 \neq d$ then adjust $\delta = H_3 \oplus d$ accordingly and compute

$$\begin{aligned}\Delta_2 &= M_2 \oplus (\tilde{F}^{-1}(\tilde{F}(M_2 \oplus K^{(0)}) \oplus \delta) \oplus K^{(0)}),\\ \Delta_1 &= M_1 \oplus (\tilde{F}^{-1}(\tilde{F}(M_1 \oplus K^{(1)}) \oplus \Delta_2) \oplus K^{(1)}),\\ \Delta_0 &= M_0 \oplus (\tilde{F}^{-1}(\tilde{F}(M_0 \oplus K^{(2)}) \oplus \Delta_1) \oplus K^{(2)}),\\ \Delta_3 &= (M_3 \oplus \delta) \oplus (\tilde{F}^{-1}(\tilde{F}(M_3 \oplus K^{(3)} \oplus \delta) \oplus \Delta_0) \oplus K^{(3)}),\end{aligned}$$

where $K^{(r)}$'s are round keys.

3. Now we have to construct m^* by adjusting m such that $H_3 = d$ as follows: $m^* = M_0 \oplus \Delta_0 || M_1 \oplus \Delta_1 || M_2 \oplus \Delta_2 || M_3 \oplus (\Delta_3 \oplus \delta)$

Hence, we can find a message block m^* such that $H_3 = d$ for an arbitrary value of d with a complexity of about 2 compression function evaluations. Therefore, we can find a collision for the compression function (and the hash function) with a complexity of about 2^{97} compression function evaluations.

In a similar way as we can construct a collision for the compression function, we can construct a preimage for the compression function. In the attack, we have to find a message m^*, such that $h(K, m^*) = H$ for the given value of H and K. Since we can find a message block m^*, where H_3 is correct (note that the value of d can be chosen freely) with a complexity of about 2 compression function evaluations, we can construct a preimage for the compression function with a complexity of 2^{193}. By repeating the attack 2^{192} times we will find a message block m^* such that H_0, H_1, and H_2 are correct.

Due to the final output transformation of the hash function we can not extend the attack to a preimage attack on the hash function. However we can use it to construct second preimages for the hash function with a complexity of about 2^{193} compression function evaluations.

12.5.1 Collision Attacks Using the Message Modification

Wang et al. showed methods for finding collisions for widely used hash functions including MD5 and SHA-1. Their approach is based on the differential cryptanalysis and the message modification technique. As for Lesamnta-256, the maximum differential characteristic probability for 12 rounds is less than 2^{-256} and the message block space is a 256-bit space. Their methods for finding collisions require a differential characteristic with a large probability and a large degree of freedom in the message block space. Considering the limited size of the message block space and very small maximum differential characteristic probability, it is very unlikely to apply their collision finding methods to Lesamnta-256. The analysis can easily be repeated for the case of Lesamnta-512.

12.6 Attacks for Non-Randomness-Finding

Despite the fact that the most threatening attacks on hash functions at this moment are differential attacks, we evaluate the security of Lesamnta with respect to various kinds of widely known attacks on block ciphers. These include not only differential attacks, but also linear attacks, interpolation attacks, and Square attacks.

The methods used to evaluate the compression function's resistance against these attacks are described below. In general, our analysis indicates that Lesamnta has large security margins against all of these attacks.

The motivation to analyze the Lesamnta compression function with respect to attacks which do not immediately apply to hash functions is that we want to ensure its security against future attacks which might borrow techniques from the field of block cipher cryptanalysis. Another motivation is that a number of block-cipher-based constructions, including the MMO mode, can be proved to be collision resistant if the underlying block cipher behaves as an ideal cipher (see [30, 7]). An ideal cipher has the true-randomness property.

The best way to ensure this randomness is to apply block cipher analysis techniques to the core function E, and to see if this reveals any weakness or non-random behavior. So far, we have not found any weakness in the full block cipher.

12.6.1 Differential and Linear Attacks

Considering the fact that the most successful attacks on hash functions are of differential nature, and that differential [5] and linear cryptanalysis [22] are two of the most powerful tools in block cipher cryptanalysis, we examined resistance of E and L against differential and linear attacks.

In order to estimate the strength of E with respect to differential and linear attacks, we compute upper bounds on the probabilities of differential and linear characteristics. As is commonly done in block cipher cryptanalysis, we will make abstraction of the exact differences or masks used in these characteristics, and just consider patterns of active S-boxes. Hereafter, we only explain our method of evaluating the security against differential cryptanalysis as we can apply a similar method regarding linear cryptanalysis because of its duality to differential cryptanalysis [8].

By applying the wide trail strategy, we can prove that the upper bounds on the probabilities of differential characteristics F_{256} and F_{512} are 2^{-54} and 2^{-150} respectively. On the other hand, it is easy to prove that four consecutive rounds has at least one active F function. As a result, it is provable that the probabilities of differential characteristics of 20 rounds of Lesamnta-256 and Lesamnta-512 are upperbounded by 2^{-256} and 2^{-512}. Furthermore, by making experiments with the Viterbi algorithm, we observed that 12 rounds of Lesamnta-256 and Lesamnta-512 have at least five active F functions, which means that 12 rounds of them achieve the above bounds as well. As a result, it is very unlikely to apply differential/linear attacks to the full Lesamnta.

12.6.2 Interpolation Attack

In the interpolation attack [16], an attacker constructs a polynomial using cipher input/output pairs and then he aims to determine key-dependent coefficients a polynomial expression of a cipher. If the number of terms in the polynomial expression is reasonably small, the interpolation attack can be mounted.

Lesamnta-256 uses the AES S-box which can be expressed as a polynomial of degree 254 over $GF(2^8)$. Lesamnta uses a fixed characteristic polynomial to represent an element over $GF(2^8)$. Our analysis only considers polynomial expressions based on this characteristic polynomial.

A few rounds of Lesamnta-256 can be expressed as a polynomial with 32 variables over $GF(2^8)$. We have confirmed that after the 10th round, an input to the F function depends on all the 32 variables. Then, due to high degree of the S-box, we expect that the number of coefficients reaches the maximum some rounds after the 10th round. This analysis is easily repeated in the case of Lesamnta-512. Thus we believe that the full 32 rounds Lesamnta is secure against interpolation attacks.

12.6.3 Square Attack

We analyze the resistance of Lesamnta against the Square attack [10]. (This attack is sometimes referred to as the *Saturation attack.*) It is a chosen-plaintext attack with security requirements in the case of block ciphers. An important characteristic of this attack is that it does not depend on the specific structure of the function $\tilde{F}$. The only requirement for this analysis to be valid, is that $\tilde{F}$ is an invertible transformation. This attack is based on our preliminary analysis and analysis of a previous version of Lesamnta by Vincent Rijmen. We present the attack for the case of Lesamnta-256. The analysis can easily be repeated for the case of Lesamnta-512.

In Table 16 we present a characteristic over 19 rounds. Here we start with a set of 2^{192} blocks such that the first 64 bits are constant and the remaining 192 bits take all values. We denote this by using the symbols b_1, b_2, b_3. Here a denotes that the input takes all possible values over the set, $-$ denotes that the input is constant, s denotes that the sum of the values over the set equals $-$, and '?' denotes that we cannot predict this input. Some explanation with this characteristic is as follows:

Round 1: Consider only the last two lines of the input. This Feistel construction is invertible hence we can write the symbols b_1, b_2, b_3 at the output. (Even if the values in the line marked by 'b_3' have changed.)

Round 4: At the output of round 4, we have the property that the 192 bits from the second, third and fourth lines take all possible values. Also the 192 bits from the first, second and third lines take all possible values. Note however that the values in the first and the fourth lines have no special relation among one another. This will cause a deterioration of property in round 8.

Round 16: The output s is the sum of 3 balanced words.

Suppose now that we would be studying a block cipher. Then, an attacker can use this characteristic to attack a 20-round version of the block ciphers E, L by guessing the last round key, partially decrypting the ciphertexts and checking whether the s property would hold. This would eliminate false guesses for the last round key.

The attacker would first construct 4 sets of 2^{192} texts with the right structure for the characteristic. Then, for each guess of the roundkeys of the last round (64 bits), the attacker would partially decrypt and verify whether he obtains an s. For a wrong guess of the roundkeys, this will happen with probability 2^{-64}. Hence after verifying against the 4 sets, all wrong guesses will have been eliminated. For most of the roundkeys, only one check needs to be done. The complexity of the attack can be roughly estimated as follows:

$$4 \times (2^{64} \text{ roundkey guesses }) \times (2^{192} \text{ partial decryptions/guess }) \times (\text{ complexity of one partial decryption})$$

Estimating the complexity of one partial decryption at $1/20 \approx 2^{-4.3}$ of a full decryption, we obtain for the total complexity the figure of $2^{253.7}$ full decryptions.

Table 16: Characteristic for the Square attack

Round	Inputs			
0	–	b_1	b_2	b_3
1	b_3	–	b_1	b_2
2	b_2	b_3	–	b_1
3	b_1	b_2	b_3	–
4	b_3	b_1	b_2	b_3
5	b_3	b_3	b_1	b_2
6	b_2	b_3	b_3	b_1
7	b_1	b_2	b_3	b_3
8	s	b_1	b_2	b_3
9	b_3	s	b_1	b_2
10	b_2	b_3	s	b_1
11	?	b_2	b_3	s
12	s	?	b_2	b_3
13	b_3	s	?	b_2
14	?	b_3	s	?
15	?	?	b_3	s
16	s	?	?	b_3
17	?	s	?	?
18	?	?	s	?
19	?	?	?	s

12.6.4 Attacks Using the Known-Key Distinguisher

Recently, a new method for attacking block ciphers has been proposed [31]. This attack is a distinguishing attack where the attacker knows the key. Therefore the distinguisher is called known-key distinguisher. We examined the resistance of Lesamnta-256 against this kind of attack. As a result, we can construct a known-key distinguisher for Lesamnta-256 reduced to 12 rounds. The distinguisher computes two plaintexts denoted by p and $\tilde{p}$ which have a special property. Let the corresponding ciphertexts be denoted by $c = (z_0, z_1, z_2, z_3)$ and $c = (\tilde{z}_0, \tilde{z}_1, \tilde{z}_2, \tilde{z}_3)$, then the following equation will hold with probability 1.

$$z_3 = \tilde{z}_3.$$

Figure 39 shows the algorithm to compute the plaintexts p and $\tilde{p}$ satisfying the equation.

Input :
The 12 subkeys $K^{(0)}, \ldots, K^{(11)}$, with $K^{(2)} \neq K^{(0)}$.

Algorithm :
1. Choose an arbitrary value for x.
2. Define the values γ, α as:

$$\gamma = K^{(2)} \oplus K^{(0)}$$
$$\alpha = \tilde{F}^{-1}(\tilde{F}(x) \oplus K^{(0)} \oplus K^{(8)}) \oplus x \oplus K^{(1)} \oplus K^{(5)}$$

3. Compute

$$p = (y_0, y_1, y_2, y_3)$$
$$\tilde{p} = (y_0, \tilde{F}^{-1}(y_2) \oplus K^{(3)}, \tilde{F}(y_1 \oplus K^{(3)}), y_3)$$

,where $y_0 = K^{(2)} \oplus \tilde{F}^{-1}(\alpha)$

It follows that $y_3 \oplus z_3 = \tilde{F}(y_2 \oplus \tilde{F}(y_1 \oplus K^{(3)}) \oplus K^{(8)}) = \tilde{y}_3 \oplus \tilde{z}_3$.
Consequently, $z_3 = \tilde{z}_3$.

Figure 39: Algorithm to compute the plaintexts p and $\tilde{p}$ satisfying the equation.

13 Extensions

13.1 Additional PRF Modes

13.1.1 Keyed-via-IV Mode

A PRF is obtained from Lesamnta by replacing the fixed initial value with a secret key. A diagram of the function, Keyed-Lesamnta, is given in Figure 40.

The security of Keyed-Lesamnta is reduced to the security of the underlying block ciphers. It resists any distinguishing attack that requires much fewer than $2^{n/2}$ queries if the underlying block ciphers are independent pseudorandom permutations. More precise statements and proofs are given in Annex C.

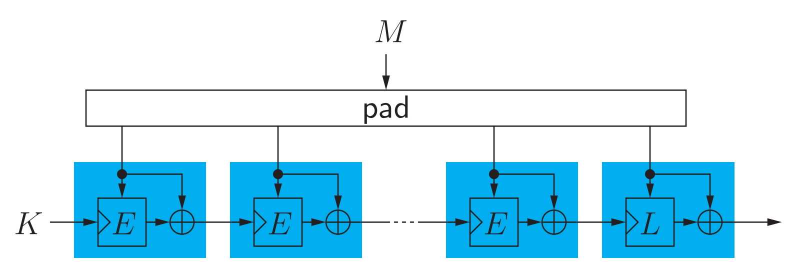

Figure 40: Diagram of Keyed-Lesamnta. E and L are underlying (n, n) block ciphers. pad is the padding algorithm. K is a secret key. M is a message input.

13.1.2 Key-Prefix Mode

The key-prefix mode is a method to construct a PRF with a given hash function. It simply feeds $K\|M$ to the hash function as an input, where K is a secret key and M is a message input. A diagram of the mode with Lesamnta is given in Figure 41. We call the function Key-Prefix-Lesamnta. This mode uses Lesamnta as a black box. In this sense, it is similar to HMAC. However, it is more efficient than HMAC.

Key-Prefix-Lesamnta resists any distinguishing attack that requires much fewer than $2^{n/2}$ queries if the underlying block ciphers are independent pseudorandom permutations and $E_{IV}(K)$ is pseudorandom. More precise statements and proofs are given in Annex C.

