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 3

2 Notations and Conventions 3
   2.1 Radix 3
   2.2 Notations 3
   2.3 List of Symbols 3
   2.4 Bit/Byte Ordering 3

3 Structure of *Camellia* 5
   3.1 List of Functions and Variables 5
   3.2 Encryption Procedure 5
      3.2.1 128-bit key 5
      3.2.2 192-bit and 256-bit key 6
   3.3 Decryption Procedure 6
      3.3.1 128-bit key 6
      3.3.2 192-bit and 256-bit key 7
   3.4 Key Schedule 8

4 Components of *Camellia* 10
   4.1 F-function 10
   4.2 FL-function 10
   4.3 FL⁻¹-function 10
   4.4 S-function 10
   4.5 s-boxes 11
   4.6 P-function 15

A Figures of the *Camellia* Algorithm 16

B Test Data 22
C Software Implementation Techniques

C.1 Setup ............................................. 22
C.1.1 Store All Subkeys ................................ 22
C.1.2 Subkey Generation Order ............................ 23
C.1.3 XOR Cancellation Property in Key Schedule ............ 23
C.1.4 Rotation Bits for $K_L$, $K_R$, $K_A$, and $K_B$ ............ 23
C.1.5 $b_{d_2}$ and $b_{d_6}$ Generation from $k_{11}$ and $k_{12}$ ......... 23
C.1.6 On-the-fly Subkey Generation ........................ 23
C.1.7 128-bit key and 192/256-bit key ......................... 23
C.1.8 How to Rotate an Element in $Q$ ....................... 23
C.1.9 $F$-function ...................................... 24
C.1.10 Keyed Functions .................................. 24

C.2 Data Randomization .................................. 24
C.2.1 Endian Conversion .................................. 24
C.2.2 1-bit Rotation in Little Endian Interpretation .......... 25
C.2.3 Whitening ......................................... 25
C.2.4 Key XOR .......................................... 25
C.2.5 $S$-function ...................................... 26
C.2.6 $P$-function ...................................... 26
C.2.7 Substitution and Permutation ......................... 27
C.2.8 Making Indices for s-box ............................ 29

C.3 General Guidelines .................................. 30

D Design Policy ........................................... 30

E Design Rationale ........................................ 31
E.1 $F$-function ........................................ 31
E.2 $P$-function ........................................ 31
E.3 s-boxes ............................................. 32

F Version Information ........................................ 32

G Object Identifier ........................................ 33

H Applications and Products ................................. 33
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.

- **B** denotes a vector space of 8-bit (byte) elements; that is, $B := \text{GF}(2)^8$.
- **W** denotes a vector space of 32-bit (word) elements; that is, $W := B^4$.
- **L** denotes a vector space of 64-bit (double word) elements; that is, $L := B^8$.
- **Q** denotes a vector space of 128-bit (quad word) elements; that is, $Q := B^{16}$.
- 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.
- $\ll_n$ The left circular rotation of the operand by $n$ bits.
- $\&$ The bitwise AND operation.
- $\lor$ The bitwise OR operation.
- $\neg$ 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.
\[ Q_{(128)} = L_{(64)} || L_{(64)} \]
\[ = W_{(32)} || W_{(32)} || W_{(32)} || W_{(32)} \]
\[ = B_{(8)} || B_{(8)} || B_{(8)} || B_{(8)} \ldots \ldots \ldots \ldots \]
\[ = E_{(1)} || E_{(1)} || E_{(1)} || E_{(1)} \]
\[ \ldots \ldots \ldots \ldots \]
\[ = E_{125(1)} || E_{126(1)} || E_{127(1)} || E_{128(1)} \]

Numerical examples:

\[ Q_{(128)} = 0x123456789ABCDEF0011223456677_{(128)} \]
\[ L_{(64)} = Q_{L(64)} = 0x123456789ABCDEF_{(64)} \]
\[ L_{(64)} = Q_{R(64)} = 0x011223456677_{(64)} \]
\[ W_{(32)} = L_{L(32)} = 0x123456789_{(32)} \]
\[ W_{(32)} = L_{R(32)} = 0x89ABCDEF_{(32)} \]
\[ W_{(32)} = L_{2L(32)} = 0x0112233_{(32)} \]
\[ W_{(32)} = L_{2R(32)} = 0x44566778_{(32)} \]
\[ B_{(8)} = 0x01_{(8)}, B_{(8)} = 0x23_{(8)}, B_{(8)} = 0x45_{(8)}, B_{(8)} = 0x67_{(8)} \]
\[ B_{(8)} = 0x09_{(8)}, B_{(8)} = 0xAB_{(8)}, B_{(8)} = 0xCD_{(8)}, B_{(8)} = 0xEF_{(8)} \]
\[ B_{(8)} = 0x00_{(8)}, B_{(8)} = 0x11_{(8)}, B_{(8)} = 0x22_{(8)}, B_{(8)} = 0x33_{(8)} \]
\[ B_{(8)} = 0x44_{(8)}, B_{(8)} = 0x55_{(8)}, B_{(8)} = 0x66_{(8)}, B_{(8)} = 0x77_{(8)} \]
\[ B_{(8)} = 0x12_{(8)}, B_{(8)} = 0x13_{(8)}, B_{(8)} = 0x14_{(8)}, B_{(8)} = 0x15_{(8)} \]
\[ E_{(1)} = 0(1) \]
\[ E_{(1)} = 0(1) \]
\[ \ldots \]

\[ Q_{(128)} \ll 1 = E_{(1)} || E_{(1)} || E_{(1)} || E_{(1)} \]
\[ = 0x02468ACF13679DE002244688AACCEE_{(128)} \]
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.
- $kw_{(64)}, k_{(64)}, kl_{(64)}$: The subkeys.
- $Y_{(64)} = F(X_{(64)}, k_{(64)})$: 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 $kw_{(64)} (t = 1, 2, 3, 4)$, $k_{(64)} (u = 1, 2, \ldots, 18)$ and $kl_{(64)} (v = 1, 2, 3, 4)$ 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 $kw_{(64)}||kw_{(64)}$ and separated into $L_{0(64)}$ and $R_{0(64)}$ of equal length, i.e., $M_{(128)} \oplus (kw_{(64)}||kw_{(64)}) = L_{0(64)}||R_{0(64)}$. Then, the following operations are performed from $r = 1$ to 18, except for $r = 6$ and 12:

- $L_{r} = R_{r-1} \oplus F(L_{r-1}, k_{r})$
- $R_{r} = L_{r-1}$
For \( r = 6 \) and \( 12 \), the following is carried out:

\[
\begin{align*}
L'_r &= R_{r-1} \oplus F(L_{r-1}, k_r), \\
R'_r &= L_{r-1}, \\
L_r &= FL(L'_r, k_{2r}/6-1), \\
R_r &= FL^{-1}(R'_r, k_{2r}/6).
\end{align*}
\]

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 \( 18 \):

\[
\begin{align*}
L_r &= R_{r-1} \oplus F(L_{r-1}, k_r), \\
R_r &= L_{r-1}.
\end{align*}
\]

For \( r = 6, 12, \) and \( 18 \), perform the following:

\[
\begin{align*}
L'_r &= R_{r-1} \oplus F(L_{r-1}, k_r), \\
R'_r &= L_{r-1}, \\
L_r &= FL(L'_r, k_{2r}/6-1), \\
R_r &= FL^{-1}(R'_r, k_{2r}/6).
\end{align*}
\]

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 $kw_{t(64)}$ ($t = 1, 2, 3, 4$), $k_u(64)$ ($u = 1, 2, \ldots, 18$), 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 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 performed from $r = 18$ down to 1, except for $r = 13$ and 7:

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

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

$$
R_{r-1} = L_r \oplus F(R_r, k_r), \\
L_{r-1} = F(L_r, kl_{2(r-1)/6}), \\
R_{r-1} = FL(R_{r-1}, kl_{2(r-1)/6}), \\
L_{r-1} = FL^{-1}(L_{r-1}, kl_{2(r-1)/6-1}).
$$

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 $kw_{t(64)}$ ($t = 1, 2, 3, 4$), $k_u(64)$ ($u = 1, 2, \ldots, 24$), and $kl_v(64)$ ($v = 1, 2, \ldots, 6$) 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:

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

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

$$
R_{r-1} = L_r \oplus F(R_r, k_r), \\
L_{r-1} = R_r, \\
R_{r-1} = FL(R_{r-1}, kl_{2(r-1)/6}), \\
L_{r-1} = FL^{-1}(L_{r-1}, kl_{2(r-1)/6-1}).
$$

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:

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

\[
\begin{align*}
K_{L(128)} & = K_{LL(64)}||K_{LR(64)}, \\
K_{R(128)} & = K_{RL(64)}||K_{RR(64)}; & \text{for any size of key.}
\end{align*}
\]

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 $K_{B(128)}$. $\Sigma_i$ 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 $k_{L(64)}$, $k_{R(64)}$, and $k_{LL(64)}$ 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>
<thead>
<tr>
<th>$\Sigma_{i(64)}$</th>
<th>\text{Value}</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Sigma_{1(64)}$</td>
<td>0xA09E667F3BCC908B</td>
</tr>
<tr>
<td>$\Sigma_{2(64)}$</td>
<td>0xB67AE8584CAA73B2</td>
</tr>
<tr>
<td>$\Sigma_{3(64)}$</td>
<td>0xC6EF372FE94F82BE</td>
</tr>
<tr>
<td>$\Sigma_{4(64)}$</td>
<td>0x54FF53A5F1D36F1C</td>
</tr>
<tr>
<td>$\Sigma_{5(64)}$</td>
<td>0x10E527FADE682D1D</td>
</tr>
<tr>
<td>$\Sigma_{6(64)}$</td>
<td>0xB05688C2B3E6C1FD</td>
</tr>
</tbody>
</table>
### Table 2: Subkeys for 128-bit secret key

