Specification of *Camellia* — a 128-bit Block Cipher

Kazumaro AOKI †, Tetsuya ICHIKAWA ‡, Masayuki KANDA †,
Mitsuru MATSUI ‡, Shiho MORIAI †, Junko NAKAJIMA ‡, Toshio TOKITA ‡

†Nippon Telegraph and Telephone Corporation, ‡Mitsubishi Electric Corporation

Version 1.0: July 12, 2000
Version 2.0: September 26, 2001

Contents

1 Introduction

This document shows a complete description of the encryption algorithm *Camellia*, which is a secret key cipher with 128-bit data block and 128/192/256-bit secret key.

2 Notations and Conventions

2.1 Radix

We use the prefix **0x** to indicate **hexadecimal** numbers.

2.2 Notations

Throughout this document, the following notations are used.

- $\mathbf{B}$ denotes a vector space of 8-bit (byte) elements; that is, $\mathbf{B} := \mathrm{GF}(2)^8$.
- $\mathbf{W}$ denotes a vector space of 32-bit (word) elements; that is, $\mathbf{W} := \mathbf{B}^4$.
- $\mathbf{L}$ denotes a vector space of 64-bit (double word) elements; that is, $\mathbf{L} := \mathbf{B}^8$.
- $\mathbf{Q}$ denotes a vector space of 128-bit (quad word) elements; that is, $\mathbf{Q} := \mathbf{B}^{16}$.
- An element with the suffix $_{(n)}$ (e.g. $x_{(n)}$) shows that the element is n-bit long.
- An element with the suffix $_L$ (e.g. x_L) denotes left-half part of x.
- An element with the suffix $_R$ (e.g. x_R) denotes right-half part of x.

The suffix $_{(n)}$ will be omitted if no ambiguity is expected. See section 2.4 for numerical examples of "left" and "right".

2.3 List of Symbols

$\oplus$	The bitwise exclusive-OR operation.
$\|\|$	The concatenation of the two operands.
$\lll_n$	The left circular rotation of the operand by n bits.
$\cap$	The bitwise AND operation.
$\cup$	The bitwise OR operation.
$\overline{x}$	The bitwise complement of x.

2.4 Bit/Byte Ordering

We adopt big endian ordering. The following example shows how to compose a 128-bit value $Q_{(128)}$ of two 64-bit values $L_{i(64)}$ $(i = 1, 2)$, four 32-bit values $W_{i(32)}$ $(i = 1, 2, 3, 4)$, sixteen 8-bit values $B_{i(8)}$ $(i = 1, 2, \ldots, 16)$, or 128 1-bit values $E_{i(1)}$ $(i = 1, 2, \ldots, 128)$, respectively.

$$
\begin{aligned}
Q_{(128)} &= L_{1(64)}||L_{2(64)}\\
&= W_{1(32)}||W_{2(32)}||W_{3(32)}||W_{4(32)}\\
&= B_{1(8)}||B_{2(8)}||B_{3(8)}||B_{4(8)}||\ldots\ldots||B_{13(8)}||B_{14(8)}||B_{15(8)}||B_{16(8)}\\
&= E_{1(1)}||E_{2(1)}||E_{3(1)}||E_{4(1)}||\ldots\ldots\ldots\ldots\ldots\ldots||E_{125(1)}||E_{126(1)}||E_{127(1)}||E_{128(1)}
\end{aligned}
$$

Numerical examples:

$$
\begin{array}{llllllllllll}
Q_{(128)} & = & \multicolumn{10}{l}{\texttt{0x0123456789ABCDEF0011223344556677}_{(128)}}\\
L_{1(64)} & = & Q_{L(64)} & = & \multicolumn{8}{l}{\texttt{0x0123456789ABCDEF}_{(64)}}\\
L_{2(64)} & = & Q_{R(64)} & = & \multicolumn{8}{l}{\texttt{0x0011223344556677}_{(64)}}\\
W_{1(32)} & = & L_{1L(32)} & = & \multicolumn{8}{l}{\texttt{0x01234567}_{(32)}}\\
W_{2(32)} & = & L_{1R(32)} & = & \multicolumn{8}{l}{\texttt{0x89ABCDEF}_{(32)}}\\
W_{3(32)} & = & L_{2L(32)} & = & \multicolumn{8}{l}{\texttt{0x00112233}_{(32)}}\\
W_{4(32)} & = & L_{2R(32)} & = & \multicolumn{8}{l}{\texttt{0x44556677}_{(32)}}\\
B_{1(8)} & = & \texttt{0x01}_{(8)}, & B_{2(8)} & = & \texttt{0x23}_{(8)}, & B_{3(8)} & = & \texttt{0x45}_{(8)}, & B_{4(8)} & = & \texttt{0x67}_{(8)},\\
B_{5(8)} & = & \texttt{0x89}_{(8)}, & B_{6(8)} & = & \texttt{0xAB}_{(8)}, & B_{7(8)} & = & \texttt{0xCD}_{(8)}, & B_{8(8)} & = & \texttt{0xEF}_{(8)},\\
B_{9(8)} & = & \texttt{0x00}_{(8)}, & B_{10(8)} & = & \texttt{0x11}_{(8)}, & B_{11(8)} & = & \texttt{0x22}_{(8)}, & B_{12(8)} & = & \texttt{0x33}_{(8)},\\
B_{13(8)} & = & \texttt{0x44}_{(8)}, & B_{14(8)} & = & \texttt{0x55}_{(8)}, & B_{15(8)} & = & \texttt{0x66}_{(8)}, & B_{16(8)} & = & \texttt{0x77}_{(8)},\\
E_{1(1)} & = & \texttt{0}_{(1)}, & E_{2(1)} & = & \texttt{0}_{(1)}, & E_{3(1)} & = & \texttt{0}_{(1)}, & E_{4(1)} & = & \texttt{0}_{(1)},\\
E_{5(1)} & = & \texttt{0}_{(1)}, & E_{6(1)} & = & \texttt{0}_{(1)}, & E_{7(1)} & = & \texttt{0}_{(1)}, & E_{8(1)} & = & \texttt{1}_{(1)},\\
 & \vdots & & & & & & & & & & \\
E_{121(1)} & = & \texttt{0}_{(1)}, & E_{122(1)} & = & \texttt{1}_{(1)}, & E_{123(1)} & = & \texttt{1}_{(1)}, & E_{124(1)} & = & \texttt{1}_{(1)},\\
E_{125(1)} & = & \texttt{0}_{(1)}, & E_{126(1)} & = & \texttt{1}_{(1)}, & E_{127(1)} & = & \texttt{1}_{(1)}, & E_{128(1)} & = & \texttt{1}_{(1)}.
\end{array}
$$

$$
\begin{aligned}
Q_{(128)} \lll_1 &= E_{2(1)}||E_{3(1)}||E_{4(1)}||E_{5(1)}||\ldots\ldots\ldots\ldots\ldots\ldots||E_{125(1)}||E_{126(1)}||E_{127(1)}||E_{128(1)}||E_{1(1)}\\
&= \texttt{0x02468ACF13579BDE0022446688AACCEE}_{(128)}
\end{aligned}
$$

3 Structure of *Camellia*

3.1 List of Functions and Variables

$M_{(128)}$	The plaintext block.
$C_{(128)}$	The ciphertext block.
K	The secret key, whose length is 128, 192, or 256 bits.
$kw_{t(64)}, k_{u(64)}, kl_{v(64)}$	The subkeys. $(t = 1,2,3,4)$ $(u = 1,2,\ldots,18)$ $(v = 1,2,3,4)$ for 128-bit secret key. $(t = 1,2,3,4)$ $(u = 1,2,\ldots,24)$ $(v = 1,2,\ldots,6)$ for 192-bit and 256-bit secret key.
$Y_{(64)} = F(X_{(64)}, k_{(64)})$	The F-function that transforms a 64-bit input $X_{(64)}$ to a 64-bit output $Y_{(64)}$ using a 64-bit subkey $k_{(64)}$.
$Y_{(64)} = FL(X_{(64)}, k_{(64)})$	The FL-function that transforms a 64-bit input $X_{(64)}$ to a 64-bit output $Y_{(64)}$ using a 64-bit subkey $k_{(64)}$.
$Y_{(64)} = FL^{-1}(X_{(64)}, k_{(64)})$	The FL^{-1}-function that transforms a 64-bit input $X_{(64)}$ to a 64-bit output $Y_{(64)}$ using a 64-bit subkey $k_{(64)}$.
$Y_{(64)} = S(X_{(64)})$	The S-function that transforms a 64-bit input $X_{(64)}$ to a 64-bit output $Y_{(64)}$.
$Y_{(64)} = P(X_{(64)})$	The P-function that transforms a 64-bit input $X_{(64)}$ to a 64-bit output $Y_{(64)}$.
$y_{(8)} = s_i(x_{(8)})$	The s-boxes that transform an 8-bit input to an 8-bit output $(i = 1,2,3,4)$.

3.2 Encryption Procedure

3.2.1 128-bit key

Figure 1 shows the encryption procedure for a 128-bit key. The data randomizing part has an 18-round Feistel structure with two FL/FL^{-1}-function layers after the 6-th and 12-th rounds, and 128-bit XOR operations before the first round and after the last round. The key schedule part generates subkeys $kw_{t(64)}$ $(t = 1,2,3,4)$, $k_{u(64)}$ $(u = 1,2,\ldots,18)$ and $kl_{v(64)}$ $(v = 1,2,3,4)$ from the secret key K; see section 3.4 for details of the key schedule part.

In the data randomizing part, first the plaintext $M_{(128)}$ is XORed with $kw_{1(64)}||kw_{2(64)}$ and separated into $L_{0(64)}$ and $R_{0(64)}$ of equal length, i.e., $M_{(128)} \oplus (kw_{1(64)}||kw_{2(64)}) = L_{0(64)}||R_{0(64)}$. Then, the following operations are perfomed from $r = 1$ to 18, except for $r = 6$ and 12;

$$\begin{aligned} L_r &= R_{r-1} \oplus F(L_{r-1}, k_r), \\ R_r &= L_{r-1}. \end{aligned}$$

For $r = 6$ and 12, the following is carried out;

$$\begin{aligned} L'_r &= R_{r-1} \oplus F(L_{r-1}, k_r), \\ R'_r &= L_{r-1}, \\ L_r &= FL(L'_r, kl_{2r/6-1}), \\ R_r &= FL^{-1}(R'_r, kl_{2r/6}). \end{aligned}$$

Lastly, $R_{18(64)}$ and $L_{18(64)}$ are concatenated and XORed with $kw_{3(64)}||kw_{4(64)}$. The resultant value is the ciphertext, i.e., $C_{(128)} = (R_{18(64)}||L_{18(64)}) \oplus (kw_{3(64)}||kw_{4(64)})$.

3.2.2 192-bit and 256-bit key

Figure 2 shows the encryption procedure for a 192-bit or 256-bit key. The data randomizing part has a 24-round Feistel structure with three FL/FL^{-1}-function layers after the 6-th, 12-th, and 18-th rounds, and 128-bit XOR operations before the first round and after the last round. The key schedule part generates subkeys $kw_{t(64)}$ ($t = 1, 2, 3, 4$), $k_{u(64)}$ ($u = 1, 2, \ldots, 24$), and $kl_{v(64)}$ ($v = 1, 2, \ldots, 6$) from the secret key K.