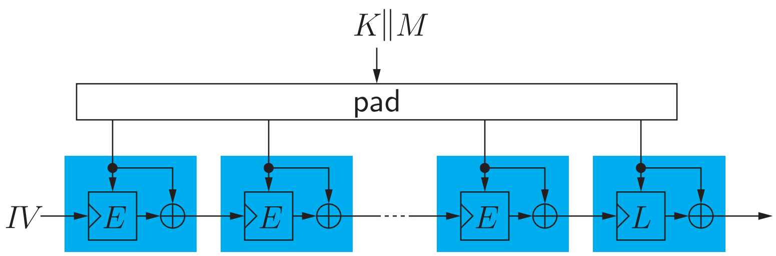

Figure 41: Diagram of Key-Prefix-Lesamnta. E and L are underlying (n, n) block ciphers. pad is the padding algorithm. K is a secret key. M is a message input.

13.2 Enhancement Against Second-preimage Attacks

To resist against the security of second-preimage attacks, we extend Lesamnta in such a way that round constants depend on not only the round index *round* but also the message-block index i. This extended version of Lesamnta is called Lesamnta-OOOe, for example, Lesamnta-256e. Since the compression function of this extended scheme depends on the message-block index i, this extended scheme is similar to HAIFA [4] and dithering hash [33] in this respect.

13.2.1 Lesamnta-224e and Lesamnta-256e

Let $C^{(i,round)}$ be a 64-bit constant for the $round^{th}$ round in the i^{th} message block. When the message block $M^{(i)}$ is processed, the Key Expansion routine **`KeyExpComp256()`**, described in Sec. 5.3.2.6 uses $C^{(i,round)}$ instead of $C^{(round)}$. Namely, **`KeyExpComp256()`** uses round constants $C^{(i,round)}$ that depend on both the message-block index i and the round index *round*, but do not depend on the message block itself. Notice that the other functions are unchanged. The constant $C^{(i,round)}$ is given by

$$C^{(i,round)} = C_0^{(i,round)} \parallel C_1^{(i,round)},$$

where $C_0^{(i,round)}$ and $C_1^{(i,round)}$ are 32-bit constants. The 32-bit constant $C_0^{(i,round)}$ is generated by the linear feedback shift register of the following primitive polynomial [29]

$$c_0(x) = x^{32} + x^{30} + x^{26} + x^{25} + 1,$$

where the initial value is 76543210 in hexadecimal. The 32-bit constant $C_1^{(i,round)}$ is the concatenation of a zero bit and a 31-bit sequence that is generated by the linear feedback shift register of the following primitive polynomial

$$c_1(x) = x^{31} + x^{28} + 1,$$

where the initial value is 01234567 in hexadecimal. Notice that the most significant bit of $C_1^{(i,round)}$ is always zero. Figure 42 shows the pseudocode for computing $C^{(i,round)}$.

```
ConstantGenerator256(word C[N-1][Nr_comp256][2])
begin
   word c0
   word c1

   c0 = 76543210 /* in hexadecimal */
   c1 = 01234567 /* in hexadecimal */
   for i = 1 to N-1
      for round = 0 to Nr_comp256 - 1
         word b0
         word b1
         /* >>: right shift, <<: left shift */
         b0 = c0 ⊕ (c0>>2) ⊕ (c0>>6) ⊕ (c0>>7)
         c0 = (c0 >> 1) ∨ (b0 << 31)
         /* ∧: bitwise AND, 00000001 in hexadecimal */
         b1 = (c1 ⊕ (c1>>3)) ∧ 00000001
         c1 = (c1 >> 1) ∨ (b1 << 30)
         C[i][round][0] = c0
         C[i][round][1] = c1
         C^(i,round) is given by C[i][round][0]||C[i][round][1].
      end for
   end for
end
```

Figure 42: Pseudocode for computing 64-bit constants

When the message block $M^{(i)}$ is processed, the Key Expansion routine **KeyExpComp256()** uses C[i][round][0] and C[i][round][1] instead of C[round][0] and C[round][1], respectively.

Some round constants $C^{(i,round)}$ in hexadecimal are given below.

$C^{(1,0)}$ = bb2a19004091a2b3, $C^{(1,1)}$ = 5d950c806048d159,
$C^{(1,2)}$ = aeca8640302468ac, $C^{(1,3)}$ = d765432058123456,
...
$C^{(1,30)}$ = 89e98c5a31072dcb, $C^{(1,31)}$ = c4f4c62d188396e5,
$C^{(2,0)}$ = 627a63164c41cb72, $C^{(2,1)}$ = b13d318b2620e5b9.

13.2.2 Lesamnta-384e and Lesamnta-512e

Let $C^{(i,round)}$ be a 128-bit constant for the $round^{th}$ round in the i^{th} message block. When the message block $M^{(i)}$ is processed, the Key Expansion routine **KeyExpComp512()** described in Sec. 5.5.2.6

uses $C^{(i,round)}$ instead of $C^{(round)}$. Notice that the other functions are unchanged. The constant $C^{(i,round)}$ is given by

$$C^{(i,round)} = C_0^{(i,round)} \parallel C_1^{(i,round)},$$

where $C_0^{(i,round)}$ and $C_1^{(i,round)}$ are 64-bit constants. The 64-bit constant $C_0^{(i,round)}$ is generated with the linear feedback shift register of the following primitive polynomial

$$c_0(x) = x^{64} + x^{63} + x^{61} + x^{60} + 1,$$

where the initial value is `fedcba9876543210` in hexadecimal. The 64-bit constant $C_1^{(i,round)}$ is the concatenation of a zero bit and a 63-bit sequence that is generated with the linear feedback shift register of the following primitive polynomial

$$c_1(x) = x^{63} + x^{62} + 1,$$

where the initial value is `0123456789abcdef` in hexadecimal. Notice that the most significant bit of $C_1^{(i,round)}$ is always zero. Figure 43 shows the pseudocode for computing $C^{(i,round)}$.

```
ConstantGenerator512(word C[N-1][Nr_comp512][2])
begin
   word c0
   word c1

   c0 = fedcba9876543210 /* in hexadecimal */
   c1 = 0123456789abcdef /* in hexadecimal */
   for i = 1 to N-1
      for round = 0 to Nr_comp512 - 1
         word b0
         word b1
         /* >>: right shift, <<: left shift */
         b0 = c0 ⊕ (c0>>1) ⊕ (c0>>3) ⊕ (c0>>4)
         c0 = (c0 >> 1) ∨ (b0 << 63)
         /* ∧: bitwise AND, 0000000000000001 in hexadecimal */
         b1 = (c1 ⊕ (c1>>1)) ∧ 0000000000000001
         c1 = (c1 >> 1) ∨ (b1 << 62)
         C[i][round][0] = c0
         C[i][round][1] = c1
         C^(i,round) is given by C[i][round][0]||C[i][round][1].
      end for
   end for
end
```

Figure 43: Pseudo code for computing 128-bit constants

Some round constants $C^{(i,round)}$ in hexadecimal are given below.

$$
\begin{aligned}
C^{(1,0)} &= \texttt{ff6e5d4c3b2a19080091a2b3c4d5e6f7}, \\
C^{(1,1)} &= \texttt{ffb72ea61d950c840048d159e26af37b}, \\
C^{(1,2)} &= \texttt{7fdb97530eca8642002468acf13579bd}, \\
C^{(1,3)} &= \texttt{bfedcba98765432140123456789abcde}, \\
&\cdots \\
C^{(1,30)} &= \texttt{89a3dcf7fdb975304d7e2b1802468acf}, \\
C^{(1,31)} &= \texttt{c4d1ee7bfedcba9826bf158c01234567}, \\
C^{(2,0)} &= \texttt{6268f73dff6e5d4c135f8ac60091a2b3}, \\
C^{(2,1)} &= \texttt{b1347b9effb72ea609afc5630048d159}.
\end{aligned}
$$

13.2.3 Selection of Polynomials

This extension uses a sequence produced by two primitive polynomials $c_0(x), c_1(x)$. We chose primitive polynomials consisting of as small terms as possible because such polynomials can be implemented efficiently on hardware. Since there is no primitive trinomial with degree 32 and 64, we chose primitive polynomials consisting of five terms. Since there are primitive trinomials with degree 31 and 63, we chose them.

In the case of Lesamnta-256e, polynomials $c_0(x), c_1(x)$ produce sequences with period $2^{32} - 1$ and $2^{31} - 1$, respectively. Since $\mathrm{GCD}(2^{32} - 1, 2^{31} - 1) = 1$ and Lesamnta-256e accepts a $(2^{64} - 1)$-bit message at most, $C^{(i,round)} = C^{(i',round')}$ if and only if $i = i'$ and $round = round'$ where $1 \leq i, i' \leq N-1$ and $0 \leq round, round' < \texttt{Nr\_comp256}$. It follows that the block cipher $EncComp_{256}$ depends on the message-block index i. Similarly, the block cipher $EncComp_{512}$ of Lesamnta-512e depends on the message-block index i because $\mathrm{GCD}(2^{64} - 1, 2^{63} - 1) = 1$ and Lesamnta-512e accepts a $(2^{128} - 1)$-bit message at most.

14 Advantages and Limitations

14.1 Advantages

Flexibility

- The number of the rounds of the underlying block ciphers is a tunable parameter. It allows the selection of a range of possible security/performance tradeoffs.
- Lesamnta can be implemented securely and efficiently on a wide variety of platforms, including constrained environments, such as smart cards.

Simplicity

- We take a rather conservative and simple approach to design Lesamnta. It is a Merkle-Damgård iterated hash function of a compression function enveloped by an output function. Furthermore, both the compression function and the output function are MMO modes using distinct block ciphers.
- The underlying block ciphers do not base its security or part of it on obscure and not well understood interactions between arithmetic operations.
- The tight design of Lesamnta does not leave enough room to hide a trapdoor.

Hardware Design Scalability

- Lesamnta is suited to be implemented in dedicated hardware. Hardware architectures of Lesamnta can be designed to meet the high-speed processing demand because of its highly parallelizable structure.
- The type-1 general Feistel network used in Lesamnta allows to process three F functions in parallel without additional delay. As for designing size-optimized architectures, Lesamnta has a nice feature that the F function is parallel and it consists of four iterations of the same function. The gate count of the Lesamnta hardware can be reduced by using a shared function module.

14.2 Limitations

- The design of the Lesamnta domain extension is performance-oriented, and it makes only a small change to the Merkle-Damgård iteration. It does not increase the resistance against Joux's multicollision attack and the Kelsey-Schneier second-preimage attack in comparison with the SHA-2 family.

15 Applications of Hash Functions

Lesamnta has the same application program interface as the SHA-2 family. Therefore, Lesamnta supports all applications that are supported by the SHA-2 family such as:

- digital signatures (FIPS 186-2);
- key derivation (NIST Special Publication 800-56A);
- hash-based message authentication codes (FIPS 198); and
- deterministic random bit generators (SP 800-90).

The proof-based and attack-based security analyses show that the security provided by Lesamnta against known attacks is not less than that provided by the SHA-2 family.

16 Trademarks

- ARM® and RealView® are registered trademarks and ARM926EJ-S™ is a trademark of ARM Limited in the United States and/or other countries.
- Atmel®, AVR® and AVR Studio® are registered trademarks of Atmel Corporation in the United States and/or other countries.
- Intel® is a registered trademark and Core™ is a trademark of Intel Corporation in the United States and/or other countries.
- Microsoft®, Visual Studio® and Windows Vista® are registered trademarks of Microsoft Corporation in the United States and/or other countries.
- Renesas® and H8® are registered trademarks of Renesas Technology Corporation in the United States and/or other countries.

17 Acknowledgments

In the first place we would like to thank Kota Ideguchi for his efficient ANSI-C and assembly implementations. Many people have been extremely helpful during the design of Lesamnta. In particular we would like to thank Kazuo Ota, Kazuo Sakiyama, Lei Wang, Yasuko Fukuzawa, Toru Owada. We would like to thank Florian Mendel, Vincent Rijmen, Orr Dunkelman, Sebastiaan Indesteege, Özgül Küçük, Bart Preneel, Hongjun Wu for their cryptanalysis of preliminary versions. We would like to thank Masahiro Ito, Satoshi Kawanami, and Yuji Matsuo who helped us with the proposal of Lesamnta from implementation perspective. This work was partially supported by the National Institute on Information and Communications Technology, Japan. Finally we would also like to thank the NIST SHA-3 team for initiating the SHA-3 process.

References

[1] M. Bellare, "New proofs for NMAC and HMAC : Security without collision-resistance," Advances in Cryptology - CRYPTO 2006, Lecture Notes in Computer Science, vol. 4117, pp. 602–619, 2006. `http://eprint.iacr.org/2006/043.pdf`.

[2] M. Bellare, R. Canetti, and H. Krawczyk, "Keying hash functions for message authentication," Advances in Cryptology - CRYPTO '96, Lecture Notes in Computer Science, vol. 1109, pp. 1–15, 1996. `http://www-cse.ucsd.edu/~mihir/papers/kmd5.pdf`.

[3] M. Bellare and T. Kohno, "Hash function balance and its impact on birthday attacks," Advances in Cryptology - EUROCRYPT 2004, Lecture Notes in Computer Science, vol. 3027, pp. 401–418, 2004. `http://www-cse.ucsd.edu/users/mihir/papers/balance.pdf`.

[4] E. Biham and O. Dunkelman, "A framework for iterative hash functions — HAIFA," The Second Cryptographic Hash Workshop, 2006. `http://csrc.nist.gov/groups/ST/hash/documents/DUNKELMAN_NIST3.pdf`.

[5] E. Biham and A. Shamir, *Differential Cryptanalysis of the Data Encryption Standard*, Springer, 1993.

[6] A. Biryukov and D. Wagner, "Advanced slide attacks," Advances in Cryptology - EUROCRYPT 2000, Lecture Notes in Computer Science, vol. 1807, pp. 589–606, 2000. `http://www.iacr.org/archive/eurocrypt2000/1807/18070595-new.pdf`.