<table>
<thead>
<tr>
<th>Subkey</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prewhitening</td>
<td>$k_{w_1}^{(64)}$</td>
<td>$(K_L \ll 0)_{L(64)}$</td>
</tr>
<tr>
<td></td>
<td>$k_{w_2}^{(64)}$</td>
<td>$(K_L \ll 0)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round1)</td>
<td>$k_1^{(64)}$</td>
<td>$(K_A \ll 0)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round2)</td>
<td>$k_2^{(64)}$</td>
<td>$(K_A \ll 0)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round3)</td>
<td>$k_3^{(64)}$</td>
<td>$(K_L \ll 15)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round4)</td>
<td>$k_4^{(64)}$</td>
<td>$(K_L \ll 15)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round5)</td>
<td>$k_5^{(64)}$</td>
<td>$(K_A \ll 15)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round6)</td>
<td>$k_6^{(64)}$</td>
<td>$(K_A \ll 15)_{R(64)}$</td>
</tr>
<tr>
<td>$F \cdot L^{-1}$</td>
<td>$k_{l_1}^{(64)}$</td>
<td>$(K_A \ll 30)_{L(64)}$</td>
</tr>
<tr>
<td></td>
<td>$k_{l_2}^{(64)}$</td>
<td>$(K_A \ll 30)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round7)</td>
<td>$k_7^{(64)}$</td>
<td>$(K_L \ll 45)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round8)</td>
<td>$k_8^{(64)}$</td>
<td>$(K_L \ll 45)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round9)</td>
<td>$k_9^{(64)}$</td>
<td>$(K_A \ll 45)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round10)</td>
<td>$k_{10}^{(64)}$</td>
<td>$(K_A \ll 45)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round11)</td>
<td>$k_{11}^{(64)}$</td>
<td>$(K_A \ll 60)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round12)</td>
<td>$k_{12}^{(64)}$</td>
<td>$(K_A \ll 60)_{R(64)}$</td>
</tr>
<tr>
<td>$F \cdot L^{-1}$</td>
<td>$k_{l_3}^{(64)}$</td>
<td>$(K_L \ll 77)_{L(64)}$</td>
</tr>
<tr>
<td></td>
<td>$k_{l_4}^{(64)}$</td>
<td>$(K_L \ll 77)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round13)</td>
<td>$k_{13}^{(64)}$</td>
<td>$(K_A \ll 94)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round14)</td>
<td>$k_{14}^{(64)}$</td>
<td>$(K_A \ll 94)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round15)</td>
<td>$k_{15}^{(64)}$</td>
<td>$(K_A \ll 94)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round16)</td>
<td>$k_{16}^{(64)}$</td>
<td>$(K_A \ll 94)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round17)</td>
<td>$k_{17}^{(64)}$</td>
<td>$(K_L \ll 111)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round18)</td>
<td>$k_{18}^{(64)}$</td>
<td>$(K_L \ll 111)_{R(64)}$</td>
</tr>
<tr>
<td>Postwhitening</td>
<td>$k_{w_3}^{(64)}$</td>
<td>$(K_A \ll 111)_{L(64)}$</td>
</tr>
<tr>
<td></td>
<td>$k_{w_4}^{(64)}$</td>
<td>$(K_A \ll 111)_{R(64)}$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Subkey</th>
<th>Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prewhitening</td>
<td>$k_{w_1}^{(64)}$</td>
<td>$(K_L \ll 0)_{L(64)}$</td>
</tr>
<tr>
<td></td>
<td>$k_{w_2}^{(64)}$</td>
<td>$(K_L \ll 0)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round1)</td>
<td>$k_1^{(64)}$</td>
<td>$(K_B \ll 0)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round2)</td>
<td>$k_2^{(64)}$</td>
<td>$(K_B \ll 0)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round3)</td>
<td>$k_3^{(64)}$</td>
<td>$(K_R \ll 15)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round4)</td>
<td>$k_4^{(64)}$</td>
<td>$(K_R \ll 15)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round5)</td>
<td>$k_5^{(64)}$</td>
<td>$(K_A \ll 15)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round6)</td>
<td>$k_6^{(64)}$</td>
<td>$(K_A \ll 15)_{R(64)}$</td>
</tr>
<tr>
<td>$F \cdot L^{-1}$</td>
<td>$k_{l_1}^{(64)}$</td>
<td>$(K_R \ll 30)_{L(64)}$</td>
</tr>
<tr>
<td></td>
<td>$k_{l_2}^{(64)}$</td>
<td>$(K_R \ll 30)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round7)</td>
<td>$k_7^{(64)}$</td>
<td>$(K_B \ll 30)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round8)</td>
<td>$k_8^{(64)}$</td>
<td>$(K_B \ll 30)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round9)</td>
<td>$k_9^{(64)}$</td>
<td>$(K_A \ll 45)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round10)</td>
<td>$k_{10}^{(64)}$</td>
<td>$(K_A \ll 45)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round11)</td>
<td>$k_{11}^{(64)}$</td>
<td>$(K_A \ll 60)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round12)</td>
<td>$k_{12}^{(64)}$</td>
<td>$(K_A \ll 60)_{R(64)}$</td>
</tr>
<tr>
<td>$F \cdot L^{-1}$</td>
<td>$k_{l_3}^{(64)}$</td>
<td>$(K_A \ll 60)_{L(64)}$</td>
</tr>
<tr>
<td></td>
<td>$k_{l_4}^{(64)}$</td>
<td>$(K_A \ll 60)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round13)</td>
<td>$k_{13}^{(64)}$</td>
<td>$(K_R \ll 60)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round14)</td>
<td>$k_{14}^{(64)}$</td>
<td>$(K_R \ll 60)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round15)</td>
<td>$k_{15}^{(64)}$</td>
<td>$(K_B \ll 60)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round16)</td>
<td>$k_{16}^{(64)}$</td>
<td>$(K_B \ll 60)_{R(64)}$</td>
</tr>
<tr>
<td>$F$ (Round17)</td>
<td>$k_{17}^{(64)}$</td>
<td>$(K_L \ll 77)_{L(64)}$</td>
</tr>
<tr>
<td>$F$ (Round18)</td>
<td>$k_{18}^{(64)}$</td>
<td>$(K_L \ll 77)_{R(64)}$</td>
</tr>
<tr>
<td>Postwhitening</td>
<td>$k_{w_3}^{(64)}$</td>
<td>$(K_A \ll 111)_{L(64)}$</td>
</tr>
<tr>
<td></td>
<td>$k_{w_4}^{(64)}$</td>
<td>$(K_A \ll 111)_{R(64)}$</td>
</tr>
</tbody>
</table>
4 Components of Camellia

4.1 $F$-function

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

\[
F : L \times L \rightarrow L \\
(X_{(64)}, k_{(64)}) \mapsto Y_{(64)} = P(S(X_{(64)} \oplus k_{(64)})).
\]

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:

\[
FL : L \times L \rightarrow L \\
(X_{(32)}||X_{R(32)}, k_{L(32)}||k_{R(32)}) \mapsto Y_{L(32)}||Y_{R(32)},
\]

where

\[
Y_{R(32)} = ((X_{L(32)} \cap k_{L(32)}) \ll_1) \oplus X_{R(32)},
\]

\[
Y_{L(32)} = (Y_{R(32)} \cup k_{R(32)}) \oplus X_{L(32)}.
\]

4.3 $FL^{-1}$-function

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

\[
FL^{-1} : L \times L \rightarrow L \\
(Y_{L(32)}||Y_{R(32)}, k_{L(32)}||k_{R(32)}) \mapsto X_{L(32)}||X_{R(32)},
\]

where

\[
X_{L(32)} = (Y_{R(32)} \cup k_{R(32)}) \oplus X_{L(32)},
\]

\[
X_{R(32)} = ((X_{L(32)} \cap k_{L(32)}) \ll_1) \oplus Y_{R(32)}.
\]

