FS-TR01-02 July. 30, 2001 (6 pages)

Technical Report

A Note on Implementing CRYPTON
- Endian-Neutral Implementation -

Chae Hoon Lim
Cryptography & Network Security Center, Future Systems, Inc.
1009-1, Daechi-Dong, Kangnam-Gu, Seoul, 135-851, Korea
Tel: +82-2-3468-7740, Fax: +82-2-3468-7800, E-mail: chlim@future.co.kr

Abstract

This note provides some guidelines for implementing the block cipher CRYPTON in big endian and little endian machines.

Cryptography & Network Security Center, Future Systems, Inc.
(http://www.future.co.kr, http://www.securitytechnet.com)

A Note on Implementing CRYPTON
- Endian-Neutral Implementation -

Chae Hoon Lim

Cryptography & Network Security Center, Future Systems, Inc.
1009-1, Daechi-Dong, Kangnam-Gu, Seoul, 135-851, Korea
Tel: +82-2-3468-7740, Fax: +82-2-3468-7800, E-mail: chlim@future.co.kr

July 21, 1999 : 1st draft, unpublished
July 30, 2001 : revised & published online

1 Introduction

The block cipher CRYPTON [1] is designed to encrypt and decrypt blocks of data consisting of 128 bits under the control of a 256-bit key. A 128-bit data block[1], internally represented in 4×4 byte array, is subject to an initial key xoring, then to 12 times round transformations and finally to an output transformation. Each round consists of byte-wise substitution γ, column-wise bit permutation π, column-to-row transposition τ, and key xoring σ. Even and odd rounds are slightly different in γ and π. The final output transformation consists of $\pi \circ \tau \circ \pi$. The decryption process is identical to the encryption process, except that different round keys are applied. See [1] for details on algorithm specification and security/performance analysis. Figure 1 shows the high level structure of CRYPTON.

2 Byte Ordering Conventions: Big Endian vs Little Endian

There are two different conventions for byte ordering within a word. In *Big Endian* addressing, the address of a data is the address of the most significant byte; in *Little Endian*, the address of a data is the address of the least significant byte. For example, suppose that a 4-byte word is read from the memory address '0x2340'. Big Endian machines put the byte addressed by 0x2340 at the most significant position in a 4-byte word, while Little Endian machines put the byte at the least significant position (see Figure 2).

Different byte orderings may cause a problem when exchanging data among different machines. Thus, for interoperability, it is necessary to specify the chosen endianness, if relevant, in any algorithm requiring word-wise operations.[2] CRYPTON can be implemented only using byte-wise logical operations (AND, XOR and ROT) and table lookups (no arithmetic operations used). This allows us to make CRYPTON endian-neutral; we can implement CRYPTON without endian change in either Big Endian machines or Little Endian machines. Though the CRYPTON specification [1] describes the algorithm using the little endian convention, we can derive a functionally-equivalent specification using the big endian convention. Note that these two implementation methods only make a difference in software implementation on large microprocessors (e.g., 32- or 64-bit μPs). In the case of hardware implementation, either convention may be taken at the designer's convenience.

[1] A 128-bit block is regarded as byte strings numbered from left to right, i.e., the left most byte (the first byte) is byte 0 and the right most byte (the last byte) is byte 15. Bits within a byte may be numbered as convenient.

[2] Of course, 4-byte words may be retrieved from / stored into memory by 4 times byte accesses. Then we don't have to care about endianness, but it is costly.