[7] J. Black, P. Rogaway, and T. Shrimpton, "Black-box analysis of the block-cipher-based hash-function constructions from PGV," Advances in Cryptology - CRYPTO 2002, Lecture Notes in Computer Science, vol. 2442, pp. 320–335, 2002.

[8] F. Chabaud and S. Vaudenay, "Links between differential and linear cryptanalysis," Advances in Cryptology - EUROCRYPT '94, Lecture Notes in Computer Science, vol. 950, pp. 356–365, 1995.

[9] J.S. Coron, Y. Dodis, C. Malinaud, and P. Puniya, "Merkle-Damgård revisited: How to construct a hash function," Advances in Cryptology - CRYPTO 2005, Lecture Notes in Computer Science, vol. 3621, pp. 430–448, 2005.

[10] J. Daemen, L. R. Knudsen, and V. Rijmen, "The block cipher SQUARE," Fast Software Encryption, FSE '97, Lecture Notes in Computer Science, vol. 1267, pp. 149–165, 1997. `http://www.esat.kuleuven.ac.be/~cosicart/pdf/VR-9700.PDF`.

[11] I. B. Damgård, "A design principle for hash functions," Advances in Cryptology - CRYPTO '89, Lecture Notes in Computer Science, vol. 435, pp. 416–427, 1990.

[12] Q. Dang, "Randomized hashing digital signatures (2nd draft)," Draft NIST Special Publication 800-106, 2008. http://csrc.nist.gov/publications/drafts/800-106/2nd-Draft_SP800-106_July2008.pdf.

[13] B. Gladman, http://fp.gladman.plus.com/cryptography_technology/.

[14] S. Halevi and H. Krawczyk, "Strengthening digital signatures via randomized hashing," Advances in Cryptology - CRYPTO 2006, Lecture Notes in Computer Science, vol. 4117, pp. 41–59, 2006. http://www.ee.technion.ac.il/~hugo/rhash/rhash.pdf, http://tools.ietf.org/html/draft-irtf-cfrg-rhash-01.

[15] S. Hirose, J. H. Park, and A. Yun, "A simple variant of the Merkle-Damgård scheme with a permutation," Advances in Cryptology - ASIACRYPT 2007, Lecture Notes in Computer Science, vol. 4833, pp. 113–129, 2007.

[16] T. Jakobsen and L. R. Knudsen, "The interpolation attack on block ciphers," Fast Software Encryption, FSE '97, Lecture Notes in Computer Science, vol. 1267, pp. 28–40, 1997. http://homes.esat.kuleuven.be/~cosicart/ps/LRK-9700.ps.gz.

[17] A. Joux, "Multicollisions in iterated hash functions. Application to cascaded construction," Advances in Cryptology - CRYPTO 2004, Lecture Notes in Computer Science, vol. 3152, pp. 306–316, 2004.

[18] J. Kelsey and B. Schneier, "Second preimages on n-bit hash functions for much less than 2^n work," Advances in Cryptology - EUROCRYPT 2005, Lecture Notes in Computer Science, vol. 3494, pp. 474–490, 2005. http://www.schneier.com/paper-preimages.pdf.

[19] L. R. Knudsen, "Truncated and higher order differentials," Fast Software Encryption – Second International Workshop, Lecture Notes in Computer Science, pp. 196–211, 1995. ftp://ftp.esat.kuleuven.ac.be/cosic/knudsen/trunc.ps.Z.

[20] P. Koche, J. Jaffe, and B. Jun, "Differential power analysis," Advances in Cryptology - CRYPTO '99, Lecture Notes in Computer Science, vol. 1666, pp. 388–397, 1999.

[21] K. Lemke, K. Schramm, and C. Paar, "DPA on n-bit sized boolean and arithmetic operations and its application to IDEA, RC6, and the HMAC-construction," Cryptographic Hardware and Embedded Systems - CHES 2004, vol. 3156, pp. 205–219, 2004.

[22] M. Matsui, "Linear cryptanalysis method for DES cipher," Lecture Notes in Computer Science Advances in Cryptology - EUROCRYPT '93, vol. 765, pp. 386–397, 1994.

[23] U. Maurer, R. Renner, and C. Holenstein, "Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology," First Theory of Cryptography Conference, TCC 2004, Lecture Notes in Computer Science, vol. 2951, pp. 21–39, 2004.

[24] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone, *HANDBOOK of APPLIED CRYPTOGRAPHY*, CRC Press, 1996.

[25] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Investigations of power analysis attacks on smartcards," Proceedings of the USENIX Workshop on Smartcard Technology on USENIX Workshop on Smartcard Technology, 1999. `http://www.usenix.org/events/smartcard99/full_papers/messerges/messerges.pdf`.

[26] National Institute of Standards and Technology, "Secure hash standard," Federal Information Processing Standards Publication 180-2, August 2002. `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf`.

[27] K. Okeya, "Side channel attacks against HMACs based on block-cipher based hash functions," Information Security and Privacy, 11th Australasian Conference, ACISP 2006, Lecture Notes in Computer Science, vol. 4058, pp. 432–443, 2006.

[28] D. A. Osvik, "Speeding up Serpent," AES Candidate Conference, pp. 317–329, 2000. `http://www.ii.uib.no/~osvik/pub/aes3.pdf`.

[29] W. W. Peterson and J. E. J. Weldon. *Error-Correcting Codes*. The MIT Press, 1972.

[30] B. Preneel, R. Govaerts, and J. Vandewalle, "Hash functions based on block ciphers: a synthetic approach," Advances in Cryptology - CRYPTO '93, Lecture Notes in Computer Science, vol. 773, pp. 368–378, 1994.

[31] L. R. Knudsen, and V. Rijmen, "Known-Key Distinguishers for Some Block Ciphers," Asiacrypt 2007, Lecture Notes in Computer Science, vol. 1267, pp. 149–165, 2007.

[32] R. Rivest, "The MD5 message-digest algorithm," Request for Comments, no. 1321, April 1992. `ftp://ftp.rfc-editor.org/in-notes/rfc1321.txt`.

[33] R. L. Rivest, "Abelian square-free dithering and recoding for iterated hash functions," First Cryptographic Hash Workshop, 2005. `http://csrc.nist.gov/groups/ST/hash/documents/rivest-asf-paper.pdf`.

[34] X. Wang, X. Lai, D. Feng, H. Chen, and X. Yu, "Cryptanalysis of the hash functions MD4 and RIPEMD," Advances in Cryptology - EUROCRYPT 2005, Lecture Notes in Computer Science, vol. 3494, pp. 1–18, 2005.

[35] X. Wang, Y. L. Yin, and H. Yu, "Finding collisions in the full SHA-1," Advances in Cryptology - CRYPTO 2005, Lecture Notes in Computer Science, vol. 3621, pp. 17–36, 2005.

[36] Y. Zheng, T. Matsumoto, and H. Imai, "On the construction of block ciphers provably secure and not relying on any unproved hypotheses," Advances in Cryptology - CRYPTO '89, Lecture Notes in Computer Science, vol. 435, pp. 461–480, 1990.

18 List of Annexes

A HMAC Using Lesamnta Is a PRF

A.1 Definitions

Let $\mathsf{Func}(D, R)$ be the set of all functions from D to R, and $\mathsf{Perm}(D)$ be the set of all permutations on D. Let $s \xleftarrow{\$} S$ represent that an element s is selected from the set S under the uniform distribution.

Pseudorandom Bit Generator Let μ be a function such that $\mu : \{0,1\}^n \to \{0,1\}^l$, where $n < l$. Let A be a probabilistic algorithm which outputs 0 or 1 for a given input in $\{0,1\}^l$. The prbg-advantage of A against μ is defined as follows:

$$\mathrm{Adv}_{\mu}^{\mathrm{prbg}}(A) = \left|\Pr[A(\mu(k)) = 1 \,|\, k \xleftarrow{\$} \{0,1\}^n] - \Pr[A(s) = 1 \,|\, s \xleftarrow{\$} \{0,1\}^l]\right| ,$$

where the probabilities are taken over the coin tosses by A and the uniform distributions on $\{0,1\}^n$ and $\{0,1\}^l$. μ is called a pseudorandom bit generator (PRBG) if $\mathrm{Adv}_{\mu}^{\mathrm{prbg}}(A)$ is negligible for any efficient A.

Pseudorandom Function Let $f : K \times D \to R$ be a keyed function or a function family. $f(k, \cdot)$ is often denoted by $f_k(\cdot)$. Let A be a probabilistic algorithm which has oracle access to a function from D to R. A first asks elements in D and obtains the corresponding elements in R with respect to the function, and then outputs 0 or 1. The prf-advantage of A against f is defined as follows:

$$\mathrm{Adv}_{f}^{\mathrm{prf}}(A) = \left|\Pr[A^{f_k} = 1 \,|\, k \xleftarrow{\$} K] - \Pr[A^{\rho} = 1 \,|\, \rho \xleftarrow{\$} \mathsf{Func}(D, R)]\right| ,$$

where the probabilities are taken over the coin tosses by A and the uniform distributions on K and $\mathsf{Func}(D, R)$. f is called a pseudorandom function (PRF) if $\mathrm{Adv}_{f}^{\mathrm{prf}}(A)$ is negligible for any efficient A.

Let $p : K \times D \to D$ be a keyed permutation or a permutation family. The prp-advantage of A against p is defined similarly:

$$\mathrm{Adv}_{p}^{\mathrm{prp}}(A) = \left|\Pr[A^{p_k} = 1 \,|\, k \xleftarrow{\$} K] - \Pr[A^{\rho} = 1 \,|\, \rho \xleftarrow{\$} \mathsf{Perm}(D)]\right| .$$

p is called a pseudorandom permutation (PRP) if $\mathrm{Adv}_{p}^{\mathrm{prp}}(A)$ is negligible for any efficient A.

Pseudorandom Function Pair Let A be a probabilistic algorithm which has oracle access to a pair of functions from D to R. The prf-pair-advantage (prfp-advantage) of A against a pair of functions (f, g) is given by

$$\mathrm{Adv}_{f,g}^{\mathrm{prfp}}(A) = \left|\Pr[A^{f_k, g_k} = 1 \,|\, k \xleftarrow{\$} K] - \Pr[A^{\rho,\rho'} = 1 \,|\, \rho, \rho' \xleftarrow{\$} \mathsf{Func}(D, R)]\right| ,$$

where the probabilities are taken over the coin tosses by A and the uniform distributions on K and $\mathsf{Func}(D, R)$. (f, g) is called a PRF pair if $\mathrm{Adv}^{\mathrm{prfp}}_{f,g}(A)$ is negligible for any efficient A.

For a pair of permutations, the prpp-advantage of an adversary and a PRP pair can also be defined similarly.

Computationally Almost Universal Function Family Computationally almost universal function families are formalized by Bellare in [1]. Let $f : K \times D \to R$ be a function family. Let A be a probabilistic algorithm which takes no inputs and produces a pair of elements in D. The au-advantage of A against f is defined as follows:

$$\mathrm{Adv}^{\mathrm{au}}_{f}(A) = \Pr[f_k(M_1) = f_k(M_2) \wedge M_1 \neq M_2 \,|\, (M_1, M_2) \leftarrow A \wedge k \stackrel{\$}{\leftarrow} K] \ ,$$

where the probabilities are taken over the coin tosses by A and the uniform distribution on K. f is called a computationally almost universal function family if $\mathrm{Adv}^{\mathrm{au}}_{f}(A)$ is negligible for any efficient A.

A.2 Analysis

In the analysis of this section, for HMAC using Lesamnta, it is assumed that the length of an input M is a multiple of n and that the padding is not applied to $K \| M$. We call this slightly generalized function $\mathsf{HMAC}[E, L, IV]$. The proof technique given by Bellare in [1] is used in the analysis.

First, the compression function construction is considered. The following lemma says that the MMO compression function is a PRF up to the birthday bound when keyed via the chaining variable if the underlying block cipher is a PRP under the chosen plaintext attack. The proof is easy and omitted.

Lemma 1 Let E be an (n, n) block cipher and h be a function such that $h_K(x) = E_K(x) \oplus x$. Let A_h be a prf-adversary against h which runs in time at most t and asks at most q queries. Then, there exists a prp-adversary A_E against E such that

$$\mathrm{Adv}^{\mathrm{prf}}_{h}(A_h) \leq \mathrm{Adv}^{\mathrm{prp}}_{E}(A_E) + \frac{q(q-1)}{2^{n+1}} \ ,$$

where A_E runs in time at most $t + O(q)$ and asks at most q queries.

The following lemma says that the pair of the MMO compression function and the MMO output function is a PRF pair up to the birthday bound if the pair of the underlying block ciphers is a PRP pair under the chosen plaintext attack. The proof is easy and omitted.

Lemma 2 Let E and L be (n, n) block ciphers. Let h and g be functions such that $h_K(x) = E_K(x) \oplus x$ and $g_K(x) = L_K(x) \oplus x$, respectively. Let $A_{h,g}$ be a prfp-adversary against (h, g) which runs in time at most t and asks at most q queries. Then, there exists a prpp-adversary $A_{E,L}$ against (E, L) such that

$$\mathrm{Adv}^{\mathrm{prfp}}_{h,g}(A_{h,g}) \leq \mathrm{Adv}^{\mathrm{prpp}}_{E,L}(A_{E,L}) + \frac{q(q-1)}{2^{n+1}} \ ,$$

where $A_{E,L}$ runs in time at most $t + O(q)$ and asks at most q queries.