4.4 $S$-function

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

\[
S : L \rightarrow L \\
t_{1(s)}||t_{2(s)}||t_{3(s)}||t_{4(s)}||t_{5(s)}||t_{6(s)}||t_{7(s)}||t_{8(s)} \mapsto t'_{1(s)}||t'_{2(s)}||t'_{3(s)}||t'_{4(s)}||t'_{5(s)}||t'_{6(s)}||t'_{7(s)}||t'_{8(s)}
\]
\[ l_1'(8) = s_1(l_1(8)), \]
\[ l_2'(8) = s_2(l_2(8)), \]
\[ l_3'(8) = s_3(l_3(8)), \]
\[ l_4'(8) = s_4(l_4(8)), \]
\[ l_5'(8) = s_2(l_5(8)), \]
\[ l_6'(8) = s_3(l_6(8)), \]
\[ l_7'(8) = s_4(l_7(8)), \]
\[ l_8'(8) = s_1(l_8(8)), \]

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

\[
\begin{align*}
s_1 & : \text{B} \rightarrow \text{B} \\
x_{(8)} & \mapsto h(g(f(0xc5 \oplus x_{(8)}))) \oplus 0x6e,
\end{align*}
\]

\[
\begin{align*}
s_2 & : \text{B} \rightarrow \text{B} \\
x_{(8)} & \mapsto s_1(x_{(8)}) \lll 1,
\end{align*}
\]

\[
\begin{align*}
s_3 & : \text{B} \rightarrow \text{B} \\
x_{(8)} & \mapsto s_1(x_{(8)}) \ggg 1,
\end{align*}
\]

\[
\begin{align*}
s_4 & : \text{B} \rightarrow \text{B} \\
x_{(8)} & \mapsto s_1(x_{(8)} \lll 1),
\end{align*}
\]

where the functions \( f, g, h \) are given as follows:

\[
f : \text{B} \rightarrow \text{B} \]
\[
a_{1(1)} || a_{2(1)} || a_{3(1)} || a_{4(1)} || a_{5(1)} || a_{6(1)} || a_{7(1)} || a_{8(1)}
\]
\[
\mapsto b_{1(1)} || b_{2(1)} || b_{3(1)} || b_{4(1)} || b_{5(1)} || b_{6(1)} || b_{7(1)} || b_{8(1)},
\]

where

\[
\begin{align*}
b_1 &= a_6 \oplus a_2, \\
b_2 &= a_7 \oplus a_1, \\
b_3 &= a_8 \oplus a_5 \oplus a_3, \\
b_4 &= a_8 \oplus a_3, \\
b_5 &= a_7 \oplus a_4, \\
b_6 &= a_5 \oplus a_2, \\
b_7 &= a_8 \oplus a_1, \\
b_8 &= a_6 \oplus a_4.
\end{align*}
\]
\[ g : B \rightarrow B \]
\[ a_{1(1)}||a_{2(1)}||a_{3(1)}||a_{4(1)}||a_{5(1)}||a_{6(1)}||a_{7(1)}||a_{8(1)} \]
\[ \rightarrow b_{1(1)}||b_{2(1)}||b_{3(1)}||b_{4(1)}||b_{5(1)}||b_{6(1)}||b_{7(1)}||b_{8(1)} , \]

where

\[
(b_8 + b_\gamma \alpha + b_6 \alpha^2 + b_5 \alpha^3) + (b_4 + b_3 \alpha + b_2 \alpha^2 + b_1 \alpha^3) \beta
\]
\[= 1/(a_8 + a_7 \alpha + a_6 \alpha^2 + a_5 \alpha^3) + (a_4 + a_3 \alpha + a_2 \alpha^2 + a_1 \alpha^3) \beta.\]

This inversion is performed in \(\text{GF}(2^8)\) assuming \(\frac{1}{\beta} = 0\), where \(\beta\) is an element in \(\text{GF}(2^8)\) that satisfies \(\beta^8 + \beta^6 + \beta^5 + \beta^3 + 1 = 0\), and \(\alpha = \beta^{2^{28}} = \beta^6 + \beta^5 + \beta^3 + \beta^2\) is an element in \(\text{GF}(2^4)\) that satisfies \(\alpha^4 + \alpha + 1 = 0\).

\[ h : B \rightarrow B \]
\[ a_{1(1)}||a_{2(1)}||a_{3(1)}||a_{4(1)}||a_{5(1)}||a_{6(1)}||a_{7(1)}||a_{8(1)} \]
\[ \rightarrow b_{1(1)}||b_{2(1)}||b_{3(1)}||b_{4(1)}||b_{5(1)}||b_{6(1)}||b_{7(1)}||b_{8(1)} , \]

where

\[
\begin{align*}
b_1 &= a_5 \oplus a_6 \oplus a_2 , \\
b_2 &= a_6 \oplus a_2 , \\
b_3 &= a_7 \oplus a_4 , \\
b_4 &= a_8 \oplus a_2 , \\
b_5 &= a_7 \oplus a_3 , \\
b_6 &= a_8 \oplus a_1 , \\
b_7 &= a_5 \oplus a_1 , \\
b_8 &= a_6 \oplus a_3.
\end{align*}
\]
Table 4: The s-box $s_1$