In the data randomizing part, first the plaintext $M_{(128)}$ is XORed with $kw_{1(64)}||kw_{2(64)}$ and separated into $L_{0(64)}$ and $R_{0(64)}$ of equal length, i.e., $M_{(128)} \oplus (kw_{1(64)}||kw_{2(64)}) = L_{0(64)}||R_{0(64)}$. Then, perform the following operations from $r = 1$ to 24, except for $r = 6$, 12, and 18;

$$\begin{aligned} L_r &= R_{r-1} \oplus F(L_{r-1}, k_r), \\ R_r &= L_{r-1}. \end{aligned}$$

For $r = 6$, 12, and 18, perform the following;

$$\begin{aligned} L'_r &= R_{r-1} \oplus F(L_{r-1}, k_r), \\ R'_r &= L_{r-1}, \\ L_r &= FL(L'_r, kl_{2r/6-1}), \\ R_r &= FL^{-1}(R'_r, kl_{2r/6}). \end{aligned}$$

Lastly, $R_{24(64)}$ and $L_{24(64)}$ are concatenated and XORed with $kw_{3(64)}||kw_{4(64)}$. The resultant value is the ciphertext, i.e., $C_{(128)} = (R_{24(64)}||L_{24(64)}) \oplus (kw_{3(64)}||kw_{4(64)})$.

See section 4 for details of the F-function and FL/FL^{-1}-functions.

3.3 Decryption Procedure

3.3.1 128-bit key

The decryption procedure of *Camellia* can be done in the same way as the encryption procedure by reversing the order of the subkeys.

Figure 3 shows the decryption procedure for a 128-bit key. The data randomizing part has an 18-round Feistel structure with two FL/FL^{-1}-function layers after the 6-th and 12-th rounds, and 128-bit XOR operations before the first round and after the last round. The key schedule part generates subkeys $kw_{t(64)}$ $(t = 1, 2, 3, 4)$, $k_{u(64)}$ $(u = 1, 2, \dots, 18)$, and $kl_{v(64)}$ $(v = 1, 2, 3, 4)$ from the secret key K; see section 3.4 for details of the key schedule part.

In the data randomizing part, first the ciphertext $C_{(128)}$ is XORed with $kw_{3(64)}||kw_{4(64)}$ and separated into $R_{18(64)}$ and $L_{18(64)}$ of equal length, i.e., $C_{(128)} \oplus (kw_{3(64)}||kw_{4(64)}) = R_{18(64)}||L_{18(64)}$. Then, the following operations are perfomed from $r = 18$ down to 1, except for $r = 13$ and 7;

$$\begin{aligned} R_{r-1} &= L_r \oplus F(R_r, k_r), \\ L_{r-1} &= R_r. \end{aligned}$$

For $r = 13$ and 7, the following is carried out;

$$\begin{aligned} R'_{r-1} &= L_r \oplus F(R_r, k_r), \\ L'_{r-1} &= R_r. \\ R_{r-1} &= FL(R'_{r-1}, kl_{2(r-1)/6}), \\ L_{r-1} &= FL^{-1}(L'_{r-1}, kl_{2(r-1)/6-1}). \end{aligned}$$

Lastly, $L_{0(64)}$ and $R_{0(64)}$ are concatenated and XORed with $kw_{1(64)}||kw_{2(64)}$. The resultant value is the plaintext, i.e., $M_{(128)} = (L_{0(64)}||R_{0(64)}) \oplus (kw_{1(64)}||kw_{2(64)})$.

3.3.2 192-bit and 256-bit key

Figure 4 shows the decryption procedure for a 192-bit or 256-bit key. The data randomizing part has a 24-round Feistel structure with three FL/FL^{-1}-function layers after the 6-th, 12-th, and 18-th rounds, and 128-bit XOR operations before the first round and after the last round. The key schedule part generates subkeys $kw_{t(64)}$ $(t = 1, 2, 3, 4)$, $k_{u(64)}$ $(u = 1, 2, \dots, 24)$, and $kl_{v(64)}$ $(v = 1, 2, \dots, 6)$ from the secret key K.

In the data randomizing part, first the ciphertext $C_{(128)}$ is XORed with $kw_{3(64)}||kw_{4(64)}$ and separated into $R_{24(64)}$ and $L_{24(64)}$ of equal length, i.e., $C_{(128)} \oplus (kw_{3(64)}||kw_{4(64)}) = R_{24(64)}||L_{24(64)}$. Then, perform the following operations from $r = 24$ down to 1, except for $r = 19$, 13, and 7;

$$\begin{aligned} R_{r-1} &= L_r \oplus F(R_r, k_r), \\ L_{r-1} &= R_r. \end{aligned}$$

For $r = 19$, 13, and 7, perform the following.

$$\begin{aligned} R'_{r-1} &= L_r \oplus F(R_r, k_r), \\ L'_{r-1} &= R_r. \\ R_{r-1} &= FL(R'_{r-1}, kl_{2(r-1)/6}), \\ L_{r-1} &= FL^{-1}(L'_{r-1}, kl_{2(r-1)/6-1}). \end{aligned}$$

Lastly, $L_{0(64)}$ and $R_{0(64)}$ are concatenated and XORed with $kw_{1(64)}||kw_{2(64)}$. The resultant value is the plaintext, i.e., $M_{(128)} = (L_{0(64)}||R_{0(64)}) \oplus (kw_{1(64)}||kw_{2(64)})$.

3.4 Key Schedule

In the key schedule part of *Camellia*, we introduce two 128-bit variables $K_{L(128)}$, $K_{R(128)}$ and four 64-bit variables $K_{LL(64)}$, $K_{LR(64)}$, $K_{RL(64)}$ and $K_{RR(64)}$, which are defined so that the following relations are satisfied:

$$
\begin{array}{lllllll}
K_{(128)} & = & K_{L(128)}, & K_{R(128)} & = & 0; & \text{for 128-bit key,} \\
K_{(192)} & = & K_{L(128)}||K_{RL(64)}, & K_{RR(64)} & = & \overline{K_{RL(64)}}; & \text{for 192-bit key,} \\
K_{(256)} & = & K_{L(128)}||K_{R(128)}; & & & & \text{for 256-bit key.}
\end{array}
$$

$$
\left.\begin{array}{lll}
K_{L(128)} & = & K_{LL(64)}||K_{LR(64)}, \\
K_{R(128)} & = & K_{RL(64)}||K_{RR(64)};
\end{array}\right. \qquad \text{for any size of key.}
$$

Using these variables, we generate two 128-bit variables $K_{A(128)}$ and $K_{B(128)}$, as shown in figure 8, where $K_{B(128)}$ is used only if the length of the secret key is 192 or 256 bits. First $K = K_{L(128)}$ is XORed with $K_{R(128)}$ and "encrypted" by two rounds using the constant values $\Sigma_{1(64)}$ and $\Sigma_{2(64)}$ as "keys". The result is XORed with $K_{L(128)}$ and again encrypted by two rounds using the constant values $\Sigma_{3(64)}$ and $\Sigma_{4(64)}$; the resultant value is $K_{A(128)}$. Lastly $K_{A(128)}$ is XORed with $K_{R(128)}$ and encrypted by two rounds using the constant values $\Sigma_{5(64)}$ and $\Sigma_{6(64)}$; the resultant value is $K_{B(128)}$. Σ_i is defined as the continuous values from the second hexadecimal place to the seventeenth hexadecimal place of the hexadecimal representation of the square root of the i-th prime. These constant values are listed in table 1.

The subkeys $kw_{t(64)}, k_{u(64)}$, and $kl_{v(64)}$ are generated from (left-half or right-half part of) rotate shifted values of $K_{L(128)}$, $K_{R(128)}$, $K_{A(128)}$, and $K_{B(128)}$. The exact details are shown in table 2 and table 3, respectively.

Therefore by setting $K_{RR(64)} = \overline{K_{RL(64)}}$, the 256-bit version is compatible with the 192-bit version.

Table 1: The key schedule constants

$\Sigma_{1(64)}$	`0xA09E667F3BCC908B`
$\Sigma_{2(64)}$	`0xB67AE8584CAA73B2`
$\Sigma_{3(64)}$	`0xC6EF372FE94F82BE`
$\Sigma_{4(64)}$	`0x54FF53A5F1D36F1C`
$\Sigma_{5(64)}$	`0x10E527FADE682D1D`
$\Sigma_{6(64)}$	`0xB05688C2B3E6C1FD`

Table 2: Subkeys for 128-bit secret key

	subkey	value
Prewhitening	$kw_{1(64)}$	$(K_L \lll_0)_{L(64)}$
	$kw_{2(64)}$	$(K_L \lll_0)_{R(64)}$
F (Round1)	$k_{1(64)}$	$(K_A \lll_0)_{L(64)}$
F (Round2)	$k_{2(64)}$	$(K_A \lll_0)_{R(64)}$
F (Round3)	$k_{3(64)}$	$(K_L \lll_{15})_{L(64)}$
F (Round4)	$k_{4(64)}$	$(K_L \lll_{15})_{R(64)}$
F (Round5)	$k_{5(64)}$	$(K_A \lll_{15})_{L(64)}$
F (Round6)	$k_{6(64)}$	$(K_A \lll_{15})_{R(64)}$
FL	$kl_{1(64)}$	$(K_A \lll_{30})_{L(64)}$
FL^{-1}	$kl_{2(64)}$	$(K_A \lll_{30})_{R(64)}$
F (Round7)	$k_{7(64)}$	$(K_L \lll_{45})_{L(64)}$
F (Round8)	$k_{8(64)}$	$(K_L \lll_{45})_{R(64)}$
F (Round9)	$k_{9(64)}$	$(K_A \lll_{45})_{L(64)}$
F (Round10)	$k_{10(64)}$	$(K_L \lll_{60})_{R(64)}$
F (Round11)	$k_{11(64)}$	$(K_A \lll_{60})_{L(64)}$
F (Round12)	$k_{12(64)}$	$(K_A \lll_{60})_{R(64)}$
FL	$kl_{3(64)}$	$(K_L \lll_{77})_{L(64)}$
FL^{-1}	$kl_{4(64)}$	$(K_L \lll_{77})_{R(64)}$
F (Round13)	$k_{13(64)}$	$(K_L \lll_{94})_{L(64)}$
F (Round14)	$k_{14(64)}$	$(K_L \lll_{94})_{R(64)}$
F (Round15)	$k_{15(64)}$	$(K_A \lll_{94})_{L(64)}$
F (Round16)	$k_{16(64)}$	$(K_A \lll_{94})_{R(64)}$
F (Round17)	$k_{17(64)}$	$(K_L \lll_{111})_{L(64)}$
F (Round18)	$k_{18(64)}$	$(K_L \lll_{111})_{R(64)}$
Postwhitening	$kw_{3(64)}$	$(K_A \lll_{111})_{L(64)}$
	$kw_{4(64)}$	$(K_A \lll_{111})_{R(64)}$