Let $\mathcal{B} = \{0,1\}^n$ and $\mathcal{B}^+ = \bigcup_{i=1} \mathcal{B}^i$. For the compression function h and the output function g, let $gh^* : \mathcal{B} \times \mathcal{B}^+ \to \mathcal{B}$ be a function family such that $gh^*(K, M)$ is defined for $K \in \mathcal{B}$ and $M \in \mathcal{B}^+$ as follows: Let $M = M^{(1)} \| \cdots \| M^{(N)}$ and $M^{(i)} \in \{0,1\}^n$ for $1 \le i \le N$. Then,

1. $a^{(0)} = K$,
2. if $N \ge 2$, then $a^{(i)} = h(a^{(i-1)}, M^{(i)})$ for $1 \le i \le N-1$,
3. $gh^*(K, M) = g(a^{(N-1)}, M^{(N)})$.

The following lemma is on the inner hashing. It says that, if (h, g) is a PRF pair, then gh^* is computationally almost universal. The proof is given in A.2.1.

Lemma 3 Let $h : \{0,1\}^\kappa \times \mathcal{B} \to \{0,1\}^\kappa$ and $g : \{0,1\}^\kappa \times \mathcal{B} \to \{0,1\}^\kappa$ be function families, and let A_{gh^*} be an au-adversary against gh^*. Suppose that A_{gh^*} outputs two messages with at most ℓ_1 and ℓ_2 blocks, respectively. Then, there exists a prfp-adversary $A_{h,g}$ against (h, g) such that

$$\mathrm{Adv}^{\mathrm{au}}_{gh^*}(A_{gh^*}) \le (\ell_1 + \ell_2 - 1)\,\mathrm{Adv}^{\mathrm{prfp}}_{h,g}(A_{h,g}) + \frac{1}{2^\kappa} \;,$$

where $A_{h,g}$ runs in time at most $O((\ell_1 + \ell_2) T_h + T_g)$ and makes at most 2 queries. T_h and T_g represent the time required to compute h and g, respectively.

Lemma 3 requires a PRF pair (h, g). However, it does not seem severe since adversaries are allowed to make only at most 2 queries to the oracles.

The following lemma is on the outer hashing. It says that, if the compression function and the output function are PRFs, then the outer-hashing function is also a PRF. The proof is easy and omitted.

Lemma 4 Let $h : \{0,1\}^\kappa \times \mathcal{B} \to \{0,1\}^\kappa$ and $g : \{0,1\}^\kappa \times \mathcal{B} \to \{0,1\}^\kappa$ be function families. Let $gh : \{0,1\}^\kappa \times \mathcal{B} \to \{0,1\}^\kappa$ be a function family defined by

$$gh(K, X) = g(h(K, X), 1 \| bin(\kappa + n)) \;,$$

where $K \in \{0,1\}^\kappa$, $X \in \mathcal{B}$ and $bin(\kappa + n)$ is the $(n-1)$-bit binary representation of $\kappa + n$. Let A_{gh} be a prf-adversary against gh that runs in time at most t and makes at most q queries. Then, there exist prf-adversaries A_h and A_g against h and g, respectively, such that

$$\mathrm{Adv}^{\mathrm{prf}}_{gh}(A_{gh}) \le \mathrm{Adv}^{\mathrm{prf}}_{h}(A_h) + q\,\mathrm{Adv}^{\mathrm{prf}}_{g}(A_g) \;,$$

where A_h runs in time at most $t + O(q\,T_g)$ and makes at most q queries, and A_g runs in time $t + O(q\,T_g)$ and makes at most 1 query.

The following lemma is Lemma 3.2 in [1]. It says that $f(K_\mathtt{o}, G(K_\mathtt{i}, \cdot))$ is a PRF if $f(K_\mathtt{o}, \cdot)$ is a PRF and $G(K_\mathtt{i}, \cdot)$ is computationally almost universal, where $K_\mathtt{o}$ and $K_\mathtt{i}$ are secret keys chosen uniformly and independently of each other.

Lemma 5 (Lemma 3.2 in [1]) Let $f : \{0,1\}^\tau \times \mathcal{B} \to \{0,1\}^\tau$ and $G : \{0,1\}^\kappa \times \mathcal{D} \to \mathcal{B}$ be function families. Let $fG : \{0,1\}^{\tau+\kappa} \times \mathcal{D} \to \{0,1\}^\tau$ be defined by $fG(K_{\mathtt{o}} \| K_{\mathtt{i}}, M) = f(K_{\mathtt{o}}, G(K_{\mathtt{i}}, M))$ for $K_{\mathtt{o}} \in \{0,1\}^\tau$, $K_{\mathtt{i}} \in \{0,1\}^\kappa$ and $M \in \mathcal{D}$. Let A_{fG} be a prf-adversary against fG that runs in time at most t and makes at most $q\,(\geq 2)$ queries each of whose lengths is at most d bits. Then, there exist a prf-adversary A_f against f and an au-adversary A_G against G such that

$$\mathrm{Adv}^{\mathrm{prf}}_{fG}(A_{fG}) \leq \mathrm{Adv}^{\mathrm{prf}}_{f}(A_f) + \frac{q(q-1)}{2}\mathrm{Adv}^{\mathrm{au}}_{G}(A_G) \enspace ,$$

where A_f runs in time at most t and makes at most q queries, and A_G runs in time $O(T_G(d))$ and the two messages it outputs have length at most d. $T_G(d)$ is the time to compute G on a d-bit input.

The following theorem is on the pseudorandomness of the NMAC-like function made from $\mathsf{HMAC}[E, L, IV](K, \cdot)$ by replacing the first calls of the compression function in inner and outer hashing with two secret keys chosen uniformly and independently of each other. The theorem states that the security of the function as a PRF is reduced to the security of the underlying block ciphers as a PRP pair. It directly follows from Lemmas 1 through 5.

Theorem 1 Let E and L be (n, n) block ciphers. Let $h : \mathcal{B} \times \mathcal{B} \to \mathcal{B}$ and $g : \mathcal{B} \times \mathcal{B} \to \mathcal{B}$ be functions such that $h_K(x) = E_K(x) \oplus x$ and $g_K(x) = L_K(x) \oplus x$. Let $ghgh^* : \mathcal{B}^2 \times \mathcal{B}^+ \to \mathcal{B}$ be defined by $ghgh^*(K_{\mathtt{o}} \| K_{\mathtt{i}}, M) = gh(K_{\mathtt{o}}, gh^*(K_{\mathtt{i}}, M))$ for $K_{\mathtt{o}}, K_{\mathtt{i}} \in \mathcal{B}$ and $M \in \mathcal{B}^+$. Let A_{ghgh^*} be a prf-adversary against $ghgh^*$ that runs in time at most t and makes at most $q\,(\geq 2)$ queries each of which has at most ℓ blocks. Then, there exist prp-adversaries A_E and A_L against E and L, and a prpp-adversary $A_{E,L}$ against (E, L) such that

$$\mathrm{Adv}^{\mathrm{prf}}_{ghgh^*}(A_{ghgh^*}) \leq \mathrm{Adv}^{\mathrm{prp}}_{E}(A_E) + q\,\mathrm{Adv}^{\mathrm{prp}}_{E}(A_L) + \ell\, q^2 \mathrm{Adv}^{\mathrm{prpp}}_{E,L}(A_{E,L}) + \frac{(\ell+1)\, q^2}{2^n} \enspace ,$$

where A_E runs in time at most $t + O(q\, T_L)$ and makes at most q queries, A_L runs in time at most $t + O(q\, T_L)$ and makes at most 1 query, and $A_{E,L}$ runs in time $O(\ell\, T_E + T_L)$ and makes at most 2 queries.

The following lemma says that, even if the secret key of a PRF is replaced by the output of a PRBG, the resulting function remains a PRF. The proof is easy and omitted.

Lemma 6 Let $\mu : \{0,1\}^\kappa \to \{0,1\}^{\kappa'}$ be a function and $F : \{0,1\}^{\kappa'} \times \mathcal{D} \to \mathcal{B}$ be a function family. Let $F\mu : \{0,1\}^\kappa \times \mathcal{D} \to \mathcal{B}$ be a function family defined by $F\mu(K, M) = F(\mu(K), M)$ for $K \in \{0,1\}^\kappa$ and $M \in \mathcal{D}$. Let $A_{F\mu}$ be a prf-adversary against $F\mu$ that runs in time at most t and makes at most q queries of length at most d bits. Then, there exist a prbg-adversary A_μ against μ and a prf-adversary A_F against F such that

$$\mathrm{Adv}^{\mathrm{prf}}_{F\mu}(A_{F\mu}) \leq \mathrm{Adv}^{\mathrm{prbg}}_{\mu}(A_\mu) + \mathrm{Adv}^{\mathrm{prf}}_{F}(A_F) \enspace ,$$

where A_μ runs in time at most $t + O(q\, T_F(d))$, and A_F runs in time t and makes at most q queries of length at most d bits.

Now, we can obtain the result on the pseudorandomness of $\mathsf{HMAC}[E, L, IV]$ simply by combining Theorem 1 and Lemma 6.

Corollary 1 Let E be an (n, n) block cipher. Let $\mu_E : \mathcal{B} \to \mathcal{B}^2$ be a function such that $\mu_E(K) = (E_{IV}(K_{\mathtt{op}}) \oplus K_{\mathtt{op}}) \| (E_{IV}(K_{\mathtt{ip}}) \oplus K_{\mathtt{ip}})$, where $K_{\mathtt{op}} = K \oplus \mathtt{opad}$ and $K_{\mathtt{ip}} = K \oplus \mathtt{ipad}$. Let A be a prf-adversary against $\mathsf{HMAC}[E, L, IV]$ that runs in time at most t and makes at most q (≥ 2) queries each of which has at most ℓ blocks. Then, there exist prp-adversaries A_E and A_L against E and L, a prpp-adversary $A_{E,L}$ against (E, L) and a prbg-adversary A_{μ_E} such that

$$\mathrm{Adv}^{\mathrm{prf}}_{\mathsf{HMAC}[E,L,IV]}(A) \leq \mathrm{Adv}^{\mathrm{prbg}}_{\mu_E}(A_{\mu_E}) + \mathrm{Adv}^{\mathrm{prp}}_{E}(A_E) + q\,\mathrm{Adv}^{\mathrm{prp}}_{L}(A_L) + \ell\, q^2 \mathrm{Adv}^{\mathrm{prpp}}_{E,L}(A_{E,L}) + \frac{(\ell+1)\, q^2}{2^n} ,$$

where A_{μ_E} runs in time at most $t + O(q\,\ell\, T_E)$, A_E runs in time at most $t + O(q\, T_L)$ and makes at most q queries, A_L runs in time at most $t + O(q\, T_L)$ and makes at most 1 query, and $A_{E,L}$ runs in time $O(\ell\, T_E + T_L)$ and makes at most 2 queries.

A.2.1 Proof of Lemma 3

For $M \in \mathcal{B}^+$, let $|M|_n = |M|/n$. For $M_1, M_2 \in \mathcal{B}^+$, let $\mathrm{LCP}(M_1, M_2) = \lfloor |M_\star|/n \rfloor$, where $M_\star$ represents the longest common prefix of M_1 and M_2.

In the following, let M_1 and M_2 be distinct elements in $\mathcal{B}^+$. Let $m_1 = |M_1|_n$ and $m_2 = |M_2|_n$. Without loss of generality, we can assume that $m_1 \leq m_2$. Let $p = \min\{\mathrm{LCP}(M_1, M_2), m_1 - 1\}$.

This proof uses the game G and the adversary A given in Figure 44.

Claim 1 Suppose that $1 \leq l \leq m_1 + m_2 - p - 1$. Then,

$$\Pr[A^{\rho,\rho'}(M_1, M_2, l) = 1 \mid \rho, \rho' \xleftarrow{\$} \mathsf{Func}(\mathcal{B}, \{0,1\}^\kappa)] = \Pr[G(M_1, M_2, l) = 1]$$
$$\Pr[A^{h_K, g_K}(M_1, M_2, l) = 1 \mid K \xleftarrow{\$} \{0,1\}^\kappa] = \Pr[G(M_1, M_2, l-1) = 1] .$$

Proof. It is first shown that $A^{\rho,\rho'}(M_1, M_2, l)$ is equivalent to $G(M_1, M_2, l)$.

If $l \leq p\, (\leq m_1 - 1)$, then, in $A^{\rho,\rho'}$, $a_1[l] \leftarrow \rho(M_1[l])$ and $a_2[l] \leftarrow a_1[l]$. $a_1[l] \leftarrow \rho(M_1[l])$ is equivalent to $a_1[l] \xleftarrow{\$} \{0,1\}^\kappa$ since ρ is random.

If $l = p + 1$, then $p + 1 \leq m_1$ and

$$a_1[p+1] \leftarrow \begin{cases} \rho(M_1[p+1]) & \text{if } p+1 \leq m_1 - 1 \\ \rho'(M_1[p+1]) & \text{if } p+1 = m_1 \end{cases}$$
$$a_2[p+1] \leftarrow \begin{cases} \rho(M_2[p+1]) & \text{if } p+1 \leq m_2 - 1 \\ \rho'(M_2[p+1]) & \text{if } p+1 = m_2 \end{cases} .$$

If $p + 1 \leq m_1 - 1$, then $p + 1 \leq m_2 - 1$ and $p = \mathrm{LCP}(M_1, M_2)$. Thus, $a_1[p+1] \leftarrow \rho(M_1[p+1])$, $a_2[p+1] \leftarrow \rho(M_2[p+1])$, and $M_1[p+1] \neq M_2[p+1]$. If $p + 1 = m_1$ and $p + 1 \leq m_2 - 1$, then $a_1[p+1] \leftarrow \rho'(M_1[p+1])$ and $a_2[p+1] \leftarrow \rho(M_2[p+1])$. If $p + 1 = m_1$ and $p + 1 = m_2$, then $m_1 = m_2$ and $p = \mathrm{LCP}(M_1, M_2)$. Otherwise, $\mathrm{LCP}(M_1, M_2) = m_1 = m_2$, and $M_1 = M_2$, which causes a contradiction. Thus, $a_1[p+1] \leftarrow \rho'(M_1[p+1])$, $a_2[p+1] \leftarrow \rho'(M_2[p+1])$, and $M_1[p+1] \neq M_2[p+1]$. In any case, $a_1[p+1]$ and $a_2[p+1]$ are selected from $\{0,1\}^\kappa$ uniformly and independently of each other.

