CRYPTON : A New 128-bit Block Cipher

– Specification and Analysis –

Chae Hoon Lim

Information and Communications Research Center
Future Systems, Inc.
372-2, Yang Jae-Dong, Seo Cho-Gu, Seoul, 137-130, Korea
Tel: +82-2-578-0581, Fax: +82-2-578-0584
chlim@future.co.kr, http://crypt.future.co.kr/ chlim

Abstract

A new 128-bit block cipher called CRYPTON is proposed as a candidate algorithm for the Advanced Encryption Standard (AES). The cipher encrypts/decrypts a 128-bit data block with a variable-length key up to 256 bits (with increment by a multiple of 32 bits) by iterating a fully parallelizable round function 12 times. The decryption process can be made identical to the encryption process by applying different round keys generated by different key schedules. CRYPTON is designed to make full use of increasing software/hardware parallelisms in today's microprocessor designs. It uses only very simple operations, such as logical ANDs, ORs, XORs, Shifts and table lookups. Furthermore, the two 8×8 S-boxes were carefully chosen by taking into account hardware efficiency. As a result, CRYPTON can achieve very high performances in various platforms, such as software implementations on 8-bit, 16-bit, 32-bit and 64-bit microprocessors and a dedicated hardware implementation. Our C language implementation could achieve the speed of about 6.4 Mbytes/sec on a 200 MHz Pentium Pro PC, running Windows 95. Key scheduling is also very simple and takes less time than encrypting one 128-bit data block. Based on our preliminary analysis we conjecture that there is no known attack on the 12-round CRYPTON faster than exhaustive key search. For example, differential and linear cryptanalysis can be shown to require more ciphertexts than available. This paper present various aspects on the design, analysis and implementation of CRYPTON.

Contents

List of Figures

List of Tables

1 Introduction

Most block ciphers have been designed based on the Feistel structure, e.g., DES, LOKI, Blowfish, CAST, etc. In these Feistel-type ciphers, the plaintext is divided into two equal sub-blocks, and in each round one sub-block is transformed by some nonlinear function and then exclusive-ored with the other sub-block. Such round function is repeated sufficiently many times to achieve adequate security.

On the other hand, some other block ciphers were designed using round functions allowing parallel nonlinear processing on the whole data block. For example, the ciphers IDEA, SAFER, 3-Way, SHARK and SQUARE belong to this category. The advantage of this approach is that the resulting cipher is highly parallelizable and easy to analyze the security against differential and linear cryptanalysis. Parallelizability is of great importance for maximizing speed, since most modern processors are supporting more and more parallelisms in software and/or hardware.

The block cipher CRYPTON is designed based on the latter approach. In fact, its design is much influenced by SQUARE. CRYPTON processes each data block by representing it into a 4×4 byte array as in SQUARE. The round transformation of CRYPTON consists of four parallelizable steps: byte-wise substitutions, column-wise bit permutation, column-to-row transposition, and then key addition. The encryption process invloves 12 repetitions of (essentially) the same round transformation. The decryption process can be made the same as the encryption process, except that different subkeys are applied in each round. Figure 1 shows the high level structure of CRYPTON.

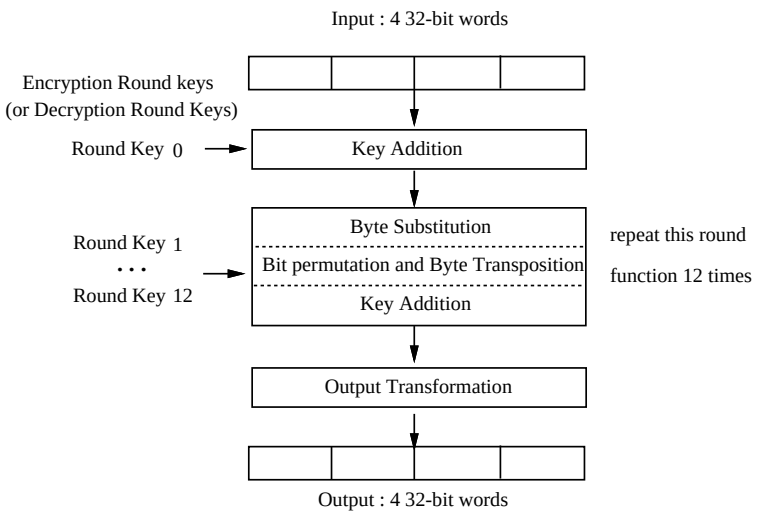

Figure 1: The structure of CRYPTON

The block cipher CRYPTON has the following features:

- 12-round self-reciprocal cipher with block length of 128 bits.
- Key lengths supported: $64 + 32k$ $(0 \leq k \leq 6)$ bits (may allow any number of key length up to 256 bits).
- Identical process for encryption and decryption (with different subkeys).
- Strong security against existing attacks: e.g, differential and linear cryptanalysis require more ciphertexts than available.
- High parallelism for fast implementation in both software and hardware.
- Tradeoffs between speed and memory: Standard software implementation of CRYPTON requires 512 bytes of storage for two 8×8 substitution boxes (S-boxes for short) and thus well suited to the environment with limited computing resources, such as smart cards and other portable devices. Using 4 Kbytes of memory, the speed can be substantially increased. In the case of VLSI

implementation, the S-boxes can be efficiently implemented using a relatively small number of nand gates.

- Ease of implementation in various platforms: easy to implement on 8-bit, 16-bit or 32-bit processors, also very efficient for hardware implementation.

We believe that CRYPTON meets all the requirements for the Advanced Encryption Standard (AES).

In this paper we present a complete specification of the proposed cipher CRYPTON and our preliminary analysis for its security and efficiency.

2 Algorithm Specification

2.1 Symbols and Notation

Throughout this paper we will use the following symbols and notation:

- We use the term 'word' to denote a 32-bit number (note however that DWORD is used in C source code). A number is usually represented in hexadecimal (with prefix '0x').
- We follow the little endian convention for byte ordering in char string $\leftrightarrow$ word conversion. That is, the first character is always placed in the least significant position.
- We write $A = (A[3], A[2], A[1], A[0])^t$ when the data variable A represent a 4×4 byte array, where $A[i]$ $(0 \leq i \leq 3)$ is a 4-byte word represented by $A[i] = a_{i3} \parallel a_{i2} \parallel a_{i1} \parallel a_{i0}$. Here $\parallel$ denote concatenation of two bit strings and the superscript t in a vector or array denotes transposition.
- $ROL(x, n)$ denotes left-rotation of integer x by n-bit positions. We also use the notation $x^{\ll n}$ interchangeably, whenever convenient.
- $f \circ g$ denotes composition of functions f and g, i.e., $(f \circ g)(x) = f(g(x))$.
- $\wedge, \vee, \oplus$: bit-wise logical operations for AND, OR and XOR (exclusive-or), respectively.
- $\overline{x}$ denotes the bit-wise complement of x.

2.2 Basic Building Blocks

2.2.1 Nonlinear Substitution γ

CRYPTON uses two 8×8 S-boxes, S_0 and S_1, for a nonlinear transformation. The two S-boxes were constructed from three 4×4 S-boxes using a 3-round Feistel structure so that $S_0(S_1(x)) = S_1(S_0(x)) = x$ for any 8-bit number x. The construction and property of these S-boxes will be described in more detail in Sect.3.1.

The S-box transformation γ consists of byte-wise substitutions on a 4×4 byte array (see Figure 2). Two different transformations are used alternatively in successive rounds: γ_o in odd rounds and γ_e in even rounds.