Table 3: Subkeys for 192/256-bit secret key

	subkey	value
Prewhitening	$kw_{1(64)}$	$(K_L \lll_0)_{L(64)}$
	$kw_{2(64)}$	$(K_L \lll_0)_{R(64)}$
F (Round1)	$k_{1(64)}$	$(K_B \lll_0)_{L(64)}$
F (Round2)	$k_{2(64)}$	$(K_B \lll_0)_{R(64)}$
F (Round3)	$k_{3(64)}$	$(K_R \lll_{15})_{L(64)}$
F (Round4)	$k_{4(64)}$	$(K_R \lll_{15})_{R(64)}$
F (Round5)	$k_{5(64)}$	$(K_A \lll_{15})_{L(64)}$
F (Round6)	$k_{6(64)}$	$(K_A \lll_{15})_{R(64)}$
FL	$kl_{1(64)}$	$(K_R \lll_{30})_{L(64)}$
FL^{-1}	$kl_{2(64)}$	$(K_R \lll_{30})_{R(64)}$
F (Round7)	$k_{7(64)}$	$(K_B \lll_{30})_{L(64)}$
F (Round8)	$k_{8(64)}$	$(K_B \lll_{30})_{R(64)}$
F (Round9)	$k_{9(64)}$	$(K_L \lll_{45})_{L(64)}$
F (Round10)	$k_{10(64)}$	$(K_L \lll_{45})_{R(64)}$
F (Round11)	$k_{11(64)}$	$(K_A \lll_{45})_{L(64)}$
F (Round12)	$k_{12(64)}$	$(K_A \lll_{45})_{R(64)}$
FL	$kl_{3(64)}$	$(K_L \lll_{60})_{L(64)}$
FL^{-1}	$kl_{4(64)}$	$(K_L \lll_{60})_{R(64)}$
F (Round13)	$k_{13(64)}$	$(K_R \lll_{60})_{L(64)}$
F (Round14)	$k_{14(64)}$	$(K_R \lll_{60})_{R(64)}$
F (Round15)	$k_{15(64)}$	$(K_B \lll_{60})_{L(64)}$
F (Round16)	$k_{16(64)}$	$(K_B \lll_{60})_{R(64)}$
F (Round17)	$k_{17(64)}$	$(K_L \lll_{77})_{L(64)}$
F (Round18)	$k_{18(64)}$	$(K_L \lll_{77})_{R(64)}$
FL	$kl_{5(64)}$	$(K_A \lll_{77})_{L(64)}$
FL^{-1}	$kl_{6(64)}$	$(K_A \lll_{77})_{R(64)}$
F (Round19)	$k_{19(64)}$	$(K_R \lll_{94})_{L(64)}$
F (Round20)	$k_{20(64)}$	$(K_R \lll_{94})_{R(64)}$
F (Round21)	$k_{21(64)}$	$(K_A \lll_{94})_{L(64)}$
F (Round22)	$k_{22(64)}$	$(K_A \lll_{94})_{R(64)}$
F (Round23)	$k_{23(64)}$	$(K_L \lll_{111})_{L(64)}$
F (Round24)	$k_{24(64)}$	$(K_L \lll_{111})_{R(64)}$
Postwhitening	$kw_{3(64)}$	$(K_B \lll_{111})_{L(64)}$
	$kw_{4(64)}$	$(K_B \lll_{111})_{R(64)}$

4 Components of *Camellia*

4.1 *F*-function

The F-function is shown in figure 5, which is defined as follows:

$$\begin{aligned} F : \mathbf{L} \times \mathbf{L} &\longrightarrow \mathbf{L} \\ (X_{(64)}, k_{(64)}) &\longmapsto Y_{(64)} = P(S(X_{(64)} \oplus k_{(64)})). \end{aligned}$$

See sections 4.4 and 4.6 for the S-function and the P-function, respectively.

4.2 *FL*-function

The FL-function is shown in figure 6, which is defined as follows:

$$\begin{aligned} FL : \mathbf{L} \times \mathbf{L} &\longrightarrow \mathbf{L} \\ (X_{L(32)} || X_{R(32)}, kl_{L(32)} || kl_{R(32)}) &\longmapsto Y_{L(32)} || Y_{R(32)}, \end{aligned}$$

where

$$\begin{aligned} Y_{R(32)} &= ((X_{L(32)} \cap kl_{L(32)}) \lll_1) \oplus X_{R(32)}, \\ Y_{L(32)} &= (Y_{R(32)} \cup kl_{R(32)}) \oplus X_{L(32)}. \end{aligned}$$

4.3 *FL*$^{-1}$-function

The FL^{-1}-function is shown in figure 7, which is defined as follows:

$$\begin{aligned} FL^{-1} : \mathbf{L} \times \mathbf{L} &\longrightarrow \mathbf{L} \\ (Y_{L(32)} || Y_{R(32)}, kl_{L(32)} || kl_{R(32)}) &\longmapsto X_{L(32)} || X_{R(32)}, \end{aligned}$$

where

$$\begin{aligned} X_{L(32)} &= (Y_{R(32)} \cup kl_{R(32)}) \oplus Y_{L(32)}, \\ X_{R(32)} &= ((X_{L(32)} \cap kl_{L(32)}) \lll_1) \oplus Y_{R(32)}. \end{aligned}$$

4.4 *S*-function

The S-function is a part of F-function, which is defined as follows:

$$\begin{aligned} S : \mathbf{L} &\longrightarrow \mathbf{L} \\ l_{1(8)} || l_{2(8)} || l_{3(8)} || l_{4(8)} || l_{5(8)} || l_{6(8)} || l_{7(8)} || l_{8(8)} &\longmapsto l'_{1(8)} || l'_{2(8)} || l'_{3(8)} || l'_{4(8)} || l'_{5(8)} || l'_{6(8)} || l'_{7(8)} || l'_{8(8)} \end{aligned}$$

$$
\begin{aligned}
l'_{1(8)} &= s_1(l_{1(8)}),\\
l'_{2(8)} &= s_2(l_{2(8)}),\\
l'_{3(8)} &= s_3(l_{3(8)}),\\
l'_{4(8)} &= s_4(l_{4(8)}),\\
l'_{5(8)} &= s_2(l_{5(8)}),\\
l'_{6(8)} &= s_3(l_{6(8)}),\\
l'_{7(8)} &= s_4(l_{7(8)}),\\
l'_{8(8)} &= s_1(l_{8(8)}),
\end{aligned}
$$

where the four s-boxes, s_1, s_2, s_3, and s_4, are described in section 4.5.

4.5 s-boxes

The four s-boxes of *Camellia* are affine equivalent to an inversion function over $\mathrm{GF}(2^8)$, which are shown in tables 4, 5, 6, and 7. An algebraic representation of the s-boxes is shown below:

$$
\begin{aligned}
s_1 &: \mathbf{B} \longrightarrow \mathbf{B}\\
x_{(8)} &\longmapsto \mathbf{h}(\mathbf{g}(\mathbf{f}(\mathtt{0xc5} \oplus x_{(8)}))) \oplus \mathtt{0x6e},\\
s_2 &: \mathbf{B} \longrightarrow \mathbf{B}\\
x_{(8)} &\longmapsto s_1(x_{(8)}) \lll_1,\\
s_3 &: \mathbf{B} \longrightarrow \mathbf{B}\\
x_{(8)} &\longmapsto s_1(x_{(8)}) \ggg_1,\\
s_4 &: \mathbf{B} \longrightarrow \mathbf{B}\\
x_{(8)} &\longmapsto s_1(x_{(8)} \lll_1),
\end{aligned}
$$

where the functions $\mathbf{f}, \mathbf{g}$, and $\mathbf{h}$ are given as follows:

$$
\begin{aligned}
&\mathbf{f} : \mathbf{B} \longrightarrow \mathbf{B}\\
&a_{1(1)}||a_{2(1)}||a_{3(1)}||a_{4(1)}||a_{5(1)}||a_{6(1)}||a_{7(1)}||a_{8(1)}\\
&\longmapsto b_{1(1)}||b_{2(1)}||b_{3(1)}||b_{4(1)}||b_{5(1)}||b_{6(1)}||b_{7(1)}||b_{8(1)},
\end{aligned}
$$

where

$$
\begin{aligned}
b_1 &= a_6 \oplus a_2,\\
b_2 &= a_7 \oplus a_1,\\
b_3 &= a_8 \oplus a_5 \oplus a_3,\\
b_4 &= a_8 \oplus a_3,\\
b_5 &= a_7 \oplus a_4,\\
b_6 &= a_5 \oplus a_2,\\
b_7 &= a_8 \oplus a_1,\\
b_8 &= a_6 \oplus a_4.
\end{aligned}
$$

$$\begin{aligned}
&\mathbf{g} : \mathbf{B} \longrightarrow \mathbf{B}\\
&a_{1(1)}||a_{2(1)}||a_{3(1)}||a_{4(1)}||a_{5(1)}||a_{6(1)}||a_{7(1)}||a_{8(1)}\\
&\longmapsto b_{1(1)}||b_{2(1)}||b_{3(1)}||b_{4(1)}||b_{5(1)}||b_{6(1)}||b_{7(1)}||b_{8(1)},
\end{aligned}$$

where

$$\begin{aligned}
&(b_8 + b_7\alpha + b_6\alpha^2 + b_5\alpha^3) + (b_4 + b_3\alpha + b_2\alpha^2 + b_1\alpha^3)\beta\\
&= 1/((a_8 + a_7\alpha + a_6\alpha^2 + a_5\alpha^3) + (a_4 + a_3\alpha + a_2\alpha^2 + a_1\alpha^3)\beta).
\end{aligned}$$

This inversion is performed in $\mathrm{GF}(2^8)$ assuming $\frac{1}{0} = 0$, where β is an element in $\mathrm{GF}(2^8)$ that satisfies $\beta^8 + \beta^6 + \beta^5 + \beta^3 + 1 = 0$, and $\alpha = \beta^{238} = \beta^6 + \beta^5 + \beta^3 + \beta^2$ is an element in $\mathrm{GF}(2^4)$ that satisfies $\alpha^4 + \alpha + 1 = 0$.

$$\begin{aligned}
&\mathbf{h} : \mathbf{B} \longrightarrow \mathbf{B}\\
&a_{1(1)}||a_{2(1)}||a_{3(1)}||a_{4(1)}||a_{5(1)}||a_{6(1)}||a_{7(1)}||a_{8(1)}\\
&\longmapsto b_{1(1)}||b_{2(1)}||b_{3(1)}||b_{4(1)}||b_{5(1)}||b_{6(1)}||b_{7(1)}||b_{8(1)},
\end{aligned}$$

where

$$\begin{aligned}
b_1 &= a_5 \oplus a_6 \oplus a_2,\\
b_2 &= a_6 \oplus a_2,\\
b_3 &= a_7 \oplus a_4,\\
b_4 &= a_8 \oplus a_2,\\
b_5 &= a_7 \oplus a_3,\\
b_6 &= a_8 \oplus a_1,\\
b_7 &= a_5 \oplus a_1,\\
b_8 &= a_6 \oplus a_3.
\end{aligned}$$

Table 4: The s-box s_1

This table below reads $s_1(0) = 112, s_1(1) = 130, \ldots, s_1(255) = 158$.