If $p+2 \leq l \leq m_1$, then $a_1[l] \leftarrow \rho(M_1[l])$ or $\rho'(M_1[l])$, and $a_2[p+1] \stackrel{\$}{\leftarrow} \{0,1\}^\kappa$. Thus, $a_1[l]$ and $a_2[p+1]$ are selected from $\{0,1\}^\kappa$ uniformly and independently of each other.

If $l \geq m_1+1$, then $a_1[m_1] \stackrel{\$}{\leftarrow} \{0,1\}^\kappa$, and $a_2[k] \leftarrow \rho(M_2[k])$ or $\rho'(M_2[k])$. Thus, $a_1[m_1]$ and $a_2[k]$ are selected from $\{0,1\}^\kappa$ uniformly and independently of each other.

It is concluded from these observations that the first equation of the claim holds.

It is shown below that the second equation holds. The proof uses the game transformations.

$G_1(M_1, M_2, l)$ given in Figure 45 is obtained simply by substituting $l-1$ to l of $G(M_1, M_2, l)$. Thus, $\Pr[G(M_1, M_2, l-1) = 1] = \Pr[G_1(M_1, M_2, l) = 1]$.

The equivalence between G_1 and G_2 given in Figure 45 is confirmed as follows. It is easy to see that the lines 506 through 509 are equivalent to the lines 608 and 609. For $p+2 \leq l \leq m_1$, the lines 513 through 521 are equivalent to the lines 619 through 621. If $l = m_1+1$, then $k \leftarrow p+1$ in G_2. Thus, the lines 519 through 524 are equivalent to the lines 622 through 624 for $m_1+1 \leq l \leq m_1+m_2-p-1$. The other parts of G_2 are copied from G_1. Thus, $\Pr[G_1(M_1, M_2, l) = 1] = \Pr[G_2(M_1, M_2, l) = 1]$.

The equivalence between G_2 and G_3 given in Figure 46 is shown below. In G_3, K in the lines 702 and 724 is sampled from $\{0,1\}^\kappa$ under the uniform distribution at the line 699. Notice that K is used either in 702 or in 724 exclusively. It is easy to see that the lines 610 through 612 are equivalent to the lines 710 through 715. The other parts of G_3 are copied from G_2. Thus, $\Pr[G_2(M_1, M_2, l) = 1] = \Pr[G_3(M_1, M_2, l) = 1]$.

The equivalence between G_3 and A^{h_K, g_K} given in Figure 46 is shown below. The lines 701 through 705 are equivalent to the lines 801 through 807. For $1 \leq l \leq p\,(\leq m_1 - 1)$, $a_2[l-1] \leftarrow a_1[l-1] = K$ at 712 in G_3, while $a_2[l] \leftarrow a_1[l] = h(K, M_1[l])$ at 812 in A^{h_K, g_K}. The evaluation of $a_2[l]$ is delayed until the line 729 in G_3. If $l = p+1$, then the evaluation of $a_2[l]$ is delayed until the line 729 or 730 in G_3. Similarly, if $m_1+1 \leq l \leq m_1+m_2-p-1$, then the evaluation of $a_2[l-m_1+p+1]$ is delayed until the line 729 or 730 in G_3. Thus, $\Pr[G_3(M_1, M_2, l) = 1] = \Pr[A^{h_K, g_K}(M_1, M_2, l) = 1 \,|\, K \stackrel{\$}{\leftarrow} \{0,1\}^\kappa]$.

From these observations, it is concluded that the second equation of the claim holds. □

Let $P^{\mathrm{col}}_{gh^*}(M_1, M_2) = \Pr[gh^*(K, M_1) = gh^*(K, M_2) \,|\, K \stackrel{\$}{\leftarrow} \{0,1\}^\kappa]$.

Claim 2 Let $m = m_1 + m_2 - p - 1$. Then,

$$\Pr[G(M_1, M_2, m) = 1] = \frac{1}{2^\kappa}$$
$$\Pr[G(M_1, M_2, 0) = 1] = P^{\mathrm{col}}_{gh^*}(M_1, M_2)\ .$$

Proof. If G is run with the argument (M_1, M_2, m), then $a_1[m_1]$ is chosen from $\{0,1\}^\kappa$ uniformly at random. Thus, $\Pr[G(M_1, M_2, m) = 1] = 1/2^\kappa$.

On the other hand, suppose that G is run with the argument $(M_1, M_2, 0)$. Then, $a_1[m_1] = gh^*(a_1[0], M_1)$, $a_2[m_2] = gh^*(a_2[0], M_2)$, and $a_2[0] = a_1[0] \xleftarrow{\$} \{0,1\}^\kappa$. Thus, $\Pr[G(M_1, M_2, 0) = 1] = P^{\mathrm{col}}_{gh^*}(M_1, M_2)$. □

Let A_1 be a prfp-adversary against (h, g) such that, for given M_1, M_2,

1. it first selects l from $\{1, 2, \ldots, m_1 + m_2 - p - 1\}$ uniformly at random, and
2. invokes $A^{u,v}$ with (M_1, M_2, l), and outputs $A^{u,v}(M_1, M_2, l)$.

Claim 3 Let $m = m_1 + m_2 - p - 1$. Then,

$$\mathrm{Adv}^{\mathrm{prfp}}_{h,g}(A_1) = \frac{1}{m}\left|P^{\mathrm{col}}_{gh^*}(M_1, M_2) - \frac{1}{2^\kappa}\right| .$$

Proof. From the definition,

$$\mathrm{Adv}^{\mathrm{prfp}}_{h,g}(A_1) = \left|\Pr[A_1^{h_K, g_K} = 1 \,|\, K \xleftarrow{\$} \{0,1\}^\kappa] - \Pr[A_1^{\rho,\rho'} = 1 \,|\, \rho, \rho' \xleftarrow{\$} \mathsf{Func}(\mathcal{B}, \{0,1\}^\kappa)]\right| .$$

On the other hand,

$$\begin{aligned}\Pr[A_1^{h_K, g_K} = 1 \,|\, K \xleftarrow{\$} \{0,1\}^\kappa] &= \sum_{i=1}^{m} \Pr[l = i \wedge A_1^{h_K, g_K} = 1 \,|\, K \xleftarrow{\$} \{0,1\}^\kappa] \\ &= \frac{1}{m}\sum_{i=1}^{m} \Pr[A^{h_K, g_K}(M_1, M_2, i) = 1 \,|\, K \xleftarrow{\$} \{0,1\}^\kappa] \\ &= \frac{1}{m}\sum_{i=1}^{m} \Pr[G(M_1, M_2, i-1) = 1] .\end{aligned}$$

Similarly,

$$\Pr[A_1^{\rho,\rho'} = 1 \,|\, \rho, \rho' \xleftarrow{\$} \mathsf{Func}(\mathcal{B}, \{0,1\}^\kappa)] = \frac{1}{m}\sum_{i=1}^{m} \Pr[G(M_1, M_2, i) = 1] .$$

Thus,

$$\begin{aligned}\mathrm{Adv}^{\mathrm{prfp}}_{h,g}(A_1) &= \left|\frac{1}{m}\Pr[G(M_1, M_2, 0) = 1] - \frac{1}{m}\Pr[G(M_1, M_2, m) = 1]\right| \\ &= \frac{1}{m}\left|P^{\mathrm{col}}_{gh^*}(M_1, M_2) - \frac{1}{2^\kappa}\right| .\end{aligned}$$

□

Let A_2 be a prfp-adversary against (h, g) such that

1. $M_1, M_2 \leftarrow A_{gh^*}$,
2. invokes $A_1^{u,v}$ with (M_1, M_2), and outputs $A_1^{u,v}(M_1, M_2)$.

Claim 4

$$\mathrm{Adv}^{\mathrm{au}}_{gh^*}(A_{gh^*}) \le (\ell_1+\ell_2-1)\mathrm{Adv}^{\mathrm{prfp}}_{h,g}(A_2) + \frac{1}{2^\kappa} \ .$$

Proof. Notice that

$$\mathrm{Adv}^{\mathrm{au}}_{gh^*}(A_{gh^*}) = \sum_{M_1,M_2} P^{\mathrm{col}}_{gh^*}(M_1,M_2)\, P_{A_{gh^*}}(M_1,M_2) \ ,$$

where $P_{A_{gh^*}}(M_1, M_2)$ is the probability that A_{gh^*} outputs M_1, M_2. From the previous claim,

$$\begin{aligned}\mathrm{Adv}^{\mathrm{au}}_{gh^*}(A_{gh^*}) &= \sum_{M_1,M_2} P^{\mathrm{col}}_{gh^*}(M_1,M_2)\, P_{A_{gh^*}}(M_1,M_2) \\ &\le \sum_{M_1,M_2} \left((\ell_1+\ell_2-1)\mathrm{Adv}^{\mathrm{prfp}}_{h,g}(A_1) + \frac{1}{2^\kappa}\right) P_{A_{gh^*}}(M_1,M_2) \\ &= (\ell_1+\ell_2-1)\mathrm{Adv}^{\mathrm{prfp}}_{h,g}(A_2) + \frac{1}{2^\kappa} \ .\end{aligned}$$

□

The time complexity of A_2 depends on that of A_{gh^*}. Notice that there exist some $\tilde{M}_1, \tilde{M}_2 \in \mathcal{B}^+$ such that $\mathrm{Adv}^{\mathrm{prfp}}_{h,g}(A_2) \le \mathrm{Adv}^{\mathrm{prfp}}_{h,g}(A_1(\tilde{M}_1, \tilde{M}_2))$. Let $A_{h,g}$ be the prf-adversary that has $\tilde{M}_1, \tilde{M}_2$ as a part of its code and runs $A_1^{u,v}(\tilde{M}_1, \tilde{M}_2)$. Then,

$$\mathrm{Adv}^{\mathrm{au}}_{gh^*}(A_{gh^*}) \le (\ell_1+\ell_2-1)\mathrm{Adv}^{\mathrm{prfp}}_{h,g}(A_{h,g}) + \frac{1}{2^\kappa} \ .$$

$A_{h,g}$ runs in time $O((\ell_1+\ell_2)T_h + T_g)$ and makes at most 2 queries.

B Indifferentiability from Random Oracle

B.1 Definitions

B.1.1 Indifferentiability

The notion of indifferentiability is introduced by Maurer et al. [23] as a generalized notion of indistinguishability. Then, it is tailored to security analysis of hash functions by Coron et al. [9].

Let C be an algorithm with oracle access to ideal primitives $\mathcal{F}_1, \ldots, \mathcal{F}_d$. In the setting of this document, C is an algorithm to construct a hash function using $\mathcal{F}_1, \ldots, \mathcal{F}_d$ with fixed input length (FIL). Let $\mathcal{H}$ be the variable-input-length (VIL) random oracle and $S_1, \ldots, S_d$ be simulators which have oracle access to $\mathcal{H}$. $S_1^{\mathcal{H}}, \ldots, S_d^{\mathcal{H}}$ try to behave like $\mathcal{F}_1, \ldots, \mathcal{F}_d$ in order to convince an adversary that $\mathcal{H}$ is $C^{\mathcal{F}_1,\ldots,\mathcal{F}_d}$. Let A be an adversary with access to oracles. The indiff-advantage of A against C with respect to $S_1, \ldots, S_d$ is given by

$$\mathrm{Adv}^{\mathrm{indiff}}_{C,S_1,\ldots,S_d}(A) = \left|\Pr[A^{C^{\mathcal{F}_1,\ldots,\mathcal{F}_d},\mathcal{F}_1,\ldots,\mathcal{F}_d} = 1] - \Pr[A^{\mathcal{H},S_1^{\mathcal{H}},\ldots,S_d^{\mathcal{H}}} = 1]\right| \ ,$$

where the probabilities are taken over the coin tosses by A, C and $S_1, \ldots, S_d$ and the distributions of ideal primitives. $C^{\mathcal{F}_1,\ldots,\mathcal{F}_d}$ is said to be indifferentiable from $\mathcal{H}$ if there exist efficient simulators $S_1^{\mathcal{H}}, \ldots, S_d^{\mathcal{H}}$ such that $\mathrm{Adv}^{\mathrm{indiff}}_{C,S_1,\ldots,S_d}(A)$ is negligible for any efficient A.