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

<table>
<thead>
<tr>
<th>112</th>
<th>130</th>
<th>44</th>
<th>236</th>
<th>179</th>
<th>39</th>
<th>192</th>
<th>229</th>
<th>228</th>
<th>133</th>
<th>87</th>
<th>53</th>
<th>234</th>
<th>12</th>
<th>174</th>
<th>65</th>
</tr>
</thead>
<tbody>
<tr>
<td>35</td>
<td>239</td>
<td>107</td>
<td>147</td>
<td>69</td>
<td>25</td>
<td>166</td>
<td>33</td>
<td>237</td>
<td>14</td>
<td>79</td>
<td>78</td>
<td>29</td>
<td>101</td>
<td>146</td>
<td>189</td>
</tr>
<tr>
<td>134</td>
<td>184</td>
<td>176</td>
<td>143</td>
<td>124</td>
<td>236</td>
<td>31</td>
<td>206</td>
<td>62</td>
<td>48</td>
<td>220</td>
<td>95</td>
<td>94</td>
<td>197</td>
<td>11</td>
<td>26</td>
</tr>
<tr>
<td>166</td>
<td>225</td>
<td>57</td>
<td>202</td>
<td>213</td>
<td>71</td>
<td>93</td>
<td>61</td>
<td>217</td>
<td>1</td>
<td>90</td>
<td>214</td>
<td>81</td>
<td>86</td>
<td>108</td>
<td>77</td>
</tr>
<tr>
<td>159</td>
<td>154</td>
<td>102</td>
<td>251</td>
<td>204</td>
<td>176</td>
<td>45</td>
<td>116</td>
<td>18</td>
<td>43</td>
<td>32</td>
<td>240</td>
<td>177</td>
<td>132</td>
<td>153</td>
<td></td>
</tr>
<tr>
<td>223</td>
<td>76</td>
<td>203</td>
<td>194</td>
<td>52</td>
<td>126</td>
<td>118</td>
<td>5</td>
<td>109</td>
<td>183</td>
<td>169</td>
<td>49</td>
<td>209</td>
<td>23</td>
<td>4</td>
<td>215</td>
</tr>
<tr>
<td>20</td>
<td>88</td>
<td>58</td>
<td>97</td>
<td>222</td>
<td>27</td>
<td>17</td>
<td>28</td>
<td>50</td>
<td>15</td>
<td>166</td>
<td>22</td>
<td>83</td>
<td>24</td>
<td>242</td>
<td>34</td>
</tr>
<tr>
<td>264</td>
<td>68</td>
<td>207</td>
<td>178</td>
<td>195</td>
<td>181</td>
<td>122</td>
<td>145</td>
<td>36</td>
<td>8</td>
<td>232</td>
<td>168</td>
<td>96</td>
<td>252</td>
<td>105</td>
<td>80</td>
</tr>
<tr>
<td>170</td>
<td>208</td>
<td>160</td>
<td>125</td>
<td>161</td>
<td>137</td>
<td>98</td>
<td>151</td>
<td>84</td>
<td>91</td>
<td>30</td>
<td>149</td>
<td>224</td>
<td>255</td>
<td>100</td>
<td>210</td>
</tr>
<tr>
<td>16</td>
<td>196</td>
<td>0</td>
<td>72</td>
<td>163</td>
<td>247</td>
<td>117</td>
<td>219</td>
<td>138</td>
<td>3</td>
<td>230</td>
<td>218</td>
<td>9</td>
<td>63</td>
<td>221</td>
<td>148</td>
</tr>
<tr>
<td>135</td>
<td>92</td>
<td>131</td>
<td>2</td>
<td>205</td>
<td>74</td>
<td>144</td>
<td>51</td>
<td>115</td>
<td>103</td>
<td>246</td>
<td>243</td>
<td>157</td>
<td>127</td>
<td>191</td>
<td>226</td>
</tr>
<tr>
<td>82</td>
<td>165</td>
<td>216</td>
<td>38</td>
<td>200</td>
<td>55</td>
<td>198</td>
<td>59</td>
<td>129</td>
<td>160</td>
<td>111</td>
<td>75</td>
<td>19</td>
<td>190</td>
<td>99</td>
<td>46</td>
</tr>
<tr>
<td>233</td>
<td>121</td>
<td>167</td>
<td>140</td>
<td>159</td>
<td>110</td>
<td>188</td>
<td>142</td>
<td>41</td>
<td>245</td>
<td>249</td>
<td>182</td>
<td>47</td>
<td>253</td>
<td>180</td>
<td>89</td>
</tr>
<tr>
<td>120</td>
<td>162</td>
<td>6</td>
<td>106</td>
<td>231</td>
<td>70</td>
<td>113</td>
<td>186</td>
<td>212</td>
<td>37</td>
<td>171</td>
<td>66</td>
<td>136</td>
<td>162</td>
<td>141</td>
<td>250</td>
</tr>
<tr>
<td>114</td>
<td>7</td>
<td>186</td>
<td>86</td>
<td>248</td>
<td>238</td>
<td>172</td>
<td>10</td>
<td>54</td>
<td>73</td>
<td>42</td>
<td>104</td>
<td>60</td>
<td>56</td>
<td>241</td>
<td>164</td>
</tr>
<tr>
<td>64</td>
<td>40</td>
<td>211</td>
<td>123</td>
<td>187</td>
<td>201</td>
<td>67</td>
<td>193</td>
<td>21</td>
<td>227</td>
<td>173</td>
<td>244</td>
<td>119</td>
<td>199</td>
<td>128</td>
<td>158</td>
</tr>
</tbody>
</table>