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

Table 5: The s-box s_2

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

Table 6: The s-box s_3

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

Table 7: The s-box s_4

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

4.6 *P*-function

The P-function is a part of F-function, which is defined as follows:

$$\begin{array}{rcl} P : \mathbf{L} & \longrightarrow & \mathbf{L} \\ z_{1(8)}||z_{2(8)}||z_{3(8)}||z_{4(8)}||z_{5(8)}||z_{6(8)}||z_{7(8)}||z_{8(8)} & \longmapsto & z'_{1(8)}||z'_{2(8)}||z'_{3(8)}||z'_{4(8)}||z'_{5(8)}||z'_{6(8)}||z'_{7(8)}||z'_{8(8)}, \end{array}$$

where

$$\begin{aligned} z'_1 &= z_1 \oplus z_3 \oplus z_4 \oplus z_6 \oplus z_7 \oplus z_8, \\ z'_2 &= z_1 \oplus z_2 \oplus z_4 \oplus z_5 \oplus z_7 \oplus z_8, \\ z'_3 &= z_1 \oplus z_2 \oplus z_3 \oplus z_5 \oplus z_6 \oplus z_8, \\ z'_4 &= z_2 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7, \\ z'_5 &= z_1 \oplus z_2 \oplus z_6 \oplus z_7 \oplus z_8, \\ z'_6 &= z_2 \oplus z_3 \oplus z_5 \oplus z_7 \oplus z_8, \\ z'_7 &= z_3 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_8, \\ z'_8 &= z_1 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7. \end{aligned}$$

Equivalently, this transformation can be given in the following form:

$$\begin{pmatrix} z_8 \\ z_7 \\ \vdots \\ z_1 \end{pmatrix} \longmapsto \begin{pmatrix} z'_8 \\ z'_7 \\ \vdots \\ z'_1 \end{pmatrix} = P \begin{pmatrix} z_8 \\ z_7 \\ \vdots \\ z_1 \end{pmatrix},$$