Game $G(M_1, M_2, l)$:

$p \leftarrow \min\{\mathrm{LCP}(M_1, M_2), m_1 - 1\}$
if $0 \le l \le m_1 - 1$ **then**
  $a_1[l] \xleftarrow{\$} \{0,1\}^\kappa$
  for $i = l + 1$ to $m_1 - 1$ **do**
    $a_1[i] \leftarrow h(a_1[i-1], M_1[i])$
  $a_1[m_1] \leftarrow g(a_1[m_1 - 1], M_1[m_1])$
if $l = m_1$ **then**
  $a_1[l] \xleftarrow{\$} \{0,1\}^\kappa$
if $m_1 + 1 \le l \le m_1 + m_2 - p - 1$ **then**
  $a_1[m_1] \xleftarrow{\$} \{0,1\}^\kappa$
if $0 \le l \le p$ **then**
  $k \leftarrow l$
  $a_2[k] \leftarrow a_1[k]$
if $l = p + 1$ **then**
  $k \leftarrow p + 1$
  $a_2[k] \xleftarrow{\$} \{0,1\}^\kappa$
if $p + 2 \le l \le m_1$ **then**
  $k \leftarrow p + 1$
  $a_2[k] \xleftarrow{\$} \{0,1\}^\kappa$
if $m_1 + 1 \le l \le m_1 + m_2 - p - 1$ **then**
  $k \leftarrow l - m_1 + p + 1$
  $a_2[k] \xleftarrow{\$} \{0,1\}^\kappa$
for $i = k + 1$ to $m_2 - 1$ **do**
  $a_2[i] \leftarrow h(a_2[i-1], M_2[i])$
$a_2[m_2] \leftarrow g(a_2[m_2 - 1], M_2[m_2])$
if $a_1[m_1] = a_2[m_2]$ **then**
  return 1
else
  return 0

Adversary $A^{u,v}(M_1, M_2, l)$:

$p \leftarrow \min\{\mathrm{LCP}(M_1, M_2), m_1 - 1\}$
if $1 \le l \le m_1 - 1$ **then**
  $a_1[l] \leftarrow u(M_1[l])$
  for $i = l + 1$ to $m_1 - 1$ **do**
    $a_1[i] \leftarrow h(a_1[i-1], M_1[i])$
  $a_1[m_1] \leftarrow g(a_1[m_1 - 1], M_1[m_1])$
if $l = m_1$ **then**
  $a_1[l] \leftarrow v(M_1[l])$
if $m_1 + 1 \le l \le m_1 + m_2 - p - 1$ **then**
  $a_1[m_1] \xleftarrow{\$} \{0,1\}^\kappa$
if $1 \le l \le p$ **then**
  $k \leftarrow l$
  $a_2[k] \leftarrow a_1[k]$
if $l = p + 1$ **then**
  $k \leftarrow p + 1$
  if $m_2 = k$ **then**
    $a_2[k] \leftarrow v(M_2[k])$
  else
    $a_2[k] \leftarrow u(M_2[k])$
if $p + 2 \le l \le m_1$ **then**
  $k \leftarrow p + 1$
  $a_2[k] \xleftarrow{\$} \{0,1\}^\kappa$
if $m_1 + 1 \le l \le m_1 + m_2 - p - 1$ **then**
  $k \leftarrow l - m_1 + p + 1$
  if $m_2 = k$ **then**
    $a_2[k] \leftarrow v(M_2[k])$
  else
    $a_2[k] \leftarrow u(M_2[k])$
for $i = k + 1$ to $m_2 - 1$ **do**
  $a_2[i] \leftarrow h(a_2[i-1], M_2[i])$
$a_2[m_2] \leftarrow g(a_2[m_2 - 1], M_2[m_2])$
if $a_1[m_1] = a_2[m_2]$ **then**
  return 1
else
  return 0

Figure 44: Pseudocodes for the game and the adversary.

Game $G_1(M_1, M_2, l)$:

```
p ← min{LCP(M1, M2), m1 − 1}
if 1 ≤ l ≤ m1 then
    a1[l − 1] ←$ {0, 1}^κ
    for i = l to m1 − 1 do
        a1[i] ← h(a1[i − 1], M1[i])
    a1[m1] ← g(a1[m1 − 1], M1[m1])
if l = m1 + 1 then
    a1[m1] ←$ {0, 1}^κ
if m1 + 2 ≤ l ≤ m1 + m2 − p − 1 then
    a1[m1] ←$ {0, 1}^κ
if 1 ≤ l ≤ p + 1 then
    k ← l − 1
    a2[k] ← a1[k]
if l = p + 2 then
    k ← p + 1
    a2[k] ←$ {0, 1}^κ
if p + 3 ≤ l ≤ m1 + 1 then
    k ← p + 1
    a2[k] ←$ {0, 1}^κ
if m1 + 2 ≤ l ≤ m1 + m2 − p − 1 then
    k ← l − m1 + p
    a2[k] ←$ {0, 1}^κ
for i = k + 1 to m2 − 1 do
    a2[i] ← h(a2[i − 1], M2[i])
a2[m2] ← g(a2[m2 − 1], M2[m2])
if a1[m1] = a2[m2] then
    return 1
else
    return 0
```

Game $G_2(M_1, M_2, l)$:

```
p ← min{LCP(M1, M2), m1 − 1}
if 1 ≤ l ≤ m1 then
    a1[l − 1] ←$ {0, 1}^κ
    for i = l to m1 − 1 do
        a1[i] ← h(a1[i − 1], M1[i])
    a1[m1] ← g(a1[m1 − 1], M1[m1])
if m1 + 1 ≤ l ≤ m1 + m2 − p − 1 then
    a1[m1] ←$ {0, 1}^κ
if 1 ≤ l ≤ p + 1 then
    k ← l − 1
    a2[k] ← a1[k]
if p + 2 ≤ l ≤ m1 then
    k ← p + 1
    a2[k] ←$ {0, 1}^κ
if m1 + 1 ≤ l ≤ m1 + m2 − p − 1 then
    k ← l − m1 + p
    a2[k] ←$ {0, 1}^κ
for i = k + 1 to m2 − 1 do
    a2[i] ← h(a2[i − 1], M2[i])
a2[m2] ← g(a2[m2 − 1], M2[m2])
if a1[m1] = a2[m2] then
    return 1
else
    return 0
```

Figure 45: Pseudocodes for the games G_1 and G_2.

Game $G_3(M_1, M_2, l)$:

$K \xleftarrow{\$} \{0,1\}^\kappa$
$p \leftarrow \min\{\mathrm{LCP}(M_1, M_2), m_1 - 1\}$
if $1 \leq l \leq m_1$ **then**
  $a_1[l-1] \leftarrow K$
  for $i = l$ to $m_1 - 1$ **do**
    $a_1[i] \leftarrow h(a_1[i-1], M_1[i])$
  $a_1[m_1] \leftarrow g(a_1[m_1 - 1], M_1[m_1])$
if $m_1 + 1 \leq l \leq m_1 + m_2 - p - 1$ **then**
  $a_1[m_1] \xleftarrow{\$} \{0,1\}^\kappa$
if $1 \leq l \leq p$ **then**
  $k \leftarrow l - 1$
  $a_2[k] \leftarrow a_1[k]$
if $l = p + 1$ **then**
  $k \leftarrow p$
  $a_2[k] \leftarrow a_1[k]$
if $p + 2 \leq l \leq m_1$ **then**
  $k \leftarrow p + 1$
  $a_2[k] \xleftarrow{\$} \{0,1\}^\kappa$
if $m_1 + 1 \leq l \leq m_1 + m_2 - p - 1$ **then**
  $k \leftarrow l - m_1 + p$
  $a_2[k] \leftarrow K$
for $i = k + 1$ to $m_2 - 1$ **do**
  $a_2[i] \leftarrow h(a_2[i-1], M_2[i])$
$a_2[m_2] \leftarrow g(a_2[m_2 - 1], M_2[m_2])$
if $a_1[m_1] = a_2[m_2]$ **then**
  return 1
else
  return 0

Adversary $A^{h_K, g_K}(M_1, M_2, l)$:

$p \leftarrow \min\{\mathrm{LCP}(M_1, M_2), m_1 - 1\}$
if $1 \leq l \leq m_1 - 1$ **then**
  $a_1[l] \leftarrow h(K, M_1[l])$
  for $i = l + 1$ to $m_1 - 1$ **do**
    $a_1[i] \leftarrow h(a_1[i-1], M_1[i])$
  $a_1[m_1] \leftarrow g(a_1[m_1 - 1], M_1[m_1])$
if $l = m_1$ **then**
  $a_1[l] \leftarrow g(K, M_1[l])$
if $m_1 + 1 \leq l \leq m_1 + m_2 - p - 1$ **then**
  $a_1[m_1] \xleftarrow{\$} \{0,1\}^\kappa$
if $1 \leq l \leq p$ **then**
  $k \leftarrow l$
  $a_2[k] \leftarrow a_1[k]$
if $l = p + 1$ **then**
  $k \leftarrow p + 1$
  if $m_2 = k$ **then**
    $a_2[k] \leftarrow g(K, M_2[k])$
  else
    $a_2[k] \leftarrow h(K, M_2[k])$
if $p + 2 \leq l \leq m_1$ **then**
  $k \leftarrow p + 1$
  $a_2[k] \xleftarrow{\$} \{0,1\}^\kappa$
if $m_1 + 1 \leq l \leq m_1 + m_2 - p - 1$ **then**
  $k \leftarrow l - m_1 + p + 1$
  if $m_2 = k$ **then**
    $a_2[k] \leftarrow g(K, M_2[k])$
  else
    $a_2[k] \leftarrow h(K, M_2[k])$
for $i = k + 1$ to $m_2 - 1$ **do**
  $a_2[i] \leftarrow h(a_2[i-1], M_2[i])$
$a_2[m_2] \leftarrow g(a_2[m_2 - 1], M_2[m_2])$
if $a_1[m_1] = a_2[m_2]$ **then**
  return 1
else
  return 0

Figure 46: Pseudocodes for the game G_3 and the adversary A^{h_K, g_K}.

B.1.2 Ideal Cipher Model

A block cipher with block length n and key length κ is called an (n, κ) block cipher. Let $E : \{0,1\}^\kappa \times \{0,1\}^n \to \{0,1\}^n$ be an (n, κ) block cipher. Then, $E(K, \cdot) = E_K(\cdot)$ is a permutation for every $K \in \{0,1\}^\kappa$. An (n, κ) block cipher E is called an ideal cipher if E_K is a truly random permutation for every K.

The lazy evaluation of an ideal cipher is described as follows. The encryption oracle E receives a pair of a key and a plaintext as a query, and returns a randomly selected ciphertext. On the other hand, the decryption oracle D receives a pair of a key and a ciphertext as a query, and returns a randomly selected plaintext. The oracles E and D share a table of triplets of keys, plaintexts and ciphertexts, which are produced by the queries and the corresponding replies. Referring to the table, they select a reply to a new query under the restriction that E_K is a permutation for every K.

B.2 Analysis

In this section, we show that Lesamnta is indifferentiable from the VIL random oracle in the ideal cipher model. The following theorem states the indifferentiability of Lesamnta in the ideal cipher model. In the remaining part of this section, L is denoted by E', and the decryption functions of E and E' are denoted by D and D', respectively.

Theorem 2 Let E and E' be (n, n) block ciphers. Let A be an adversary that asks at most $q_{\mathcal{H}}$ queries to the VIL oracle, $q_{\mathcal{E}}$ ($q_{\mathcal{D}}$) queries to the encryption (decryption) oracle for E, and $q_{\mathcal{E}'}$ ($q_{\mathcal{D}'}$) queries to the encryption (decryption) oracle for E'. Let ℓ be the maximum number of message blocks in a VIL query. Suppose that $\ell q_{\mathcal{H}} + q_{\mathcal{E}} + q_{\mathcal{D}} + q_{\mathcal{E}'} + q_{\mathcal{D}'} \leq 2^{n-1}$ and $\ell q_{\mathcal{H}} \geq 1$, $q_{\mathcal{E}} \geq 1$, $q_{\mathcal{D}} \geq 1$, $q_{\mathcal{E}'} \geq 1$, $q_{\mathcal{D}'} \geq 1$. Then, for Lesamnta, in the ideal cipher model,

$$\mathrm{Adv}^{\mathrm{indiff}}_{\mathrm{Lesamnta}, S_E, S_D, S_{E'}, S_{D'}}(A) \leq \frac{3\,(\ell q_{\mathcal{H}} + q_{\mathcal{E}} + q_{\mathcal{D}} + q_{\mathcal{E}'} + q_{\mathcal{D}'})^2}{2^n} ,$$

where the simulators S_E, S_D and $S_{E'}, S_{D'}$ are given in Figure 47. S_E (S_D) is a simulator of the encryption (decryption) oracle for E. $S_{E'}$ ($S_{D'}$) is a simulator of the encryption (decryption) oracle for E'. S_E runs in time $O(q_{\mathcal{E}}(q_{\mathcal{E}} + q_{\mathcal{D}}))$. S_D runs in time $O(q_{\mathcal{D}}(q_{\mathcal{E}} + q_{\mathcal{D}}))$. $S_{E'}$ makes at most $2\,q_{\mathcal{E}'}$ queries and runs in time $O(q_{\mathcal{E}'}(q_{\mathcal{E}} + q_{\mathcal{D}}))$. $S_{D'}$ makes at most $2\,q_{\mathcal{D}'}$ queries and runs in time $O(q_{\mathcal{D}'}(q_{\mathcal{E}} + q_{\mathcal{D}}))$.

The simulators simulate the ideal ciphers using lazy evaluation. In Figure 47, $\mathcal{P}(s)$ and $C(s)$ ($\mathcal{P}'(s)$ and $C'(s)$) represent the set of plaintexts and that of ciphertexts for E (E'), respectively, which are available for the reply to the current query with the key s. They are initially $\{0,1\}^n$, and their elements are deleted one by one as the simulation proceeds.