Table 5: The s-box $s_2$

<table>
<thead>
<tr>
<th>224</th>
<th>5</th>
<th>88</th>
<th>217</th>
<th>103</th>
<th>78</th>
<th>129</th>
<th>203</th>
<th>201</th>
<th>11</th>
<th>174</th>
<th>106</th>
<th>213</th>
<th>24</th>
<th>93</th>
<th>130</th>
</tr>
</thead>
<tbody>
<tr>
<td>70</td>
<td>223</td>
<td>214</td>
<td>39</td>
<td>138</td>
<td>50</td>
<td>76</td>
<td>66</td>
<td>219</td>
<td>28</td>
<td>158</td>
<td>156</td>
<td>68</td>
<td>202</td>
<td>37</td>
<td>123</td>
</tr>
<tr>
<td>13</td>
<td>113</td>
<td>96</td>
<td>31</td>
<td>248</td>
<td>215</td>
<td>62</td>
<td>157</td>
<td>124</td>
<td>96</td>
<td>185</td>
<td>190</td>
<td>188</td>
<td>139</td>
<td>22</td>
<td>52</td>
</tr>
<tr>
<td>77</td>
<td>195</td>
<td>114</td>
<td>149</td>
<td>171</td>
<td>142</td>
<td>186</td>
<td>122</td>
<td>179</td>
<td>2</td>
<td>180</td>
<td>173</td>
<td>162</td>
<td>172</td>
<td>216</td>
<td>154</td>
</tr>
<tr>
<td>23</td>
<td>26</td>
<td>53</td>
<td>204</td>
<td>247</td>
<td>153</td>
<td>97</td>
<td>90</td>
<td>232</td>
<td>36</td>
<td>86</td>
<td>64</td>
<td>225</td>
<td>99</td>
<td>9</td>
<td>51</td>
</tr>
<tr>
<td>191</td>
<td>152</td>
<td>151</td>
<td>133</td>
<td>104</td>
<td>252</td>
<td>236</td>
<td>10</td>
<td>218</td>
<td>111</td>
<td>83</td>
<td>98</td>
<td>163</td>
<td>46</td>
<td>8</td>
<td>175</td>
</tr>
<tr>
<td>40</td>
<td>176</td>
<td>116</td>
<td>194</td>
<td>189</td>
<td>94</td>
<td>84</td>
<td>66</td>
<td>100</td>
<td>30</td>
<td>57</td>
<td>44</td>
<td>166</td>
<td>48</td>
<td>229</td>
<td>68</td>
</tr>
<tr>
<td>253</td>
<td>136</td>
<td>159</td>
<td>101</td>
<td>135</td>
<td>107</td>
<td>244</td>
<td>35</td>
<td>72</td>
<td>16</td>
<td>209</td>
<td>81</td>
<td>192</td>
<td>249</td>
<td>210</td>
<td>160</td>
</tr>
<tr>
<td>85</td>
<td>161</td>
<td>65</td>
<td>260</td>
<td>67</td>
<td>19</td>
<td>196</td>
<td>47</td>
<td>168</td>
<td>182</td>
<td>60</td>
<td>43</td>
<td>193</td>
<td>255</td>
<td>200</td>
<td>165</td>
</tr>
<tr>
<td>32</td>
<td>137</td>
<td>0</td>
<td>144</td>
<td>71</td>
<td>239</td>
<td>234</td>
<td>183</td>
<td>21</td>
<td>6</td>
<td>205</td>
<td>181</td>
<td>18</td>
<td>126</td>
<td>187</td>
<td>41</td>
</tr>
<tr>
<td>15</td>
<td>194</td>
<td>7</td>
<td>4</td>
<td>155</td>
<td>148</td>
<td>33</td>
<td>102</td>
<td>230</td>
<td>206</td>
<td>237</td>
<td>231</td>
<td>59</td>
<td>254</td>
<td>127</td>
<td>197</td>
</tr>
<tr>
<td>164</td>
<td>55</td>
<td>177</td>
<td>76</td>
<td>145</td>
<td>110</td>
<td>141</td>
<td>118</td>
<td>3</td>
<td>45</td>
<td>222</td>
<td>150</td>
<td>38</td>
<td>125</td>
<td>198</td>
<td>92</td>
</tr>
<tr>
<td>211</td>
<td>242</td>
<td>79</td>
<td>25</td>
<td>63</td>
<td>220</td>
<td>121</td>
<td>29</td>
<td>82</td>
<td>235</td>
<td>243</td>
<td>109</td>
<td>94</td>
<td>251</td>
<td>105</td>
<td>178</td>
</tr>
<tr>
<td>240</td>
<td>49</td>
<td>12</td>
<td>212</td>
<td>207</td>
<td>140</td>
<td>226</td>
<td>117</td>
<td>169</td>
<td>74</td>
<td>87</td>
<td>132</td>
<td>17</td>
<td>69</td>
<td>27</td>
<td>246</td>
</tr>
<tr>
<td>228</td>
<td>14</td>
<td>115</td>
<td>170</td>
<td>241</td>
<td>221</td>
<td>89</td>
<td>20</td>
<td>108</td>
<td>146</td>
<td>94</td>
<td>208</td>
<td>120</td>
<td>112</td>
<td>227</td>
<td>73</td>
</tr>
<tr>
<td>128</td>
<td>80</td>
<td>167</td>
<td>246</td>
<td>119</td>
<td>147</td>
<td>134</td>
<td>131</td>
<td>42</td>
<td>199</td>
<td>91</td>
<td>233</td>
<td>238</td>
<td>143</td>
<td>1</td>
<td>61</td>
</tr>
</tbody>
</table>
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 | 64 | 166 |
| 197 | 134 | 77 | 51 | 263 | 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 | 60 | 105 |
| 8 | 98 | 0 | 36 | 209 | 261 | 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 | 166 | 137 | 95 | 177 | 23 |
| 244 | 188 | 211 | 70 | 207 | 55 | 94 | 71 | 148 | 250 | 252 | 91 | 151 | 264 | 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 | 166 | 237 | 79 | 29 | 146 |
| 134 | 175 | 124 | 31 | 62 | 220 | 94 | 11 | 166 | 57 | 213 | 93 | 217 | 90 | 81 | 108 |
| 139 | 164 | 261 | 176 | 116 | 43 | 240 | 132 | 223 | 203 | 52 | 118 | 109 | 169 | 209 | 4 |
| 20 | 68 | 222 | 17 | 50 | 166 | 83 | 242 | 264 | 207 | 196 | 122 | 36 | 232 | 96 | 105 |
| 170 | 160 | 161 | 98 | 84 | 30 | 224 | 100 | 16 | 0 | 163 | 117 | 138 | 230 | 9 | 221 |
| 135 | 131 | 206 | 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 | 66 | 239 | 147 | 26 | 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 \textit{P}-function