where

$$P = \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}.$$

A Figures of the *Camellia* Algorithm

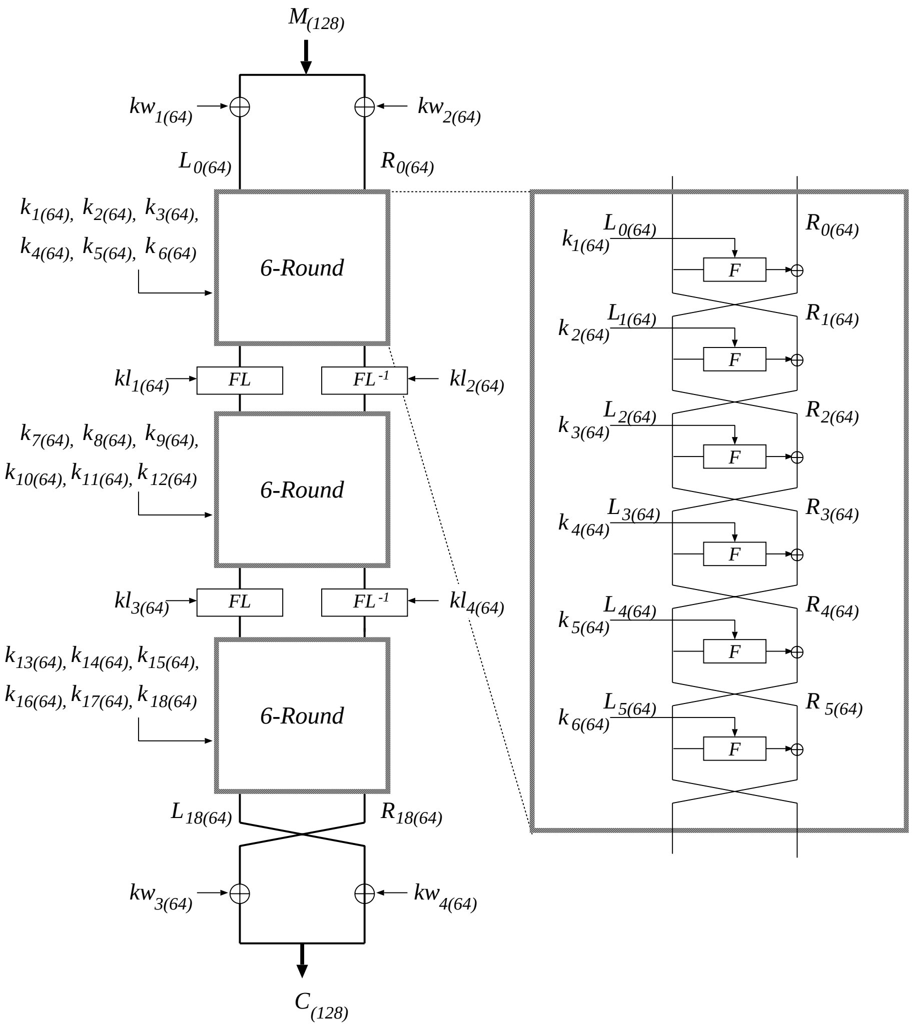

Figure 1: Encryption Procedure of *Camellia* for 128-bit key

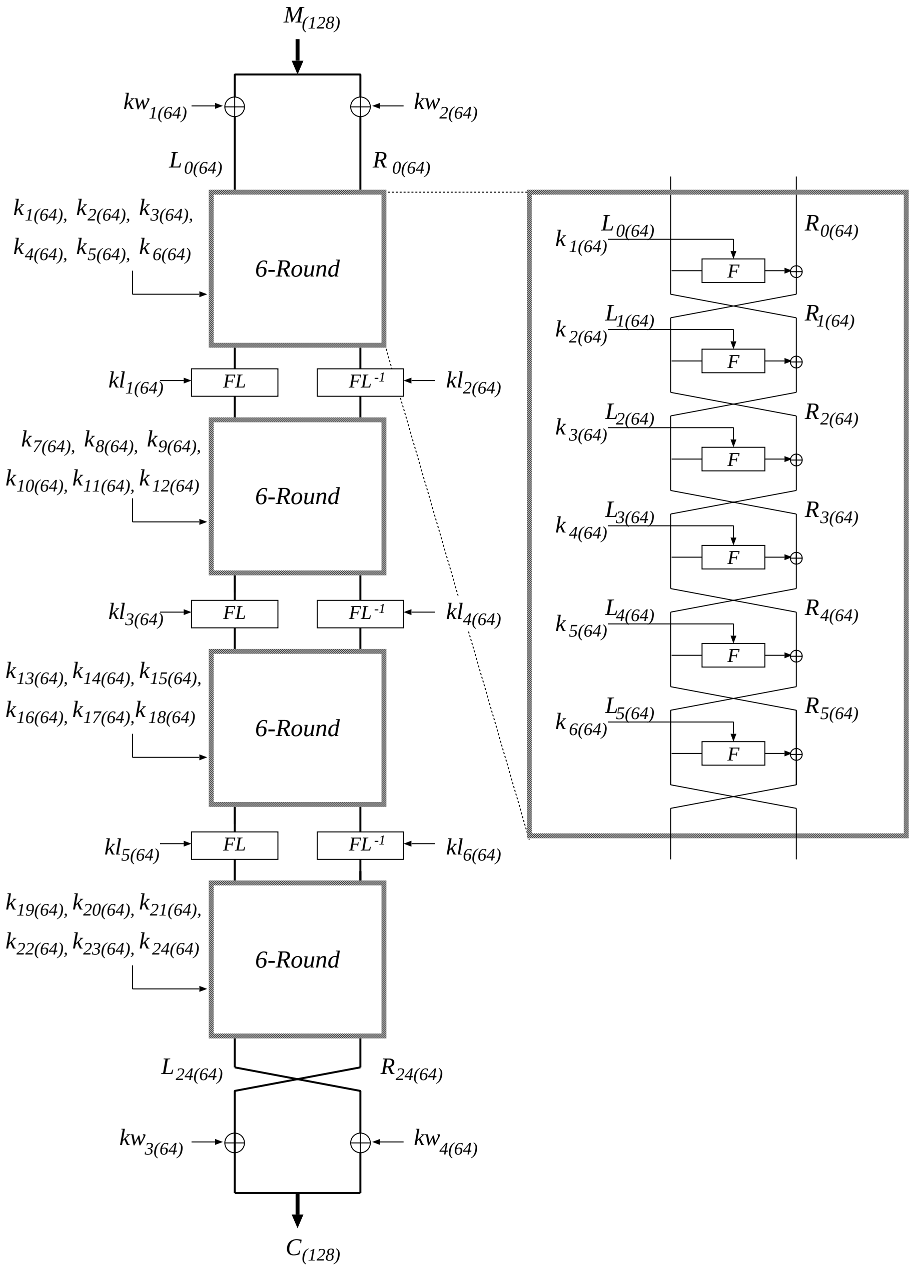

Figure 2: Encryption Procedure of *Camellia* for 192-bit and 256-bit key

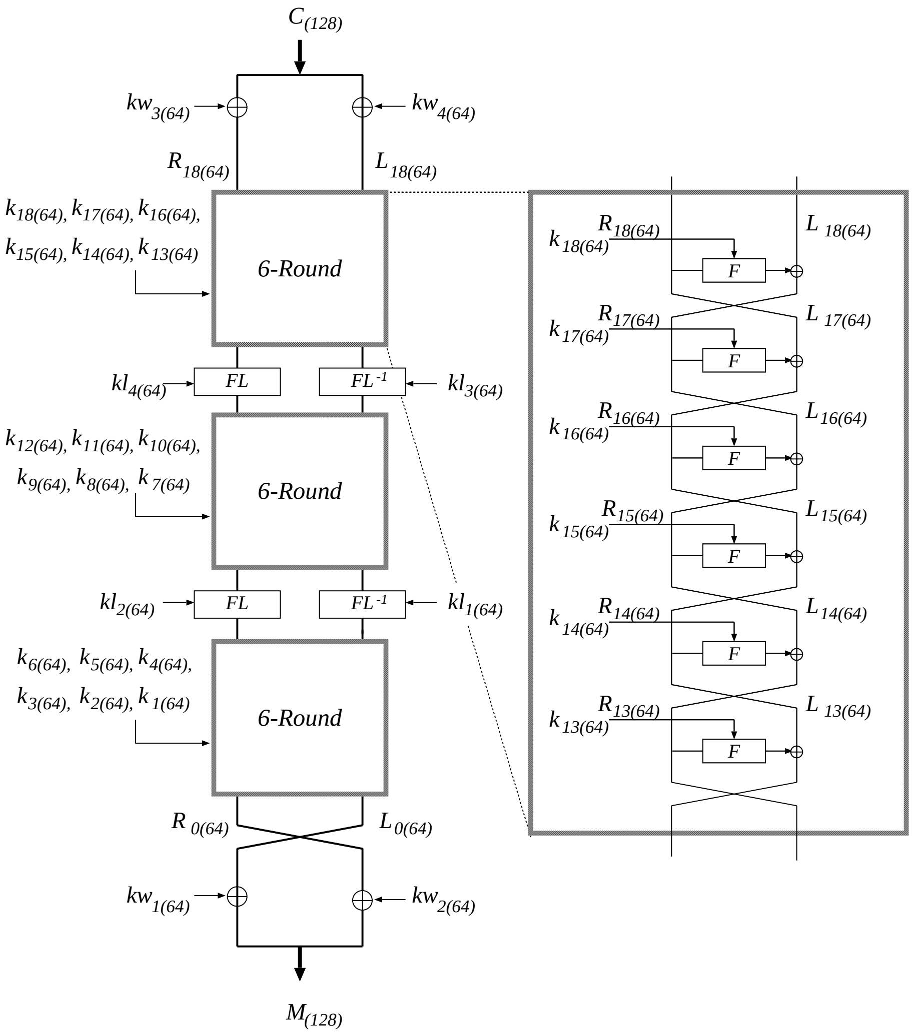

Figure 3: Decryption Procedure of *Camellia* for 128-bit key

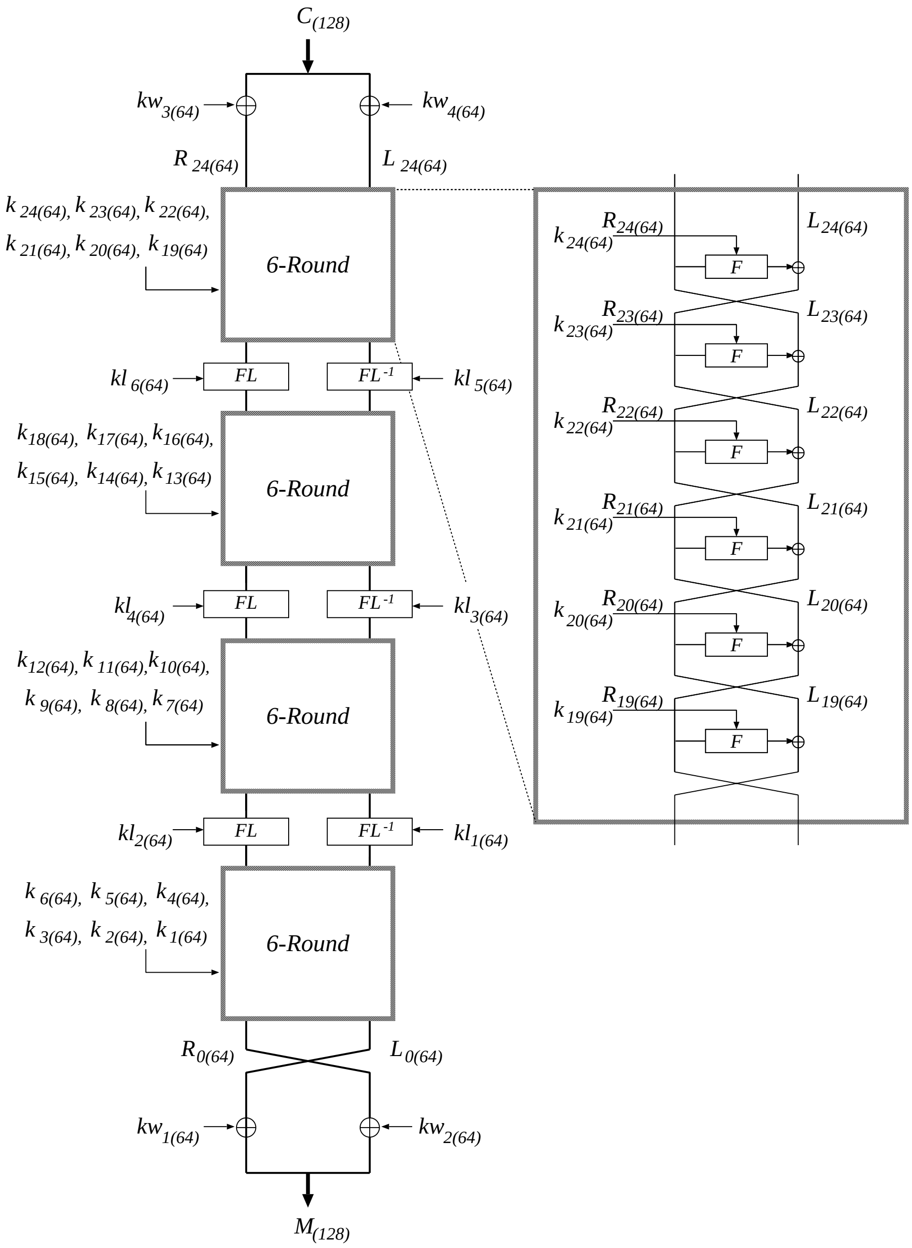

Figure 4: Decryption Procedure of *Camellia* for 192-bit and 256-bit key

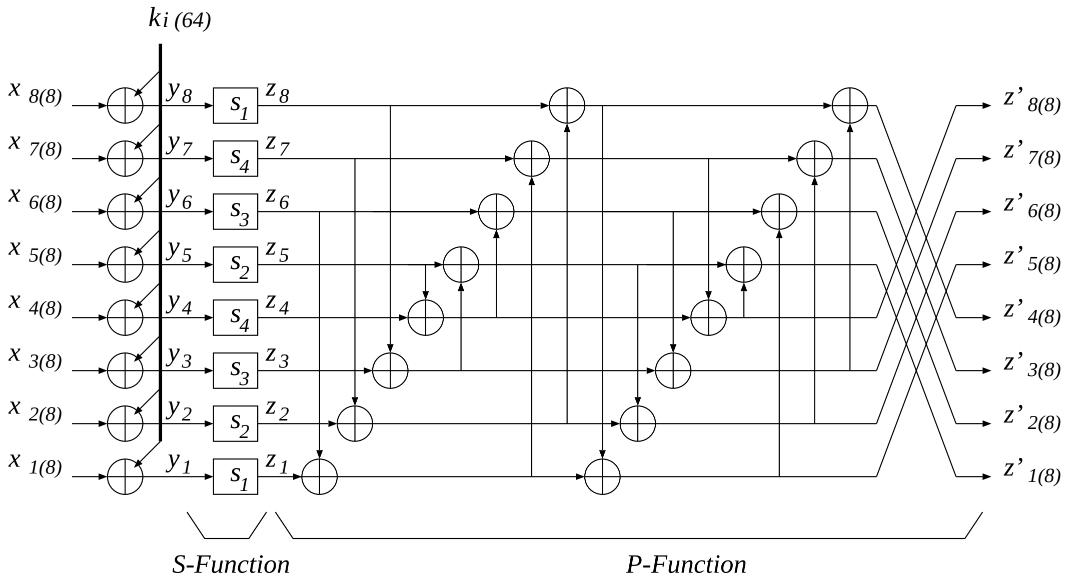

Figure 5: F-function

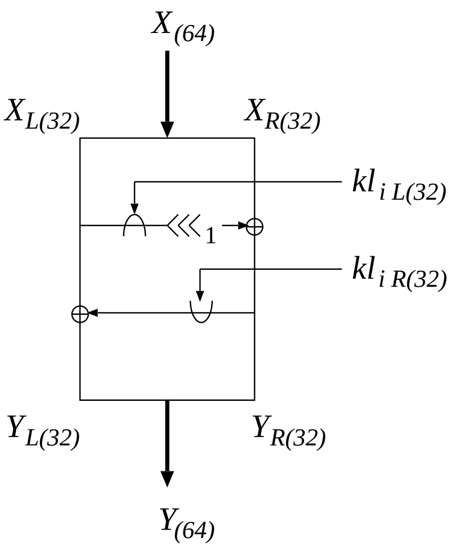

Figure 6: FL-function

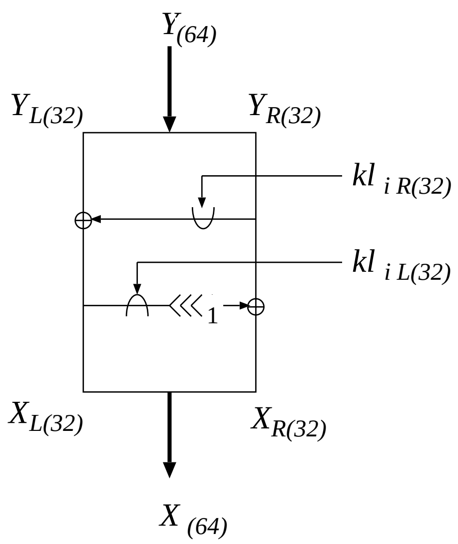

Figure 7: FL^{-1}-function

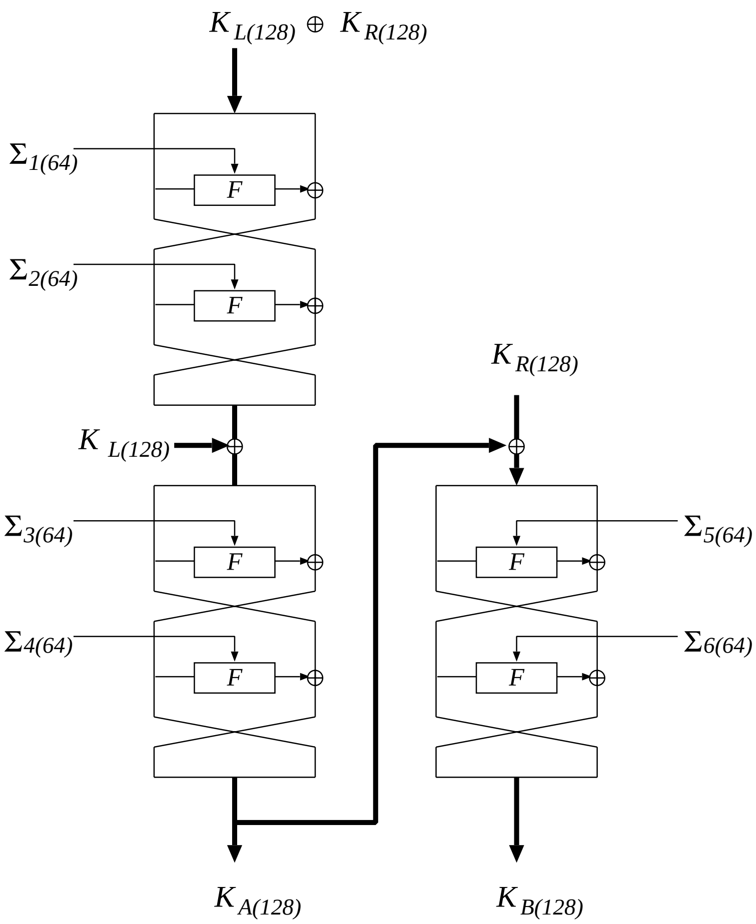

Figure 8: Key Schedule

B Test Data

The following is test data for *Camellia* in hexadecimal form:

128-bit key

key	`01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10`
plaintext	`01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10`
ciphertext	`67 67 31 38 54 96 69 73 08 57 06 56 48 ea be 43`

192-bit key

key	`01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10`
	`00 11 22 33 44 55 66 77`
plaintext	`01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10`
ciphertext	`b4 99 34 01 b3 e9 96 f8 4e e5 ce e7 d7 9b 09 b9`

256-bit key

key	`01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10`
	`00 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff`
plaintext	`01 23 45 67 89 ab cd ef fe dc ba 98 76 54 32 10`
ciphertext	`9a cc 23 7d ff 16 d7 6c 20 ef 7c 91 9e 3a 75 09`

C Software Implementation Techniques

This section describes how to implement Camellia efficiently in software. In most cases, an implementation can be divided into two parts: *setup* including key schedule and *data randomization*, that is, encryption or decryption. We first describe how to optimize the setup code, and then describe how to optimize the data randomization code.

This section describes specific techniques for 8-, 32-, or 64-bit processors. However, a technique for 8-bit processors may be applicable to 32- or 64-bit processors and a technique for 32-bit processors may be applicable to 64-bit processors. Other word sizes may need to be considered.

We assume that you first implement Camellia using the specification as it is. This section will optimize the resulting code.

Note that in this section "word" means the natural size of the target processor. For example, the words of IA-32 without MMX technology, IA-32 with MMX technology and Alpha are 32-, 64-, and 64-bits long respectively.

C.1 Setup

C.1.1 Store All Subkeys

Store all subkeys into memory once you generate them if you have sufficient memory, and use the stored subkeys for data randomization.