Let (s_i, x_i, y_i) be the triplet determined by the i-th query of the adversary and the corresponding answer, where $E_{s_i}(x_i) = y_i$. Then, for the MMO compression function, s_i is a chaining variable, and x_i is a message block. The triplets naturally defines a graph which initially consists of a single node labeled by the initial value IV and grows as the simulation proceeds. (s_i, x_i, y_i) adds two nodes

Initialize:

$\mathcal{V} \leftarrow \emptyset$
$\mathcal{T} \leftarrow \{IV\}$
$\mathcal{P}(s) \leftarrow \{0,1\}^n$
$C(s) \leftarrow \{0,1\}^n$
$\mathcal{P}'(s) \leftarrow \{0,1\}^n$
$C'(s) \leftarrow \{0,1\}^n$

Interface $\mathcal{E}(s,x)$:

if $s \in \mathcal{T}$ **then**
    $E_s(x) \xleftarrow{\$} C(s) \setminus \mathcal{S}_{\text{bad}}$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{E_s(x) \oplus x\}$
else
    $E_s(x) \xleftarrow{\$} C(s)$
$\mathcal{V} \leftarrow \mathcal{V} \cup \{s\}$
$\mathcal{P}(s) \leftarrow \mathcal{P}(s) \setminus \{x\}$
$C(s) \leftarrow C(s) \setminus \{E_s(x)\}$
return $E_s(x)$

Interface $\mathcal{D}(s,x)$:

if $s \in \mathcal{T}$ **then**
    $D_s(x) \xleftarrow{\$} \mathcal{P}(s) \setminus \mathcal{S}_{\text{bad}}$
    $\mathcal{T} \leftarrow \mathcal{T} \cup \{D_s(x) \oplus x\}$
else
    $D_s(x) \xleftarrow{\$} \mathcal{P}(s)$
$\mathcal{V} \leftarrow \mathcal{V} \cup \{s\}$
$\mathcal{P}(s) \leftarrow \mathcal{P}(s) \setminus \{D_s(x)\}$
$C(s) \leftarrow C(s) \setminus \{x\}$
return $D_s(x)$

Interface $\mathcal{E}'(s,x)$:

if $s \in \mathcal{T}$ **then**
    $\tilde{M} \leftarrow \mathsf{getnode}(s)$
    if $x \in \{lb(M^{(0)}), lb(M^{(1)})\}$ **then**
        if $x = lb(M^{(0)})$ **then**
            $E'_s(x) \leftarrow H(M^{(0)}) \oplus lb(M^{(0)})$
        else
            $E'_s(x) \leftarrow H(M^{(1)}) \oplus lb(M^{(1)})$
        if $E'_s(x) \notin C'(s)$ **then**
            return `fail`
    else
        $E'_s(x) \xleftarrow{\$} C'(s)$
else
    $E'_s(x) \xleftarrow{\$} C'(s)$
$\mathcal{V} \leftarrow \mathcal{V} \cup \{s\}$
$\mathcal{P}'(s) \leftarrow \mathcal{P}'(s) \setminus \{x\}$
$C'(s) \leftarrow C'(s) \setminus \{E'_s(x)\}$
return $E'_s(x)$

Interface $\mathcal{D}'(s,x)$:

if $s \in \mathcal{T}$ **then**
    $\tilde{M} \leftarrow \mathsf{getnode}(s)$
    if $x \in \{H(M^{(i)}) \oplus lb(M^{(i)}) \mid i = 0,1\}$ **then**
        if $x = H(M^{(0)}) \oplus lb(M^{(0)})$ **then**
            $D'_s(x) \leftarrow lb(M^{(0)})$
        else if $x = H(M^{(1)}) \oplus lb(M^{(1)})$ **then**
            $D'_s(x) \leftarrow lb(M^{(1)})$
    else
        $D'_s(x) \xleftarrow{\$} \mathcal{P}'(s) \setminus \{lb(M^{(0)}), lb(M^{(1)})\}$
else
    $D'_s(x) \xleftarrow{\$} \mathcal{P}'(s)$
$\mathcal{V} \leftarrow \mathcal{V} \cup \{s\}$
$\mathcal{P}'(s) \leftarrow \mathcal{P}'(s) \setminus \{D'_s(x)\}$
$C'(s) \leftarrow C'(s) \setminus \{x\}$
return $D'_s(x)$

Figure 47: Pseudocode for the simulators S_E, S_D and $S_{E'}, S_{D'}$. H represents the VIL random oracle. $\mathcal{S}_{\text{bad}} = \{y \mid y \in \{0,1\}^n \wedge x \oplus y \in \mathcal{V} \cup \mathcal{T}\}$. $\mathsf{pad}(M^{(0)}) = \tilde{M} \| lb(M^{(0)})$, and $\mathsf{pad}(M^{(1)}) = \tilde{M} \| lb(M^{(1)})$. $\tilde{M} = M^{(0)} \| 10^l$ $(0 \le l \le n-2)$ and $lb(M^{(0)}) = 0 \| bin(|M^{(0)}|)$. $\tilde{M} = M^{(1)}$ and $lb(M^{(1)}) = 1 \| bin(|M^{(1)}|)$.

labeled by s_i and $z_i = x_i \oplus y_i$, and an edge labeled by x_i from s_i to z_i. The additions avoid duplication of nodes with the same labels.

The simulators use two sets $\mathcal{V}$ and $\mathcal{T}$. $\mathcal{V}$ keeps all the labels of the nodes with outgoing edge(s) in the graph. $\mathcal{T}$ keeps all the labels of the nodes reachable from the node labeled by IV following the paths. The procedure $\mathsf{getnode}(s)$ returns the sequence of labels of the edges on the path from the node labeled by IV to the node labeled by s.

S_E and S_D select a reply not simply from $\mathcal{C}(s)$ and $\mathcal{P}(s)$ but from $\mathcal{C}(s) \setminus \boldsymbol{S}_{\text{bad}}$ and $\mathcal{P}(s) \setminus \boldsymbol{S}_{\text{bad}}$. It prevents most of the events which make the simulators fail. For example, since $\{y \mid x \oplus y \in \mathcal{T}\} \subset \boldsymbol{S}_{\text{bad}}$, every node in $\mathcal{T}$ has a unique path from the node labeled by IV. Thus, $\tilde{M}$ is uniquely identified at the lines 401 and 501.

The most critical work of the simulators is to reply to the decryption query related to the output function in Lesamnta for some input M. Let (s, x) be the query to the simulator $S_{D'}$. In order to reply to (s, x) properly, $S_{D'}$ has to ask M to the VIL random oracle H and return $H(M) \oplus x$. Owing to the padding scheme pad, there exist only two possibilities for M, $M^{(0)}$ and $M^{(1)}$, which correspond to the message blocks $\tilde{M}$ fed to the compression functions before the output function. Thus, $S_{D'}$ can accomplish the work.

C PRF Modes Using Lesamnta

Some notations and definitions used in the remaining part are given in Appendix A.

C.1 Pseudorandomness with Multi-Oracle

Let $\mathcal{B} = \{0, 1\}^n$. Let A be an adversary with acces to m pairs of oracles $u_1, u'_1, u_2, u'_2, \ldots, u_m, u'_m$. The m-prfp-advantage of A against (h, g) is defined as follows:

$$\mathrm{Adv}^{m\text{-prfp}}_{h,g}(A) = \Big| \Pr[A^{h_{K_1}, g_{K_1}, \ldots, h_{K_m}, g_{K_m}} = 1 \mid K_1, \ldots, K_m \xleftarrow{\$} \mathcal{B}] - \Pr[A^{\rho_1, \rho'_1, \ldots, \rho_m, \rho'_m} = 1 \mid \rho_1, \rho'_1, \ldots, \rho_m, \rho'_m \xleftarrow{\$} \mathsf{Func}(\mathcal{B}, \mathcal{B})] \Big|$$

Lemma 7 Let $h_K(x) = E_K(x) \oplus x$ and $g_K(x) = L_K(x) \oplus x$. Let A be a prfp-adversary with $2m$ oracles. Suppose that A runs in time at most t, and makes at most q queries. Then, there exists a prfp-adversary B such that

$$\mathrm{Adv}^{m\text{-prfp}}_{h,g}(A) \leq m \cdot \mathrm{Adv}^{\text{prpp}}_{E,L}(B) + \frac{q(q-1)}{2^{n+1}} \ .$$

B makes at most q queries and runs in time at most $t + O(q(T_h + T_g))$, where T_h and T_g represent the time required to compute h and g, respectively.

Proof. For a permutation $\pi \in \mathsf{Perm}(\mathcal{B})$, let $\tilde{\pi}(x) = \pi(x) \oplus x$.

$$\begin{aligned}
\mathrm{Adv}^{m\text{-prfp}}_{h,g}(A) &= \Big|\Pr[A^{h_{K_1},g_{K_1},\ldots,h_{K_m},g_{K_m}} = 1 \mid K_1,\ldots,K_m \xleftarrow{\$} \mathcal{B}] \\
&\qquad - \Pr[A^{\rho_1,\rho'_1,\ldots,\rho_m,\rho'_m} = 1 \mid \rho_1,\rho'_1,\ldots,\rho_m,\rho'_m \xleftarrow{\$} \mathsf{Func}(\mathcal{B},\mathcal{B})]\Big| \\
&\le \Big|\Pr[A^{h_{K_1},g_{K_1},\ldots,h_{K_m},g_{K_m}} = 1 \mid K_1,\ldots,K_m \xleftarrow{\$} \mathcal{B}] \\
&\qquad - \Pr[A^{\tilde{\pi}_1,\tilde{\pi}'_1,\ldots,\tilde{\pi}_m,\tilde{\pi}'_m} = 1 \mid \pi_1,\pi'_1,\ldots,\pi_m,\pi'_m \xleftarrow{\$} \mathsf{Perm}(\mathcal{B})]\Big| + \\
&\quad \Big|\Pr[A^{\tilde{\pi}_1,\tilde{\pi}'_1,\ldots,\tilde{\pi}_m,\tilde{\pi}'_m} = 1 \mid \pi_1,\pi'_1,\ldots,\pi_m,\pi'_m \xleftarrow{\$} \mathsf{Perm}(\mathcal{B})] \\
&\qquad - \Pr[A^{\rho_1,\rho'_1,\ldots,\rho_m,\rho'_m} = 1 \mid \rho_1,\rho'_1,\ldots,\rho_m,\rho'_m \xleftarrow{\$} \mathsf{Func}(\mathcal{B},\mathcal{B})]\Big| \ .
\end{aligned}$$

For $0 \le i \le m$, let $\mathcal{O}_i$ be $2m$ oracles such that $h_{K_1}, g_{K_1}, \ldots, h_{K_i}, g_{K_i}, \tilde{\pi}_{i+1}, \tilde{\pi}'_{i+1}, \ldots, \tilde{\pi}_m, \tilde{\pi}'_m$, where $K_1, \ldots, K_i \xleftarrow{\$} \mathcal{B}$ and $\pi_{i+1}, \pi'_{i+1}, \ldots, \pi_m, \pi'_m \xleftarrow{\$} \mathsf{Perm}(\mathcal{B})$. A prpp-adversary B is constructed using A as a subroutine. The algorithm of B with oracle u, u' is as follows:

1. $i \xleftarrow{\$} \{1, 2, \ldots, m\}$.
2. runs A with oracles $h_{K_1}, g_{K_1}, \ldots, h_{K_{i-1}}, g_{K_{i-1}}, \tilde{u}, \tilde{u}', \tilde{\pi}_{i+1}, \tilde{\pi}'_{i+1}, \ldots, \tilde{\pi}_m, \tilde{\pi}'_m$, where $K_1, \ldots, K_{i-1} \xleftarrow{\$} \mathcal{B}$ and $\pi_{i+1}, \pi'_{i+1}, \ldots, \pi_m, \pi'_m \xleftarrow{\$} \mathsf{Perm}(\mathcal{B})$.
3. outputs A's output.

Then,

$$\Pr[B^{E_K,L_K} = 1 \mid K \xleftarrow{\$} \mathcal{B}] = \frac{1}{m}\sum_{i=1}^{m} \Pr[A^{\mathcal{O}_i} = 1]$$

and

$$\Pr[B^{\pi,\pi'} = 1 \mid \pi, \pi' \xleftarrow{\$} \mathsf{Perm}(\mathcal{B})] = \frac{1}{m}\sum_{i=0}^{m-1} \Pr[A^{\mathcal{O}_i} = 1] \ .$$

Thus,

$$\begin{aligned}
\mathrm{Adv}^{\mathrm{prpp}}_{E,L}(B) &= \Big|\Pr[B^{E_K,L_K} = 1 \mid K \xleftarrow{\$} \mathcal{B}] - \Pr[B^{\pi,\pi'} = 1 \mid \pi,\pi' \xleftarrow{\$} \mathsf{Perm}(\mathcal{B})]\Big| \\
&= \frac{1}{m}\Big|\Pr[A^{\mathcal{O}_m} = 1] - \Pr[A^{\mathcal{O}_0} = 1]\Big| \ .
\end{aligned}$$

B makes at most q queries and runs in time at most $t + O(q(T_h + T_g))$. There may exist an algorithm with the same resources and larger advantage. Let us also call it B. Then,

$$\Big|\Pr[A^{\mathcal{O}_m} = 1] - \Pr[A^{\mathcal{O}_0} = 1]\Big| \le m \cdot \mathrm{Adv}^{\mathrm{prpp}}_{E,L}(B) \ .$$

It is possible to distinguish $\tilde{\pi}_1, \tilde{\pi}'_1, \ldots, \tilde{\pi}_m, \tilde{\pi}'_m$ and $\rho_1, \rho'_1, \ldots, \rho_m, \rho'_m$ only by the fact that there may be a collision for $\rho(x) \oplus x$ for $\rho \in \mathsf{Func}(\mathcal{B}, \mathcal{B})$. Thus, since A makes at most q queries,

$$\begin{aligned} &\Big|\Pr[A^{\tilde{\pi}_1, \tilde{\pi}'_1, \ldots, \tilde{\pi}_m, \tilde{\pi}'_m} = 1 \mid \pi_1, \pi'_1, \ldots, \pi_m, \pi'_m \xleftarrow{\$} \mathsf{Perm}(\mathcal{B})] \\ &\qquad - \Pr[A^{\rho_1, \rho'_1, \ldots, \rho_m, \rho'_m} = 1 \mid \rho_1, \rho'_1, \ldots, \rho_m, \rho'_m \xleftarrow{\$} \mathsf{Func}(\mathcal{B}, \mathcal{B})]\Big| \\ &\leq \frac{q(q-1)}{2^{n+1}} \ . \end{aligned}$$

□

C.2 Security of Keyed-via-IV Mode

For the compression function h and the output function g, let $gh^* : \mathcal{B} \times \mathcal{B}^+ \to \mathcal{B}$ be a function family such that $gh^*(K, M)$ is defined for $K \in \mathcal{B}$ and $M \in \mathcal{B}^+$ as follows: Let $M = M^{(1)} \| \cdots \| M^{(N)}$ and $M^{(i)} \in \{0,1\}^n$ for $1 \leq i \leq N$. Then,

1. $a^{(0)} = K$,
2. If $N \geq 2$, then $a^{(i)} = h(a^{(i-1)}, M^{(i)})$ for $1 \leq i \leq N-1$,
3. $gh^*(K, M) = g(a^{(N-1)}, M^{(N)})$.