The \textit{P}-function is a part of \textit{F}-function, which is defined as follows:

\[
P : L \rightarrow L
\]

\[
z_1(8) \parallel z_2(8) \parallel z_3(8) \parallel z_4(8) \parallel z_5(8) \parallel z_6(8) \parallel z_7(8) \parallel z_8(8) \rightarrow z'_1(8) \parallel z'_2(8) \parallel z'_3(8) \parallel z'_4(8) \parallel z'_5(8) \parallel z'_6(8) \parallel z'_7(8) \parallel z'_8(8),
\]

where

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

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

\[
\begin{pmatrix}
z_8 \\
z_7 \\
\vdots \\
z_1
\end{pmatrix}
\rightarrow
\begin{pmatrix}
z'_8 \\
z'_7 \\
\vdots \\
z'_1
\end{pmatrix}
= P
\begin{pmatrix}
z_8 \\
z_7 \\
\vdots \\
z_1
\end{pmatrix},
\]

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.

\[
\begin{align*}
\text{(left half)} & = F(K_{LR} \oplus \text{(right half)}, \Sigma_2) \\
\text{(right half)} & = F(K_{LL}, \Sigma_1)
\end{align*}
\]

for 128-bit keys

\[
\begin{align*}
\text{(left half)} & = K_{RL} \oplus F(K_{LR} \oplus \text{(right half)}, \Sigma_2) \\
\text{(right half)} & = K_{RR} \oplus F(K_{LL} \oplus K_{RL}, \Sigma_1)
\end{align*}
\]

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 $(kl_5, kl_6) \equiv (k_{11}, k_{12}) \ll 32$. 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.

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

The technique has high parallelism.

Minimum operations without rotation.

$$x \leftarrow (x \ll_{16}) + (x \gg_{16})$$

$$x \leftarrow ((x \cap 0xff00ff) \ll 8) + ((x \gg 8) \cap 0xff00ff)$$
Using rotations.
\[
x \leftarrow (x \cap 0xff00ff) \gg 8 \right) + ((x \ll_8) \cap 0xff00ff)
\]

Using SSE. New Intel Pentium family processors including Pentium III has very effective instruction for reordering data, which is called pshufw [199]. 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 \(F^{-1}\)-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.

\[
x \leftarrow ((2x) \cap 0xfee0ef) + ((x \gg 15) \cap 0xfee0ef)
\]

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 \(\ll_1\), \(\ll_1\) 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, \(0xfee0ef\).

C.2.3 Whitening

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

\[
\begin{align*}
(x \oplus k) \oplus y & = (x \oplus y) \oplus k, \\
(x \oplus k) \oplus l & = x \oplus (k \oplus l), \\
(x \oplus k) \cap l & = (x \cap l) \oplus (k \cap l), \\
(x \oplus k) \ll_1 & = (x \ll_1) \oplus (k \ll_1), \\
(x \oplus k) \cup l & = (x \cup l) \oplus (k \cap l),
\end{align*}
\]

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 $\text{GF}(2^8)$. However, do not compute $\text{GF}(2^8)$ 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.

$$
Z_L \leftarrow Z_L \oplus (Z_R \ll 8) \\
Z_R \leftarrow Z_R \oplus (Z_L \ll 16) \\
Z_L \leftarrow Z_L \oplus (Z_R \ll 8) \\
Z_R \leftarrow Z_R \oplus (Z_L \ll 8) \\
Z'_L \leftarrow Z_R \\
Z'_R \leftarrow Z_L
$$

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

$$
Z_R \leftarrow Z_R \ll 8 \\
Z_L \leftarrow Z_L \oplus Z_R \\
Z_R \leftarrow Z_R \ll 8 \\
Z_L \leftarrow Z_L \gg 8 \\
Z_R \leftarrow Z_R \oplus Z_L \\
Z_L \leftarrow Z_L \ll 8 \\
Z_R \leftarrow Z_R \oplus Z_L \\
Z'_L \leftarrow Z_R \\
Z'_R \leftarrow Z_L
$$

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.

$$
z'_8 \leftarrow z_1 \oplus z_4 \oplus z_5 \oplus z_6 \oplus z_7$$
Next, compute the following equation:

\[ z'_4 \leftarrow z'_0 \oplus z_1 \oplus z_2 \oplus z_3 \]
\[ z'_7 \leftarrow z'_4 \oplus z_2 \oplus z_7 \oplus z_8 \]
\[ z'_3 \leftarrow z'_7 \oplus z_1 \oplus z_2 \oplus z_4 \]
\[ z'_6 \leftarrow z'_3 \oplus z_1 \oplus z_6 \oplus z_7 \]
\[ z'_2 \leftarrow z'_6 \oplus z_1 \oplus z_3 \oplus z_4 \]
\[ z'_5 \leftarrow z'_2 \oplus z_1 \oplus z_5 \oplus z_6 \]
\[ z'_1 \leftarrow z'_5 \oplus z_2 \oplus z_3 \oplus z_4 \]

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

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

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).

\[
\begin{align*}
SP_1(y_1) &= (s_1(y_1), s_1(y_1), s_1(y_1), 0, s_1(y_1), 0, 0, s_1(y_1)) \\
SP_2(y_2) &= (0, s_2(y_2), s_2(y_2), s_2(y_2), s_2(y_2), 0, 0, 0) \\
SP_3(y_3) &= (s_3(y_3), 0, s_3(y_3), s_3(y_3), 0, s_3(y_3), s_3(y_3), 0) \\
SP_4(y_4) &= (s_4(y_4), s_4(y_4), 0, s_4(y_4), 0, 0, s_4(y_4), s_4(y_4)) \\
SP_5(y_5) &= (0, s_2(y_5), s_2(y_5), s_2(y_5), 0, s_2(y_5), s_2(y_5), s_2(y_5)) \\
SP_6(y_6) &= (s_3(y_6), 0, s_3(y_6), s_3(y_6), s_3(y_6), 0, s_3(y_6), s_3(y_6)) \\
SP_7(y_7) &= (s_4(y_7), s_4(y_7), 0, s_4(y_7), s_4(y_7), s_4(y_7), 0, s_4(y_7)) \\
SP_8(y_8) &= (s_1(y_8), s_1(y_8), s_1(y_8), 0, s_1(y_8), s_1(y_8), s_1(y_8), 0)
\end{align*}
\]  