C.1.2 Subkey Generation Order

You do not have to compute subkeys in order. For example, when you compute subkeys for a 128-bit key, first compute the subkeys that only depend on K_L, and then compute subkeys that only depend on K_A. You can save registers or memory for storing K_A.

C.1.3 XOR Cancellation Property in Key Schedule

The key schedule of Camellia is based on the Feistel structure. Between the 2nd round and the 3rd round, K_L is XORed to an intermediate value. This structure causes cancellations of K_L. More precisely, the input of the 3rd round can be computed by the following equations.

$$\begin{cases} \text{(left half)} &= F(K_{LR} \oplus \text{(right half)}, \Sigma_2) \\ \text{(right half)} &= F(K_{LL}, \Sigma_1) \end{cases} \quad \text{for 128-bit keys}$$

$$\begin{cases} \text{(left half)} &= K_{RL} \oplus F(K_{LR} \oplus \text{(right half)}, \Sigma_2) \\ \text{(right half)} &= K_{RR} \oplus F(K_{LL} \oplus K_{RL}, \Sigma_1) \end{cases} \quad \text{for 192- and 256-bit keys}$$

Using the above equations, we can eliminate 3 and 2 XORs in **L** for 128- and 192/256-bit keys, respectively, compared to the straightforward implementation of the specification.

C.1.4 Rotation Bits for K_L, K_R, K_A, and K_B

You do not need to keep K_L, K_R, K_A, and K_B, but you should keep their rotated values when generating subkeys. You can generate subkeys by rotating the kept values by integral multiples of 16 ± 1 bits.

C.1.5 kl_5 and kl_6 generation from k_{11} and k_{12}

For 192- and 256- bit keys, you can use a word-oriented rotation to generate (kl_5, kl_6) from (k_{11}, k_{12}), since (kl_5, kl_6) equals $(k_{11}, k_{12}) \lll_{32}$. This saves a few instructions compared to general rotation.

C.1.6 On-the-fly Subkey Generation

You can generate subkeys *on-the-fly*. All subkeys are one of the rotated values of K_L, K_R, K_A, and K_B. Thus, you first generate K_L, K_R, K_A, and K_B, and then rotate them to get the subkeys. Refer Section C.1.4 for the rotated numbers of bits for K_L, K_R, K_A, and K_B.

C.1.7 128-bit key and 192/256-bit key

If your code does not need to use larger key sizes than 128 bits, you do not need to generate K_B. That is, you can omit the computations for the last two F-functions.

C.1.8 How to Rotate an Element in Q

8-bit processor. As stated in Section C.1.4, the rotation bits are integral multiples of 16 ± 1. Thus, you can rotate an element in **Q** by 16 ± 1 bits by rotating 1-bit left or right followed by a 2-byte move.

32-bit processor. Consider the use of a double precision shift instruction: `shrd` or `shld` if you are programming on IA-32.

C.1.9 F-function

Key schedule includes F-functions, but the main usage of the F-function is for data randomization. Refer to Section C.2.

C.1.10 Keyed Functions

Camellia has three keyed functions: bitwise XOR, bitwise OR, and bitwise AND. Consider the use of a self-modifying code, if possible.

C.2 Data Randomization

C.2.1 Endian Conversion

Camellia prefers big endian. Thus, the code for little endian processors needs additional code for endian conversions.

The most straightforward implementation converts the endian when loading a register from memory and storing a register to memory. Only FL- and FL^{-1}-functions are endian dependent. More precisely, only the 1-bit rotation in FL- or FL^{-1}-function is endian dependent. This means that you can convert endians just before or just after the 1-bit rotation with the appropriate subkey generation scheme. A combination of computing endian conversion and 1-bit rotation may increase the performance of Camellia using this technique. Details are described in Section C.2.2.

Some processors have a special instruction for endian conversion. For example, IA-32 (after 80486) has `bswap` instruction. Use these instructions. However, do not use the byte swap technique described in [C98, Appendix A]. The technique reduces the code size, but it is not fast, since the memory load and store instruction incurs long latency.

As described above, the endian problem only effects the 1-bit rotation of a 32-bit word. Thus, we do not need full 64-bit word endian conversion.

The following are general methods to realize endian conversion for 32-bit register x. In the following techniques, you can use either $\cup$ or $\oplus$ instead of $+$ in the equations, and you can switch the computational order between shifts including rotations and ANDs with an appropriate conversion of masked constants.

Straightforward.

$$x \leftarrow (x \ll_{24}) + ((x \cap \texttt{0xff00}) \ll_{8}) + ((x \gg_{8}) \cap \texttt{0xff00}) + (x \gg_{24})$$

The technique has high parallelism.

Minimum operations without rotation.

$$\begin{aligned} x &\leftarrow (x \ll_{16}) + (x \gg_{16}) \\ x &\leftarrow ((x \cap \texttt{0xff00ff}) \ll_{8}) + ((x \gg_{8}) \cap \texttt{0xff00ff}) \end{aligned}$$

Using rotations.

$$x \leftarrow ((x \cap \texttt{0xff00ff}) \ggg_8) + ((x \lll_8) \cap \texttt{0xff00ff})$$

Using SSE. New Intel Pentium family processors including Pentium III has very effective instruction for reordering data, which is called `pshufw` [I99]. 5 instructions including `pshufw` are sufficient to convert endian for 64-bit data.

C.2.2 1-bit Rotation in Little Endian Interpretation

As described in Section C.2.1, we do not need endian conversion when loading and storing texts if we can efficiently implement 1-bit rotation in FL- and FL^{-1}-functions.

Considering x as a 32-bit register that contains little endian data to be rotated by 1-bit. We can compute 1-bit rotation by the following equation.

$$x \leftarrow ((2x) \cap \texttt{0xfefefefe}) + ((x \ggg_{15}) \cap \overline{\texttt{0xfefefefe}}) \tag{1}$$

Of course, this technique requires an appropriate changes to subkey setup and other functions.

Note that $+$ in Equation (1) can be replaced with $\cup$ or $\oplus$, and computing $2x$ can be done by $\ll_1$, $\lll_1$ or addition with x itself, and you can switch the computational order between shifts including rotations and ANDs with an appropriate conversion of masked constants.

Confirm whether your processor has ANDNOT instruction, such as `pandn` in IA-32 and `bic` in Alpha. In this case, you do not need to prepare the constant, $\overline{\texttt{0xfefefefe}}$.

C.2.3 Whitening

The key additions kw_2 and kw_4 can be combined into other keyed operations using the following equations.

$$\begin{aligned}
(x \oplus k) \oplus y &= (x \oplus y) \oplus k, \\
(x \oplus k) \oplus l &= x \oplus (k \oplus l), \\
(x \oplus k) \cap l &= (x \cap l) \oplus (k \cap l), \\
(x \oplus k) \lll_1 &= (x \lll_1) \oplus (k \lll_1), \\
(x \oplus k) \cup l &= (x \cup l) \oplus (k \cap \bar{l}),
\end{aligned} \tag{2}$$

where x, y, k, l are bit strings. Adjust subkeys at setup to eliminate 2 XORs in **L**.

C.2.4 Key XOR

Using Equations (2), you can move key XORs to any place if the movement does not go through the S-function. For example, changing F-function computation $P(S(X \oplus k))$ to $P(S(X)) \oplus k'$ may improve instruction scheduling.

C.2.5 S-function

s_1 is defined by the arithmetics in GF(2^8). However, do not compute GF(2^8) arithmetics; instead precompute and hard-code a table in your program, see Table 4 in the specification.

We strongly suggest that you also precompute and hard-code s_2, s_3, and s_4 tables in addition to s_1, if you have a sufficient memory and 8-bit rotation is expensive. If you do not have sufficient memory, please compute s_2, s_3, and s_4 from looked-up values in s_1 table using rotation.

If you have sufficient memory, and cost of table lookup is heavy as is true for Java, consider the use of a two s-box combined table, for example $(s_1(y_1), s_2(y_2))$.

C.2.6 P-function

32-bit processor. Let $(Z_L, Z_R) = ((z_1, z_2, z_3, z_4), (z_5, z_6, z_7, z_8))$ be the input of P-function and $(Z'_L, Z'_R) = ((z'_1, z'_2, z'_3, z'_4), (z'_5, z'_6, z'_7, z'_8))$ be the output of P-function.

From Figure 5 in the specification, you can see that P-function can be computed as follows.

$$
\begin{array}{lcl}
Z_L & \leftarrow & Z_L \oplus (Z_R \lll_8) \\
Z_R & \leftarrow & Z_R \oplus (Z_L \lll_{16}) \\
Z_L & \leftarrow & Z_L \oplus (Z_R \ggg_8) \\
Z_R & \leftarrow & Z_R \oplus (Z_L \ggg_8) \\
Z'_L & \leftarrow & Z_R \\
Z'_R & \leftarrow & Z_L
\end{array}
$$

The critical path of the new computation is long. We can modify the computation as follows.

$$
\begin{array}{lcl lcl}
 & & & Z_R & \leftarrow & Z_R \lll_8 \\
Z_L & \leftarrow & Z_L \oplus Z_R & Z_R & \leftarrow & Z_R \lll_8 \\
Z_L & \leftarrow & Z_L \ggg_8 & Z_R & \leftarrow & Z_R \oplus Z_L \\
Z_L & \leftarrow & Z_L \oplus Z_R & Z_R & \leftarrow & Z_R \lll_{16} \\
Z_L & \leftarrow & Z_L \lll_8 & Z_R & \leftarrow & Z_R \oplus Z_L \\
Z'_L & \leftarrow & Z_R & Z'_R & \leftarrow & Z_L
\end{array}
$$

The critical path of the above computation is decreased. It seems that the technique requires one additional rotation, however, you can probably combine the first step of the above computation and S-function without any additional cost.

8-bit processor (orthogonal mnemonics). If the instruction in your processor can XOR any combination of registers and has sufficient registers, you can compute P-function by using just 16 XORs using Figure 5 in the specification.

8-bit processor (accumulator based). If your processor is accumulator based, minimizing the number of XORs is not always a good idea, since the computation may require register load from memory and store into memory many times. The following computation is optimized for an accumulator based processor.

$$z'_8 \leftarrow z_1 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7$$

$$
\begin{aligned}
z'_4 &\leftarrow z'_8 \oplus z_1 \oplus z_2 \oplus z_3 \\
z'_7 &\leftarrow z'_4 \oplus z_2 \oplus z_7 \oplus z_8 \\
z'_3 &\leftarrow z'_7 \oplus z_1 \oplus z_2 \oplus z_4 \\
z'_6 &\leftarrow z'_3 \oplus z_1 \oplus z_6 \oplus z_7 \\
z'_2 &\leftarrow z'_6 \oplus z_1 \oplus z_3 \oplus z_4 \\
z'_5 &\leftarrow z'_2 \oplus z_4 \oplus z_5 \oplus z_6 \\
z'_1 &\leftarrow z'_5 \oplus z_2 \oplus z_3 \oplus z_4
\end{aligned}
$$

When indexing z'_i costs many operations, the following is useful.

$$
\begin{aligned}
\sigma &\leftarrow z_1 \oplus z_2 \oplus z_3 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7 \oplus z_8 \\
z'_1 &\leftarrow \sigma \oplus z_2 \oplus z_5 \\
z'_2 &\leftarrow \sigma \oplus z_3 \oplus z_6 \\
z'_3 &\leftarrow \sigma \oplus z_4 \oplus z_7 \\
z'_4 &\leftarrow \sigma \oplus z_1 \oplus z_8 \\
z'_5 &\leftarrow \sigma \oplus z_3 \oplus z_4 \oplus z_5 \\
z'_6 &\leftarrow \sigma \oplus z_1 \oplus z_4 \oplus z_6 \\
z'_7 &\leftarrow \sigma \oplus z_1 \oplus z_2 \oplus z_7 \\
z'_8 &\leftarrow \sigma \oplus z_2 \oplus z_3 \oplus z_8
\end{aligned}
$$

C.2.7 Substitution and Permutation

This section describes how to efficiently compute $P \circ S$ compared to independently computing S and P.