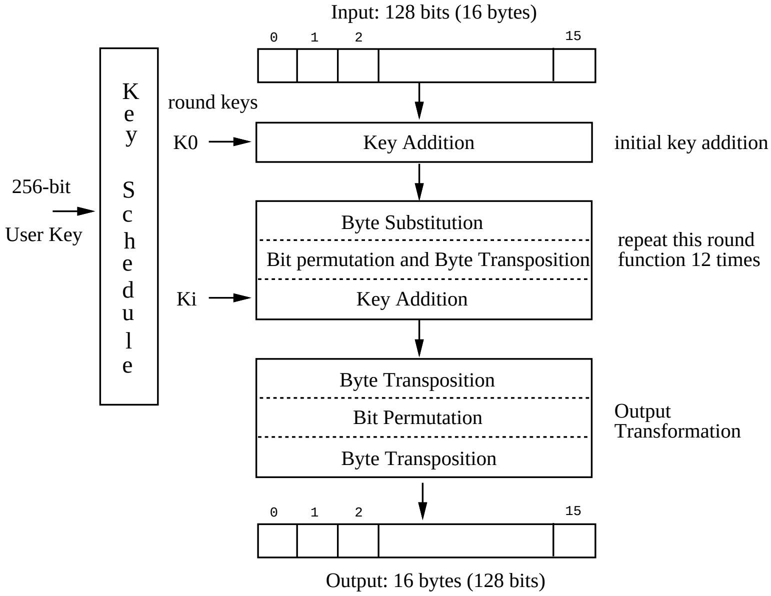

Figure 1: The high-level structure of CRYPTON

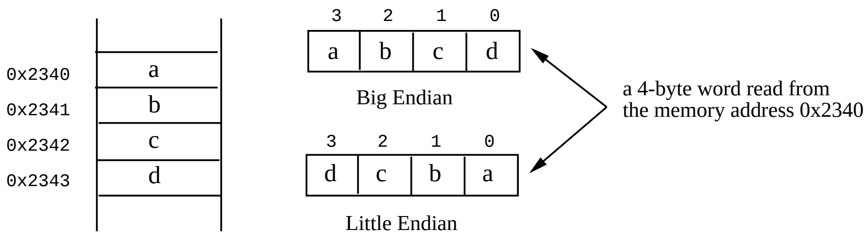

Figure 2: 4-byte word access in Big Endian and Little Endian addressing

3 Implementing CRYPTON in Little Endian Machines

Since the original specification describes CRYPTON according to the little endian convention, we first review an implementation based on the little endian convention and takes it as a reference for a big endian implementation.

In Little Endian machines, a 16-byte input message is mapped into internal data as in Figure 3.[3] The first 4-byte word is byte-ordered according to the little endian convention and placed in the first row of the 4×4 matrix and the next 4-byte word in the second row, and so on. Note that this is equivalent to converting a given 16-byte string into a 4-word array of unsigned integers by type-casting each 4-byte word as an unsigned integer in a 32-bit little endian machine.

The input plaintext represented in 4×4 matrix is now subject to encryption as specified in the CRYPTON spec. [1]. The internal round transformation in this little endian convention is depicted in Figure 4. After completion of encryption, the ciphertext internally represented in 4×4 matrix should be transformed back into a 16-byte output message by the reverse process of input transformation. In this little endian specification, the right-top byte position serves as the origin of 4×4 byte matrix and as a starting point of byte order.

[3]The k-th byte a_k in this document is exactly equal to the (i, j)-th byte a_{ij} in the CRYPTON spec. [1]; just interpret k as two digits in the mod-4 number system, i.e., $k = 4i + j$.

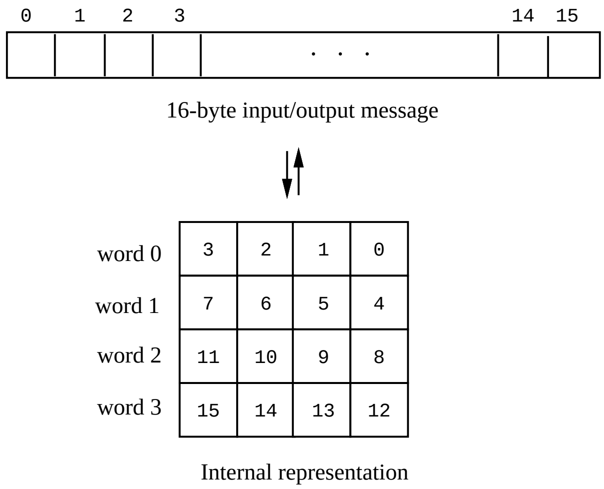

Figure 3: Internal data representation in the little endian convention

The CRYPTON key schedule accepts a 32-byte user key. These key bytes are mapped into two 4×4 byte matrices as shown in Figure 5, one with even-numbered bytes and the other with odd-numbered bytes (denoted by U and V respectively in [1]). These two user-key matrices are subject to key expansion (consisting of round transformations with all-zero key and mixing steps) to produce two matrices of expanded keys. Then, round keys are generated by successively updating the expanded key matrices (see [1] for details). Note that the operations relevant to byte ordering in the key schedule include round transformations, left rotations of a 4-byte word and operations involving round constants.