(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.
Table definitions:

<table>
<thead>
<tr>
<th># of Table Lookups</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td># of XORs</td>
<td>7</td>
</tr>
<tr>
<td>Size of Table (KB)</td>
<td>16</td>
</tr>
</tbody>
</table>

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

\[
SP_a(y) = (s_1(y), s_1(y), s_1(y), s_1(y), s_1(y), s_1(y))
\]
\[
SP_b(y) = (s_2(y), s_2(y), s_2(y), s_2(y), s_2(y), s_2(y))
\]
\[
SP_c(y) = (s_3(y), s_3(y), s_3(y), s_3(y), s_3(y), s_3(y))
\]
\[
SP_d(y) = (s_4(y), s_4(y), s_4(y), s_4(y), s_4(y), s_4(y))
\]

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

<table>
<thead>
<tr>
<th># of Table Lookups</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td># of XORs</td>
<td>7</td>
</tr>
<tr>
<td># of ANDs</td>
<td>8</td>
</tr>
<tr>
<td>Size of Table (KB)</td>
<td>8</td>
</tr>
</tbody>
</table>

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 SP₁, SP₂, SP₃, and SP₄ defined in equation (3). Then, compute the following equation:

\[
(z'_1, z'_2, z'_3, z'_4, z'_5, z'_6, z'_7, z'_8)
\leftarrow SP_1(y_1) \oplus SP_2(y_2) \oplus SP_3(y_3) \oplus SP_4(y_4) \oplus \nu(SP_1(y_8) \oplus SP_2(y_5) \oplus SP_3(y_6) \oplus SP_4(y_7)),
\]

where \(\nu\) denotes the operation that copies the first 4 bytes to the last 4 bytes. This technique requires the following operations.

<table>
<thead>
<tr>
<th># of Table Lookups</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td># of XORs</td>
<td>7</td>
</tr>
<tr>
<td># of (\nu)s</td>
<td>1</td>
</tr>
<tr>
<td>Size of Table (KB)</td>
<td>8</td>
</tr>
</tbody>
</table>

**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):

\[
SP_{1110}(y) = (s_1(y), s_1(y), s_1(y), 0)
\]
\[
SP_{0222}(y) = (0, s_2(y), s_2(y), s_2(y))
\]
\[
SP_{3033}(y) = (s_3(y), 0, s_3(y), s_3(y))
\]
\[
SP_{4401}(y) = (s_4(y), s_4(y), 0, s_4(y))
\]
Then, compute as follows:

\[
D \leftarrow SP_{1110}(y_8) \oplus SP_{0001}(y_6) \oplus SP_{4401}(y_7)
\]

\[
U \leftarrow SP_{1110}(y_1) \oplus SP_{0001}(y_2) \oplus SP_{4401}(y_4)
\]

\[
(z'_1, z'_2, z'_3, z'_4) \leftarrow D \oplus U
\]

\[
(z''_5, z''_6, z''_7, z''_8) \leftarrow (z'_1, z'_2, z'_3, z'_4) \oplus (U \gg 8)
\]

This technique requires the following operations.

<table>
<thead>
<tr>
<th># of Table Lookups</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td># of XORs</td>
<td>8</td>
</tr>
<tr>
<td># of Rotations</td>
<td>1</td>
</tr>
<tr>
<td>Size of Table (KB)</td>
<td>4</td>
</tr>
</tbody>
</table>

[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):

\[
SP_{1001}(y) = (s_1(y), 0, 0, s_1(y))
\]

\[
SP_{2200}(y) = (s_2(y), s_2(y), 0, 0)
\]

\[
SP_{0001}(y) = (0, s_3(y), s_3(y), 0)
\]

\[
SP_{0044}(y) = (0, 0, s_4(y), s_4(y))
\]

Then, compute as follows:

\[
D \leftarrow SP_{1110}(y_8) \oplus SP_{0001}(y_6) \oplus SP_{4401}(y_7)
\]

\[
(z'_1, z'_2, z'_3, z'_4) \leftarrow D \oplus SP_{1110}(y_1) \oplus SP_{0001}(y_2) \oplus SP_{4401}(y_4)
\]

\[
(z''_5, z''_6, z''_7, z''_8) \leftarrow D \oplus SP_{1001}(y_1) \oplus SP_{2200}(y_2) \oplus SP_{0330}(y_3) \oplus SP_{0044}(y_4)
\]

This technique requires the following operations.

<table>
<thead>
<tr>
<th># of Table Lookups</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td># of XORs</td>
<td>11</td>
</tr>
<tr>
<td>Size of Table (KB)</td>
<td>8</td>
</tr>
</tbody>
</table>

### 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 [199], `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(\_rotl)" and use "\_rotl", 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 affect the running time.

A few processors have an instruction to get the time stamp. For example, IA-32 (after Pentium) has rdtsc [199] 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 GF($2^8$). Moreover, we reduced the inversion function in GF($2^8$) to a few 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 bitwise 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 $\text{GF}(2^8)$ for enhanced security and small hardware design.

It is well known that the smallest of the maximum differential probability of functions in $\text{GF}(2^8)$ was proven to be $2^{-6}$, and the smallest of the maximum linear probability of functions in $\text{GF}(2^8)$ is conjectured to be $2^{-6}$. There is a function affine equivalent to the inversion function in $\text{GF}(2^8)$ 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 $\text{GF}(2^8)$ 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,

- International Workshop SAC 2000

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 key 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


Errata

- C.2.7 The equation to calculate using only four tables, $SP_1$, $SP_2$, $SP_3$, and $SP_4$, has been corrected.

- Section D,E,F,G, and H have been added.