64-bit processor. If your processor has a sufficiently large first level cache, use the technique described in [RDP+96]. The technique prepares the following tables defined by equation (3).

$$
\begin{array}{rcl}
SP_1(y_1) &=& (s_1(y_1),\ s_1(y_1),\ s_1(y_1),\ 0,\ s_1(y_1),\ 0,\ 0,\ s_1(y_1)) \\
SP_2(y_2) &=& (0,\ s_2(y_2),\ s_2(y_2),\ s_2(y_2),\ s_2(y_2),\ s_2(y_2),\ 0,\ 0) \\
SP_3(y_3) &=& (s_3(y_3),\ 0,\ s_3(y_3),\ s_3(y_3),\ 0,\ s_3(y_3),\ s_3(y_3),\ 0) \\
SP_4(y_4) &=& (s_4(y_4),\ s_4(y_4),\ 0,\ s_4(y_4),\ 0,\ 0,\ s_4(y_4),\ s_4(y_4)) \\
SP_5(y_5) &=& (0,\ s_2(y_5),\ s_2(y_5),\ s_2(y_5),\ 0,\ s_2(y_5),\ s_2(y_5),\ s_2(y_5)) \\
SP_6(y_6) &=& (s_3(y_6),\ 0,\ s_3(y_6),\ s_3(y_6),\ s_3(y_6),\ 0,\ s_3(y_6),\ s_3(y_6)) \\
SP_7(y_7) &=& (s_4(y_7),\ s_4(y_7),\ 0,\ s_4(y_7),\ s_4(y_7),\ s_4(y_7),\ 0,\ s_4(y_7)) \\
SP_8(y_8) &=& (s_1(y_8),\ s_1(y_8),\ s_1(y_8),\ 0,\ s_1(y_8),\ s_1(y_8),\ s_1(y_8),\ 0)
\end{array}
\tag{3}
$$

Next, compute the following equation:

$$(z'_1, z'_2, z'_3, z'_4, z'_5, z'_6, z'_7, z'_8) \leftarrow \bigoplus_{i=1}^{8} SP_i(y_i)$$

This technique requires the following operations.

# of Table Lookups	8
# of XORs	7
Size of Table (KB)	16

If the first cache of the target processor is moderately large, replace a few of tables defined by equation (3) with the tables below.

$$\begin{aligned}
SP_\alpha(y) &= (s_1(y),\, s_1(y),\, s_1(y),\, s_1(y),\, s_1(y),\, s_1(y),\, s_1(y),\, s_1(y))\\
SP_\beta(y) &= (s_2(y),\, s_2(y),\, s_2(y),\, s_2(y),\, s_2(y),\, s_2(y),\, s_2(y),\, s_2(y))\\
SP_\gamma(y) &= (s_3(y),\, s_3(y),\, s_3(y),\, s_3(y),\, s_3(y),\, s_3(y),\, s_3(y),\, s_3(y))\\
SP_\delta(y) &= (s_4(y),\, s_4(y),\, s_4(y),\, s_4(y),\, s_4(y),\, s_4(y),\, s_4(y),\, s_4(y))
\end{aligned} \tag{4}$$

Then, mask the necessary byte positions. This technique requires the following operations if you use only tables of equation (4).

# of Table Lookups	8
# of XORs	7
# of ANDs	8
Size of Table (KB)	8

When implementing this technique on Alpha architecture [C98], and if the number of registers is insufficient for storing constants for masking operation, use `zap` or `zapnot` instructions.

If your processor can efficiently copy half bits of a register to the other half, for example, `punpckldq/punpckhdq` or `pshufw` instructions in IA-32 [I99] which are realized after Pentium with MMX technology and Pentium III, respectively, prepare SP_1, SP_2, SP_3, and SP_4 defined in equation (3). Then, compute the following equation:

$$\begin{aligned}
&(z'_1, z'_2, z'_3, z'_4, z'_5, z'_6, z'_7, z'_8)\\
&\leftarrow SP_1(y_1) \oplus SP_2(y_2) \oplus SP_3(y_3) \oplus SP_4(y_4) \oplus \nu(SP_1(y_8) \oplus SP_2(y_5) \oplus SP_3(y_6) \oplus SP_4(y_7)),
\end{aligned}$$

where ν denotes the operation that copies the first 4 bytes to the last 4 bytes. This technique requires the following operations.

# of Table Lookups	8
# of XORs	7
# of νs	1
Size of Table (KB)	8

32-bit processor. [AU00] shows efficient implementations of Camellia-type substitution and permutation networks. One of the technique prepares the following tables defined by equation (5):

$$\begin{aligned}
SP_{1110}(y) &= (s_1(y),\, s_1(y),\, s_1(y),\, 0)\\
SP_{0222}(y) &= (0,\, s_2(y),\, s_2(y),\, s_2(y))\\
SP_{3033}(y) &= (s_3(y),\, 0,\, s_3(y),\, s_3(y))\\
SP_{4404}(y) &= (s_4(y),\, s_4(y),\, 0,\, s_4(y))
\end{aligned} \tag{5}$$

Then, compute as follows:

$$\begin{aligned}
D &\leftarrow SP_{1110}(y_8) \oplus SP_{0222}(y_5) \oplus SP_{3033}(y_6) \oplus SP_{4404}(y_7) \\
U &\leftarrow SP_{1110}(y_1) \oplus SP_{0222}(y_2) \oplus SP_{3033}(y_3) \oplus SP_{4404}(y_4) \\
(z'_1, z'_2, z'_3, z'_4) &\leftarrow D \oplus U \\
(z'_5, z'_6, z'_7, z'_8) &\leftarrow (z'_1, z'_2, z'_3, z'_4) \oplus (U \ggg_8)
\end{aligned}$$

This technique requires the following operations.

# of Table Lookups	8
# of XORs	8
# of Rotations	1
Size of Table (KB)	4

[AU00] also shows an implementation that is suitable for a processor in which rotation is very costly. The technique prepares the following tables in addition to tables defined by equation (5):

$$\begin{aligned}
SP_{1001}(y) &= (s_1(y), \quad 0, \quad 0, s_1(y)) \\
SP_{2200}(y) &= (s_2(y), s_2(y), \quad 0, \quad 0) \\
SP_{0330}(y) &= (\quad 0, s_3(y), s_3(y), \quad 0) \\
SP_{0044}(y) &= (\quad 0, \quad 0, s_4(y), s_4(y))
\end{aligned}$$

Then, compute as follows:

$$\begin{aligned}
D &\leftarrow SP_{1110}(y_8) \oplus SP_{0222}(y_5) \oplus SP_{3033}(y_6) \oplus SP_{4404}(y_7) \\
(z'_1, z'_2, z'_3, z'_4) &\leftarrow D \oplus SP_{1110}(y_1) \oplus SP_{0222}(y_2) \oplus SP_{3033}(y_3) \oplus SP_{4404}(y_4) \\
(z'_5, z'_6, z'_7, z'_8) &\leftarrow D \oplus SP_{1001}(y_1) \oplus SP_{2200}(y_2) \oplus SP_{0330}(y_3) \oplus SP_{0044}(y_4)
\end{aligned}$$

This technique requires the following operations.

# of Table Lookups	12
# of XORs	11
Size of Table (KB)	8

C.2.8 Making Indices for s-box

You can make an index for s-box by simply using shifts and ANDs. However, several processors have special instructions for making an index, for example, `movzx` in IA-32 [I99], `extbl` in Alpha [C98].

`movzx` is a fast operation in P6, but it can be used only for the two least significant bytes. A straightforward implementation uses `eax`, `ebx`, `ecx`, and `edx` registers for storing (L_r, R_r), and 2 rotations are used for making indices; 2 rotations are used for recovering byte order in the registers every round. However, you can remove 2 rotations for recovering byte order every round if you prepare rotated tables. Note that the byte order in registers returns to a natural order every 4 rounds.

C.3 General Guidelines

This section describes general guidelines. The guidelines are useful to optimize Camellia as well as other block ciphers. Please refer to the optimization manuals for each processor.

Avoid misaligned data accesses. Almost all processors penalize misaligned data access. Align data to the word boundary.

Avoid partial data accesses. Most processors have a function to access a smaller part than word size. However, this function may cause a penalty. Do not access partial data, even if you do not need full size of word and you have sufficient memory.

Be careful of the size of the cache. If the program or its data exceeds the size of the cache, the speed of the program will significantly decrease. Loop unrolling and table expansion are good techniques to speed up the program, but do not exceed the size of the cache.

Use intrinsic functions. Several compilers support intrinsic functions. For example, when you use Microsoft Visual C++ 6 compiler on IA-32, and declare "`#pragma intrinsic(_lrotl)`" and use "`_lrotl`", the compiler generates rotation instructions in assembly language. Refer to the manual of the compiler that you use for details.

Measuring precise speeds is difficult. The running time of your code depends on many factors: cache hit misses, OS interrupts, and so on. Furthermore, the cryptographic properties, for example, the number of blocks to be encrypted, also effect the running time.

A few processors have an instruction to get the time stamp. For example, IA-32 (after Pentium) has `rdtsc` [I99] and Alpha has `rpcc` [C98]. It is a good idea to use the time stamp counter for measuring speeds, but you should not directly apply these instructions to out-of-order architectures such as P6 and EV6.

If you want to measure speed precisely, consult good guidebooks. For example, if you use Pentium family processors, refer to [F00].

D Design Policy

This paper presents a 128-bit block cipher called *Camellia*, which was jointly developed by NTT and Mitsubishi Electric Corporation. Camellia supports 128-bit block size and 128-, 192-, and 256-bit key lengths, and so offers the same interface specifications as the Advanced Encryption Standard (AES). The design goals of Camellia are as follows.

High level of security. The recent advances in cryptanalytic techniques are remarkable. A quantitative evaluation of security against powerful cryptanalytic techniques such as differential cryptanalysis [BS93] and linear cryptanalysis [M94] is considered to be essential in designing any new block cipher. We evaluated the security of Camellia by utilizing state-of-art cryptanalytic techniques. We have confirmed that Camellia has no differential and linear characteristics that hold with probability more than 2^{-128}. Moreover, Camellia was designed to offer security against other advanced cryptanalytic attacks including higher order differential attacks [K95, JK97], interpolation attacks [JK97, A00], related-key attacks [B94, KSW96], truncated differential attacks [K95, MT99], boomerang attacks [W99], and slide attacks [BW99, BW00].

Efficiency on multiple platforms. As cryptographic systems are needed in various applications, encryption algorithms that can be implemented efficiently on a wide range of platforms are desirable, however, few 128-bit block ciphers are suitable for both software and hardware implementation. Camellia was designed to offer excellent efficiency in hardware and software implementations, including gate count for hardware design, memory requirements in smart card implementations, as well as performance on multiple platforms.