Keyed-Lesamnta is a function $gh^* : \mathcal{B} \times D \to \mathcal{B}$ such that $D = \{X \mid X = \mathsf{pad}(M) \text{ for some } M \in \{0,1\}^*\} \subset \mathcal{B}^+$, where pad is the padding function. Thus, in the following part, gh^* is analyzed instead of Keyed-Lesamnta. The analysis is similar to that of [15].

Lemma 8 Let A be a prf-adversary against gh^*. Suppose that A runs in time at most t, and makes at most q queries, and each query has at most ℓ blocks. Then, there exists a prfp-adversary B with access to $2q$ oracles such that

$$\mathrm{Adv}^{\mathrm{prf}}_{gh^*}(A) \leq \ell \cdot \mathrm{Adv}^{q\text{-prfp}}_{h,g}(B) \ .$$

B makes at most q queries and runs in time at most $t + O(q\,(\ell\, T_h + T_g))$.

Proof. Let $\mathcal{B}^{\leq i} = \bigcup_{d=0}^{i} \mathcal{B}^d$. For $i \in \{0, 1, \ldots, \ell\}$ and two functions $\alpha : \mathcal{B}^{\leq i} \to \mathcal{B}$ and $\beta : \mathcal{B}^i \to \mathcal{B}$, a function $I_i[\alpha, \beta] : \mathcal{B}^{\leq \ell} \to \mathcal{B}$ is defined as follows:

$$I_i[\alpha,\beta](M_1 M_2 \cdots M_k) = \begin{cases} \alpha(M_1 \cdots M_k) & \text{if } k \leq i, \\ gh^*(\beta(M_1 \cdots M_i), M_{i+1} \cdots M_k) & \text{if } k > i. \end{cases}$$

Notice that α and β are just random elements from $\mathcal{B}$ if $i = 0$. Let

$$P_i = \Pr[A^{I_i[\alpha,\beta]}) = 1 \mid \alpha \xleftarrow{\$} \mathsf{Func}(\mathcal{B}^{\leq i}, \mathcal{B}) \wedge \beta \xleftarrow{\$} \mathsf{Func}(\mathcal{B}^i, \mathcal{B})] \ .$$

Note that

$$\mathrm{Adv}^{\mathrm{prf}}_{gh^*}(A) = \left|P_0 - P_\ell\right| \ .$$

A q-prfp-adversary B with $2q$ oracles is constructed using A as a subroutine. For $i \in \{1, \ldots, \ell\}$, a q-prfp-adversary $B_i^{u_1,u'_1,\ldots,u_q,u'_q}$ is first defined.

B_i first picks $\gamma \xleftarrow{\$} \mathsf{Func}(\mathcal{B}^{\le i-1}, \mathcal{B})$. Actually, B_i implements γ via lazy sampling. Then, B_i runs A. B_i has to answer q queries of A appropriately. In order to do that, B_i maintains a counter idx, which is initially set to 0. When B_i receives the j-th query $M_j = M_j^{(1)} M_j^{(2)} \cdots M_j^{(k)}$ of A, B_i returns

$$\begin{cases} \gamma(M_j^{(1)} \cdots M_j^{(k)}) & \text{if } k < i, \\ u'_{\mathsf{idx}(M_j^{(1)} \cdots M_j^{(i-1)})}(M_j^{(i)}) & \text{if } k = i, \\ gh^*(u_{\mathsf{idx}(M_j^{(1)} \cdots M_j^{(i-1)})}(M_j^{(i)}), M_j^{(i+1)} \cdots M_j^{(k)}) & \text{if } k > i. \end{cases}$$

In the above, $\mathsf{idx}(M_j^{(1)} \cdots M_j^{(i-1)})$ is a unique integer in $\{1, \ldots, q\}$ which depends on the query $M_j^{(1)} \cdots M_j^{(i-1)}$. It can be defined using the counter idx, which is initially 0. If there is a previous query M_p ($p < j$) such that $M_p^{(1)} \cdots M_p^{(i-1)} = M_j^{(1)} \cdots M_j^{(i-1)}$, then define $\mathsf{idx}(M_j^{(1)} \cdots M_j^{(i-1)}) = \mathsf{idx}(M_p^{(1)} \cdots M_p^{(i-1)})$, and otherwise increase idx by 1 and define $\mathsf{idx}(M_j^{(1)} \cdots M_j^{(i-1)}) = idx$.

Now, suppose that B_i is given oracles u_l, u'_l such that $u_l = h_{K_l}$ and $u'_l = g_{K_l}$ with $K_l \xleftarrow{\$} \mathcal{B}$ for $1 \le l \le q$. Then, when A makes the j-th query $M_j = M_j^{(1)} M_j^{(2)} \cdots M_j^{(k)}$, B_i returns

$$\begin{cases} \gamma(M_j^{(1)} \cdots M_j^{(k)}) & \text{if } k < i, \\ g_{K_{\mathsf{idx}(M_j^{(1)} \cdots M_j^{(i-1)})}}(M_j^{(i)}) = g(K_{\mathsf{idx}(M_j^{(1)} \cdots M_j^{(i-1)})}, M_j^{(i)}) & \text{if } k = i, \\ gh^*(h_{K_{\mathsf{idx}(M_j^{(1)} \cdots M_j^{(i-1)})}}(M_j^{(i)}), M_j^{(i+1)} \cdots M_j^{(k)}) = gh^*(K_{\mathsf{idx}(M_j^{(1)} \cdots M_j^{(i-1)})}, M_j^{(i)} M_j^{(i+1)} \cdots M_j^{(k)}) & \text{if } k > i. \end{cases}$$

Since $K_{\mathsf{idx}(M_1^j \cdots M_{i-1}^j)}$ is a random function of $M_1^j \cdots M_{i-1}^j$, we can say that A hash oracle access to $I_{i-1}[\alpha, \beta]$ with $\alpha \xleftarrow{\$} \mathsf{Func}(\mathcal{B}^{\le i-1}, \mathcal{B})$ and $\beta \xleftarrow{\$} \mathsf{Func}(\mathcal{B}^{i-1}, \mathcal{B})$. Therefore,

$$\Pr[B_i^{h_{K_1}, g_{K_1}, \ldots, h_{K_q}, g_{K_q}} = 1 \mid K_1, \ldots, K_q \xleftarrow{\$} \mathcal{B}] = P_{i-1} \ .$$

Next, suppose that B_i is given $2q$ independent random oracles $\rho_1, \rho'_1, \ldots, \rho_q, \rho'_q \xleftarrow{\$} \mathsf{Func}(\mathcal{B}, \mathcal{B})$. Then, B_i returns

$$\begin{cases} \gamma(M_j^{(1)} \cdots M_j^{(k)}) & \text{if } k < i, \\ \rho'_{\mathsf{idx}(M_j^{(1)} \cdots M_j^{(i-1)})}(M_j^{(i)}) & \text{if } k = i, \\ gh^*(\rho_{\mathsf{idx}(M_j^{(1)} \cdots M_j^{(i-1)})}(M_j^{(i)}), M_j^{(i+1)} \cdots M_j^{(k)}) & \text{if } k > i. \end{cases}$$

Since $\rho_{\mathsf{idx}(M_j^{(1)} \cdots M_j^{(i-1)})}(M_j^{(i)})$ and $\rho'_{\mathsf{idx}(M_j^{(1)} \cdots M_j^{(i-1)})}(M_j^{(i)})$ are independent random functions of $M_j^{(1)} \cdots M_j^{(i-1)} M_j^{(i)}$, we can say that A has oracle access to $I_i[\alpha, \beta]$ with $\alpha \xleftarrow{\$} \mathsf{Func}(\mathcal{B}^{\le i}, \mathcal{B})$ and $\beta \xleftarrow{\$} \mathsf{Func}(\mathcal{B}^{i}, \mathcal{B})$. Therefore,

$$\Pr[B_i^{\rho_1, \rho'_1, \ldots, \rho_q, \rho'_q} = 1 \mid \rho_1, \rho'_1, \ldots, \rho_q, \rho'_q \xleftarrow{\$} \mathsf{Func}(\mathcal{B}, \mathcal{B})] = P_i \ .$$

Finally, B is defined as follows: It first chooses $i \xleftarrow{\$} \{1, \ldots, \ell\}$, then behaves identically to B_i. Then,

$$\begin{aligned}\mathrm{Adv}_{h,g}^{q\text{-prfp}}(B) &= \Big|\Pr[B^{h_{K_1},g_{K_1},\ldots,h_{K_q},g_{K_q}} = 1 \mid K_1, \ldots, K_q \xleftarrow{\$} \mathcal{B}] \\ &\qquad - \Pr[B^{\rho_1,\rho_1',\ldots,\rho_q,\rho_q'} = 1 \mid \rho_1, \rho_1', \ldots, \rho_q, \rho_q' \xleftarrow{\$} \mathsf{Func}(\mathcal{B}, \mathcal{B})]\Big| \\ &= \frac{1}{\ell}\Big|P_0 - P_\ell\Big| = \frac{1}{\ell}\mathrm{Adv}_{gh^*}^{\mathrm{prf}}(A) \ .\end{aligned}$$

B makes at most q queries and runs in time at most $t+O(q\,(\ell\, T_h + T_g))$. There may exist an algorithm with the same resources and larger advantage. Let us also call it B. Then,

$$\mathrm{Adv}_{gh^*}^{\mathrm{prf}}(A) \leq \ell \cdot \mathrm{Adv}_{h,g}^{q\text{-prfp}}(B) \ .$$

□

The following theorem directly follows from Lemmas 7 and 8.

Theorem 3 Let A be a prf-adversary against gh^*. Suppose that A runs in time at most t, and makes at most q queries, and each query has at most ℓ blocks. Then, there exists a prpp-adversary B such that

$$\mathrm{Adv}_{gh^*}^{\mathrm{prf}}(A) \leq \ell q \cdot \mathrm{Adv}_{E,L}^{\mathrm{prpp}}(B) + \frac{\ell q(q-1)}{2^{n+1}} \ .$$

B makes at most q queries and runs in time at most $t + O(q\,(\ell\, T_h + T_g))$.

The following corollary is immediate from Theorem 3. It is on the pseudorandomness of Keyed-Lesamnta.

Corollary 2 Let A be a prf-adversary against Keyed Lesamnta. Suppose that A runs in time at most t, and makes at most q queries, and each query has at most ℓ blocks. Then, there exists a prpp-adversary B such that

$$\mathrm{Adv}_{\mathrm{Keyed}}^{\mathrm{prf}}(A) \leq \ell q \cdot \mathrm{Adv}_{E,L}^{\mathrm{prpp}}(B) + \frac{\ell q(q-1)}{2^{n+1}} \ .$$

B makes at most q queries and runs in time at most $t + O(q\,(\ell\, T_E + T_L))$, where T_E and T_L represent the time required to compute E and L, respectively.

C.3 Security of Key-Prefix Mode

Let $\nu_E : \mathcal{B} \to \mathcal{B}$ be a function such that $\nu_E(K) = E_{IV}(K) \oplus K$. Key-Prefix-Lesamnta with a key $K \in \mathcal{B}$ and a message input $M \in \{0,1\}^*$ is $gh^*(\nu_E(K), M')$, where $M' \in \mathcal{B}^+$ and $\mathsf{pad}(K\|M) = K\|M'$.

The following lemma says that Key-Prefix-Lesamnta is a PRF if gh^* is a PRF and ν_E is a PRBG.

Lemma 9 Let A be a prf-adversary against Key-Prefix-Lesamnta. Suppose that A runs in time at most t and makes at most q queries, and each query has at most ℓ blocks. Then, there exist a prf-adversary B against gh^* and a prbg-adversary B' against ν_E such that

$$\mathrm{Adv}^{\mathrm{prf}}_{\text{Key-prefix}}(A) \leq \mathrm{Adv}^{\mathrm{prf}}_{gh^*}(B) + \mathrm{Adv}^{\mathrm{prbg}}_{\nu_E}(B') \ .$$

B runs in time at most $t + O(\ell nq)$, makes at most q queries, and each query has at most ℓ blocks. B' runs in time at most $t + O(q\,(\ell\, T_h + T_g))$.

Now, the security of Key-Prefix-Lesamnta as a PRF is reduced to the security of E and L as a PRP pair and that of ν_E as a PRBG.

Theorem 4 Let A be a prf-adversary against Key-Prefix-Lesamnta. Suppose that A runs in time at most t, and makes at most q queries, and each query has at most ℓ blocks. Then, there exist a prpp-adversary B against E and L, and a prbg-adversary B' against ν_E such that

$$\mathrm{Adv}^{\mathrm{prf}}_{\text{Key-prefix}}(A) \leq \ell q \cdot \mathrm{Adv}^{\mathrm{prpp}}_{E,L}(B) + \mathrm{Adv}^{\mathrm{prbg}}_{\nu_E}(B') + \frac{\ell q(q-1)}{2^{n+1}} \ .$$

B makes at most q queries and runs in time at most $t + O(q\,(\ell\, T_E + T_L))$. B' runs in time at most $t + O(q\,(\ell\, T_E + T_L))$.