4 Implementing CRYPTON in Big Endian Machines

In the big endian convention, the byte order, when reading a 4-byte word from memory as an unsigned integer, is exactly reverse to the little endian case. Byte reordering cosumes non-negligible cycles in software implementations. So it would be nice for an algorithm to allow restructuring of internal processing logic to avoid such overhead. This is possible in CRYPTON since it essentially processes the input message in byte-by-byte fashion. For this, however, we have to rearrange the internal processing components so that the final output is the same as before.

In this environment, a 16-byte input message is mapped into internal 4×4 byte matrix as shown in Figure 6. The only difference from the little endian convention is byte order within a 4-byte word. Note that in this case the left-top byte position is serving as the origin of the 4×4 matrix and as a starting point of byte order. After completetion of internal encryption/decryption processing, the final 4×4 matrix should be transformed back into a 16-byte ciphertext message using the reverse process of the input message transformation.

To obtain the same encryption/decryption result as in the little endian convention, we have to adjust basic operations, γ, π and τ, in such a way that each byte undergoes the same operations as in the little endian convention (note that key xoring is independent of byte ordering if each round key is also byte-reversed). Figure 7 shows the equivalent round transformations corresponding to the big endian byte order.

The internal key scheduling process must also be subject to change according to the big endian byte order. Note that the key schedule also makes use of components of the round transformation. Therefore, to guarantee the same internal processing on each byte as in the little endian convention, we have to rearrange a user key and produce round keys in the big endian byte order. Figure 8 shows the mapping

S_3	S_2	S_1	S_0
S_0	S_3	S_2	S_1
S_1	S_0	S_3	S_2
S_2	S_1	S_0	S_3

γ_o

S_1	S_0	S_3	S_2
S_2	S_1	S_0	S_3
S_3	S_2	S_1	S_0
S_0	S_3	S_2	S_1

γ_e

$\Downarrow$

π_3	π_2	π_1	π_0

π_o

π_1	π_0	π_3	π_2

π_e

$\Downarrow$

a_3	a_2	a_1	a_0
a_7	a_6	a_5	a_4
a_{11}	a_{10}	a_9	a_8
a_{15}	a_{14}	a_{13}	a_{12}

$\xRightarrow{\tau}$

a_{12}	a_8	a_4	a_0
a_{13}	a_9	a_5	a_1
a_{14}	a_{10}	a_6	a_2
a_{15}	a_{11}	a_7	a_3

Figure 4: Round transformations (key xoring omitted) in Little Endian machines

of a 32-byte user key into its internal representation for key scheduling in big endian byte order. The mapped user key is now subject to key expansion and round key generation as specified in [1], with the following changes:[4]

- All component functions (γ, π and τ) involved in key scheduling should be replaced with the big endian equivalents depicted in Figure 7.
- Left rotate by n bits should be replaced with right rotate by n bits (or equivalently left rotate by $32 - n$ bits).
- All encryption and decryption round constants should be byte-reversed, e.g., $C_e[0]$ = `0xa54ff53a` must be changed into $C_e[0]$ = `0x3af54fa5`.

References

[1] C.H.Lim, Specification and analysis of CRYPTON Version 1.0, May 18, 2000.

[4] Of course, the basic operations, γ, π and τ, should be replaced with the big endian equivalents defined in Figure 7.