Camellia consists of only 8-by-8-bit substitution tables (s-boxes) and logical operations that can be efficiently implemented on a wide variety of platforms. Therefore, it can be implemented efficiently in software, including the 8-bit processors used in low-end smart cards, 32-bit processors widely used in PCs, and 64-bit processors. Camellia doesn't use 32-bit integer additions and multiplications, which are extensively used in some software-oriented 128-bit block ciphers. Such operations perform well on platforms providing a high degree of support, e.g., Pentium II/III or Athlon, but not as well on others. These operations can cause a longer critical path and larger hardware implementation requirements.

The s-boxes of Camellia are designed to minimize hardware size. The four s-boxes are affine equivalent to the inversion function in the finite field $\mathrm{GF}(2^8)$. Moreover, we reduced the inversion function in $\mathrm{GF}(2^8)$ to a few $\mathrm{GF}(2^4)$ arithmetic operations. It enabled us to implement the s-boxes by fewer gate counts.

The key schedule is very simple and shares part of its procedure with encryption. It supports on-the-key subkey generation and subkeys are computable in any order. The memory requirement for generating subkeys is quite small; an efficient implementation requires about 32-byte RAM for 128-bit keys and about 64-byte RAM for 192- and 256-bit keys.

E Design Rationale

E.1 *F*-function

The design strategy of the F-function of Camellia follows that of the F-function of E2 [KMA+98]. The main difference between E2 and Camellia is the adoption of the 1-round (conservative) SPN (Substitution-Permutation Network), not the 2-round SPN, i.e. S-P-S. When the 1-round SPN is used as the round function in a Feistel cipher, the theoretical evaluation of the upper bound of differential and linear characteristic probability becomes more complicated, but the speed under the same level of "real" security is expected to be improved. See Section 6 for detailed discussions on security.

E.2 *P*-function

The design rationale of the P-function is similar to that of the P-function of E2. That is, for computational efficiency, it should be represented using only bytewise exclusive-ORs and for security against differential and linear cryptanalysis, its branch number should be optimal [KTM+99]. From among the linear transformations that satisfy these conditions, we chose one considering highly efficient implementation on 32-processors [AU00] and high-end smart cards, as well as 8-bit processors.

E.3 s-boxes

As the s-boxes we adopted functions affine equivalent to the inversion function in GF(2^8) for enhanced security and small hardware design.

It is well known that the smallest of the maximum differential probability of functions in GF(2^8) was proven to be 2^{-6}, and the smallest of the maximum linear probability of functions in GF(2^8) is conjectured to be 2^{-6}. There is a function affine equivalent to the inversion function in GF(2^8) that achieves the best known of the maximum differential and linear probabilities, 2^{-6}. We choose this kind of functions as s-boxes. Moreover, the high degree of the Boolean polynomial of every output bit of the s-boxes makes it difficult to attack Camellia by higher order differential attacks. The two affine functions that are performed at the input and output of the inversion function in GF(2^8) complicates

F Version Information

Camellia has been proposed in the following activities, where the proposed specification is exactly the same as the specification described in this document.

Papers

- Technical report of IEICE,
  K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita, "Camellia – A 128-bit Block Cipher", Technical Report ISEC2000-6, The Institute of Electronics, Information and Communication Engineers, 2000. (in Japanese).
- International Workshop SAC 2000
  K. Aoki, T. Ichikawa, M. Kanda, M. Matsui, S. Moriai, J. Nakajima, and T. Tokita, "Camellia: A 128-Bit Block Cipher Suitable for Multiple Platforms — Design and Analysis —," In Selected Areas in Cryptography, 7th Annual International Workshop, SAC 2000, Waterloo, Ontario, Canada, August 2000, Proceedings, Lecture Notes in Computer Science 2012, pp.39-56, Springer-Verlag, 2001.

Standardization

- ISO 18033
- NESSIE
- IETF
  The followings were submitted as Internet-Drafts.
  - J. Nakajima and S. Moriai,"A Description of the Camellia Encryption Algorithm"
    `<draft-nakajima-camellia-02.txt>`
  - S. Moriai,"Addition of the Camellia Encryption Algorithm to TLS"
    `<draft-ietf-tls-camellia-01.txt>`

G Object Identifier

The object identifier of Camellia is described in the Internet-Draft, " A Description of the Camellia Encryption Algorithm ". The following is extracted from the document.

The Object Identifier for Camellia in Cipher Block Chaining (CBC) mode is as follows:

- 128-bit key length, CBC mode
  `id-camellia128-cbc OBJECT IDENTIFIER ::=`
  `{ iso(1) member-body(2) 392 200011 61 security(1)`
  `algorithm(1) symmetric-encryption-algorithm(1) camellia128-cbc(2) }`
- 192-bit ke length, CBC mode
  `id-camellia192-cbc OBJECT IDENTIFIER ::=`
  `{ iso(1) member-body(2) 392 200011 61 security(1)`
  `algorithm(1) symmetric-encryption-algorithm(1) camellia192-cbc(3) }`
- 256-bit key length, CBC mode
  `id-camellia256-cbc OBJECT IDENTIFIER ::=`
  `{ iso(1) member-body(2) 392 200011 61 security(1)`
  `algorithm(1) symmetric-encryption-algorithm(1) camellia256-cbc(4) }`

H Applications and Products

Camellia can be used for all applications of symmetric block ciphers. In particular, it is suitable for secure communications and authentication.

Camellia can be implemented efficiently on a wide range of platforms, including software implementations on 32-bit/64-bit CPUs and low-end/high-end smart cards, and compact and high-speed hardware implementations on ASICs and FPGAs.

Most of the information about applications of Camellia can be found at `http://www.security.melco.co.jp/`

References

[A00] K. Aoki. Practical Evaluation of Security against Generalized Interpolation Attack. *IEICE Transactions Fundamentals of Electronics, Communications and Computer Sciences (Japan)*, Vol. E83-A, No. 1, pp. 33–38, 2000. (A preliminary version was presented at SAC'99).

[AU00] K. Aoki and H. Ueda. Optimized Software Implementations of E2. *IEICE Transactions Fundamentals of Electronics, Communications and Computer Sciences (Japan)*, Vol. E83-A, No. 1, pp. 101–105, 2000. (The full paper is available on `http://info.isl.ntt.co.% linebreak[3]jp/e2/RelDocs/`).

[B94] E. Biham. New Types of Cryptanalytic Attacks Using Related Keys. *Journal of Cryptology*, Vol. 7, No. 4, pp. 229–246, 1994. (The extended abstract was appeared at EUROCRYPT'93).

[BS93] E. Biham and A. Shamir. *Differential Cryptanalysis of the Data Encryption Standard.* Springer-Verlag, Berlin, Heidelberg, New York, 1993.

[BW99] A. Biryukov and D. Wagner. Slide Attacks. In L. Knudsen, editor, *Fast Software Encryption — 6th International Workshop, FSE'99*, Volume 1636 of *Lecture Notes in Computer Science*, pp. 245–259, Berlin, Heidelberg, New York, 1999. Springer-Verlag.

[BW00] A. Biryukov and D. Wagner. Advanced Slide Attacks. In S. Vaudenay, editor, *Advances in Cryptology — EUROCRYPT2000*, Volume 1807 of *Lecture Notes in Computer Science*, pp. 589–606, Berlin, Heidelberg, New York, 2000. Springer-Verlag.

[C98] Compaq Computer Corporation. *Alpha Architecture Handbook (Version 4)*, 1998. (You can download the manual from Compaq's technical documentation library: `http://www.support.compaq.com/alpha-tools/documentation/current/chip-docs.html`).

[F00] A. Fog. *How to optimize for the Pentium microprocessors*, 2000. (`http://www.agner.org/assem/`).

[I99] Intel Corporation. *Intel Architecture Software Developer's Manual (Volume 2: Instruction Set Reference)*, 1999. (You can download the manual from Intel's developer site: `http://developer.intel.com/`).

[JK97] T. Jakobsen and L. R. Knudsen. The Interpolation Attack on Block Cipher. In E. Biham, editor, *Fast Software Encryption — 4th International Workshop, FSE'97*, Volume 1267 of *Lecture Notes in Computer Science*, pp. 28–40, Berlin, Heidelberg, New York, 1997. Springer-Verlag.

[K95] L. R. Knudsen. Truncated and Higher Order Differentials. In B. Preneel, editor, *Fast Software Encryption — Second International Workshop*, Volume 1008 of *Lecture Notes in Computer Science*, pp. 196–211. Springer-Verlag, Berlin, Heidelberg, New York, 1995.

[KMA+98] M. Kanda, S. Moriai, K. Aoki, H. Ueda, M. Ohkubo, Y. Takashima, K. Ohta, and T. Matsumoto. A New 128-bit Block Cipher ***E2***. Technical Report ISEC98-12, The Institute of Electronics, Information and Communication Engineers, 1998. (in Japanese).

[KSW96] J. Kelsey, B. Schneier, and D. Wagner. Key-Schedule Cryptanalysis of IDEA, G-DES, GOST, SAFER, and Triple-DES. In N. Koblitz, editor, *Advances in Cryptology — CRYPTO'96*, Volume 1109 of *Lecture Notes in Computer Science*, pp. 237–251. Springer-Verlag, Berlin, Heidelberg, New York, 1996.

[KTM+99] M. Kanda, Y. Takashima, T. Matsumoto, K. Aoki, and K. Ohta. A Strategy for Constructing Fast Round Functions with Practical Security against Differential and Linear Cryptanalysis. In S. Tavares and H. Meijer, editors, *Selected Areas in Cryptography — 5th Annual International Workshop, SAC'98*, Volume 1556 of *Lecture Notes in Computer Science*, pp. 264–279, Berlin, Heidelberg, New York, 1999. Springer-Verlag.

[M94] M. Matsui. Linear Cryptanalysis Method for DES Cipher. In T. Helleseth, editor, *Advances in Cryptology — EUROCRYPT'93*, Volume 765 of *Lecture Notes in Computer Science*, pp. 386–397. Springer-Verlag, Berlin, Heidelberg, New York, 1994. (A preliminary version written in Japanese was presented at SCIS93-3C).

[MT99] M. Matsui and T. Tokita. Cryptanalysis of a Reduced Version of the Block Cipher E2. In L. Knudsen, editor, *Fast Software Encryption — 6th International Workshop, FSE'99*, Volume 1636 of *Lecture Notes in Computer Science*, pp. 71–80, Berlin, Heidelberg, New York, 1999. Springer-Verlag. (Japanese version was presented at SCIS99.).

[RDP+96] V. Rijmen, J. Daemen, B. Preneel, A. Bosselaers, and E. De Win. The Cipher SHARK. In D. Gollmann, editor, *Fast Software Encryption — Third International Workshop*, Volume 1039 of *Lecture Notes in Computer Science*, pp. 99–111. Springer-Verlag, Berlin, Heidelberg, New York, 1996.

[W99] D. Wagner. The Boomerang Attack. In L. R. Knudsen, editor, *Fast Software Encryption — 6th International Workshop, FSE'99*, Volume 1636 of *Lecture Notes in Computer Science*, pp. 156–170, Berlin, Heidelberg, New York, 1999. Springer-Verlag.

Errata

- C.2.7 The equation to calculate using only four tables, SP_1, SP_2, SP_3, and SP_4, has be corrected.
- Section D,E,F,G, and H have been added.