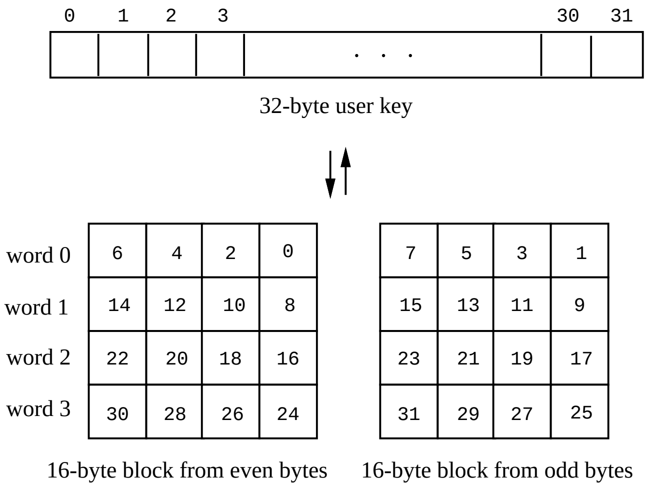

Figure 5: Internal key representation in the little endian convention

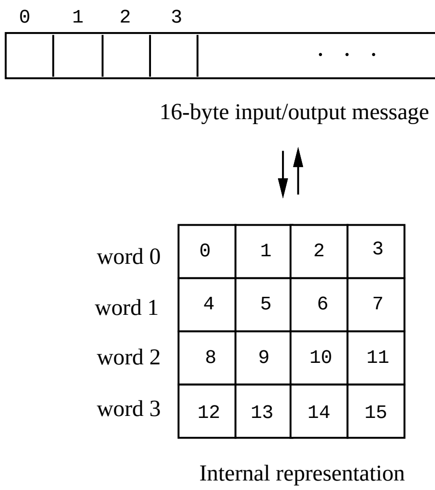

Figure 6: Internal data representation in Big Endian convention

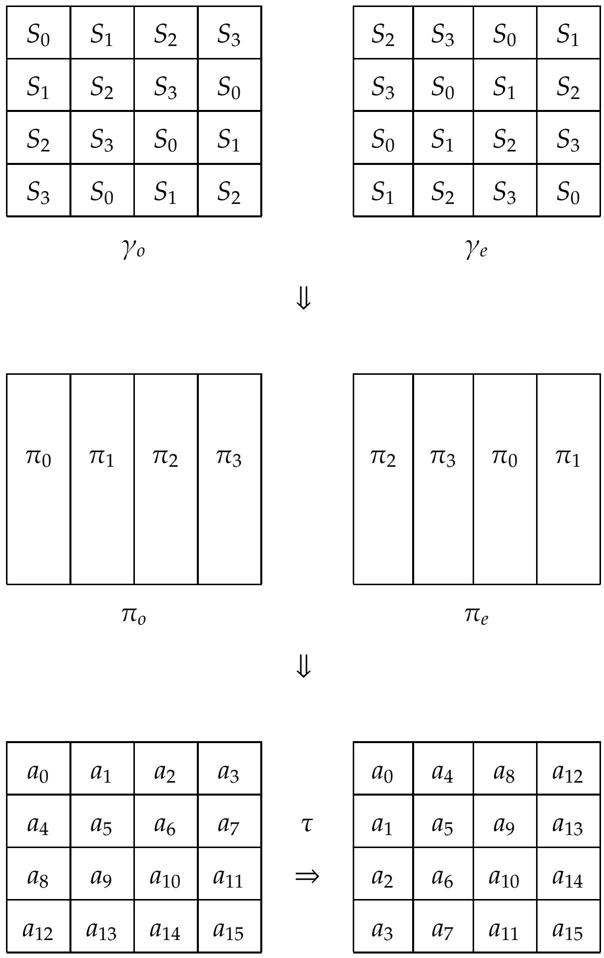

Figure 7: Round transformations (key xoring omitted) in Big Endian machines

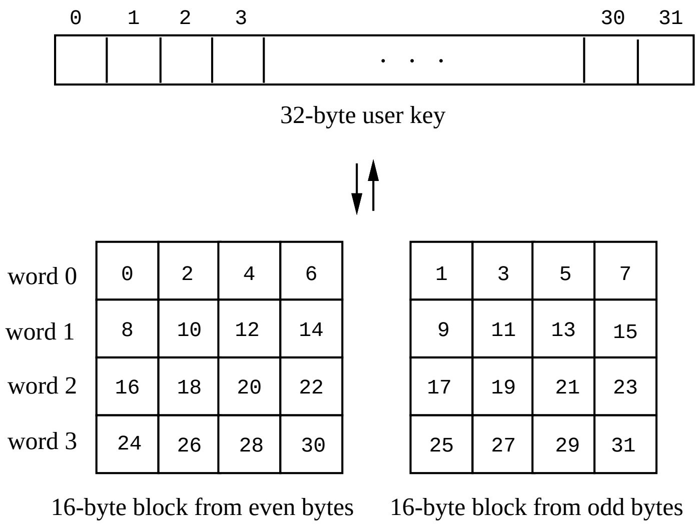

Figure 8: Internal key representation in the big endian convention