- S-box transformation γ_o for odd rounds (i.e., rounds 1, 3, etc.): $B = \gamma_o(A)$ defined by

$$B[0] \leftarrow S_1(a_{03}) \parallel S_0(a_{02}) \parallel S_1(a_{01}) \parallel S_0(a_{00}),$$
$$B[1] \leftarrow S_0(a_{13}) \parallel S_1(a_{12}) \parallel S_0(a_{11}) \parallel S_1(a_{10}),$$
$$B[2] \leftarrow S_1(a_{23}) \parallel S_0(a_{22}) \parallel S_1(a_{21}) \parallel S_0(a_{20}),$$
$$B[3] \leftarrow S_0(a_{33}) \parallel S_1(a_{32}) \parallel S_0(a_{31}) \parallel S_1(a_{30}).$$

- S-box transformation γ_e for even rounds (i.e., rounds 2, 4, etc.): $B = \gamma_e(A)$ defined by

$$B[0] \leftarrow S_0(a_{03}) \parallel S_1(a_{02}) \parallel S_0(a_{01}) \parallel S_1(a_{00}),$$
$$B[1] \leftarrow S_1(a_{13}) \parallel S_0(a_{12}) \parallel S_1(a_{11}) \parallel S_0(a_{10}),$$
$$B[2] \leftarrow S_0(a_{23}) \parallel S_1(a_{22}) \parallel S_0(a_{21}) \parallel S_1(a_{20}),$$
$$B[3] \leftarrow S_1(a_{33}) \parallel S_0(a_{32}) \parallel S_1(a_{31}) \parallel S_0(a_{30}).$$

Note that two S-boxes are arranged so that $\gamma_o(\gamma_e(A)) = \gamma_o(\gamma_e(A)) = A$ for any 4×4 byte array A. This property will be used to derive the identical process for encryption and decryption.

$A[0]$	a_{03}	a_{02}	a_{01}	a_{00}		$B[0]$	$S_1(a_{03})$	$S_0(a_{02})$	$S_1(a_{01})$	$S_0(a_{00})$
$A[1]$	a_{13}	a_{12}	a_{11}	a_{10}	γ_o	$B[1]$	$S_0(a_{13})$	$S_1(a_{12})$	$S_0(a_{11})$	$S_1(a_{10})$
$A[2]$	a_{23}	a_{22}	a_{21}	a_{20}	$\Longrightarrow$	$B[2]$	$S_1(a_{23})$	$S_0(a_{22})$	$S_1(a_{21})$	$S_0(a_{20})$
$A[3]$	a_{33}	a_{32}	a_{31}	a_{30}		$B[3]$	$S_0(a_{33})$	$S_1(a_{32})$	$S_0(a_{31})$	$S_1(a_{30})$

$A[0]$	a_{03}	a_{02}	a_{01}	a_{00}		$B[0]$	$S_0(a_{03})$	$S_1(a_{02})$	$S_0(a_{01})$	$S_1(a_{00})$
$A[1]$	a_{13}	a_{12}	a_{11}	a_{10}	γ_e	$B[1]$	$S_1(a_{13})$	$S_0(a_{12})$	$S_1(a_{11})$	$S_0(a_{10})$
$A[2]$	a_{23}	a_{22}	a_{21}	a_{20}	$\Longrightarrow$	$B[2]$	$S_0(a_{23})$	$S_1(a_{22})$	$S_0(a_{21})$	$S_1(a_{20})$
$A[3]$	a_{33}	a_{32}	a_{31}	a_{30}		$B[3]$	$S_1(a_{33})$	$S_0(a_{32})$	$S_1(a_{31})$	$S_0(a_{30})$

Figure 2: The S-box transformation γ

2.2.2 Linear Transformations π and τ

As linear transformations, CRYPTON uses a bit-wise permutation and a byte-wise transposition in sequence. The bit permutation π mixes four bytes in each byte column of a 4×4 byte array and the byte transposition τ transposes the resulting byte columns into byte rows.

For bit-extraction we define the four masking words (M_3, M_2, M_1, M_0) as

$$\begin{aligned} M_0 &= m_3 \| m_2 \| m_1 \| m_0 = \texttt{0x3fcff3fc}, \\ M_1 &= m_0 \| m_3 \| m_2 \| m_1 = \texttt{0xfc3fcff3}, \\ M_2 &= m_1 \| m_0 \| m_3 \| m_2 = \texttt{0xf3fc3fcf}, \\ M_3 &= m_2 \| m_1 \| m_0 \| m_3 = \texttt{0xcff3fc3f}, \end{aligned}$$

where $m_0 = \texttt{0xfc}, m_1 = \texttt{0xf3}, m_2 = \texttt{0xcf}, m_3 = \texttt{0x3f}$. We will use two versions of π to make the encryption and decryption processes identical: π_o is used in odd rounds and π_e is used in even rounds. They are defined as follows:

- Bit permutation π_o for odd rounds: $B = \pi_o(A)$ defined by

$$\begin{aligned} B[0] &\leftarrow (A[0] \wedge M_0) \oplus (A[1] \wedge M_1) \oplus (A[2] \wedge M_2) \oplus (A[3] \wedge M_3), \\ B[1] &\leftarrow (A[0] \wedge M_1) \oplus (A[1] \wedge M_2) \oplus (A[2] \wedge M_3) \oplus (A[3] \wedge M_0), \\ B[2] &\leftarrow (A[0] \wedge M_2) \oplus (A[1] \wedge M_3) \oplus (A[2] \wedge M_0) \oplus (A[3] \wedge M_1), \\ B[3] &\leftarrow (A[0] \wedge M_3) \oplus (A[1] \wedge M_0) \oplus (A[2] \wedge M_1) \oplus (A[3] \wedge M_2). \end{aligned}$$

- Bit permutation π_e for even rounds: $B = \pi_e(A)$ defined by

$$\begin{aligned} B[0] &\leftarrow (A[0] \wedge M_1) \oplus (A[1] \wedge M_2) \oplus (A[2] \wedge M_3) \oplus (A[3] \wedge M_0), \\ B[1] &\leftarrow (A[0] \wedge M_2) \oplus (A[1] \wedge M_3) \oplus (A[2] \wedge M_0) \oplus (A[3] \wedge M_1), \\ B[2] &\leftarrow (A[0] \wedge M_3) \oplus (A[1] \wedge M_0) \oplus (A[2] \wedge M_1) \oplus (A[3] \wedge M_2), \\ B[3] &\leftarrow (A[0] \wedge M_0) \oplus (A[1] \wedge M_1) \oplus (A[2] \wedge M_2) \oplus (A[3] \wedge M_3). \end{aligned}$$

Note that π_o and π_e can be implemented by the same function as follows:

$$\pi_e((A[3], A[2], A[1], A[0])^t) = \pi_o((A[2], A[1], A[0], A[3])^t).$$

The bit permutation π may be viewed as consisting of two steps: Each byte column is rearranged in such a way that each byte in the same byte column contributes two bits to each new byte and then the mod-2 sum of the four input bytes is exclusive-ored with each new byte to form the final output byte.

For example, the expression for π_o can be rewritten as follows (see also Figure 3 for graphical view of π_o, where addition of T is omitted.):

$$\begin{aligned}
T &= A[0] \oplus A[1] \oplus A[2] \oplus A[3], \\
B[0] &\leftarrow (A[0] \wedge MI_0) \oplus (A[1] \wedge MI_1) \oplus (A[2] \wedge MI_2) \oplus (A[3] \wedge MI_3) \oplus T, \\
B[1] &\leftarrow (A[0] \wedge MI_1) \oplus (A[1] \wedge MI_2) \oplus (A[2] \wedge MI_3) \oplus (A[3] \wedge MI_0) \oplus T, \\
B[2] &\leftarrow (A[0] \wedge MI_2) \oplus (A[1] \wedge MI_3) \oplus (A[2] \wedge MI_0) \oplus (A[3] \wedge MI_1) \oplus T, \\
B[3] &\leftarrow (A[0] \wedge MI_3) \oplus (A[1] \wedge MI_0) \oplus (A[2] \wedge MI_1) \oplus (A[3] \wedge MI_2) \oplus T,
\end{aligned}$$

where MI_i is the bit-wise complement of M_i, i.e.,

$$\begin{aligned}
MI_0 &= \texttt{0xc0300c03} = 11000000\ 00110000\ 00001100\ 00000011_{(2)}, \\
MI_1 &= \texttt{0x03c0300c} = 00000011\ 11000000\ 00110000\ 00001100_{(2)}, \\
MI_2 &= \texttt{0x0c03c030} = 00001100\ 00000011\ 11000000\ 00110000_{(2)}, \\
MI_3 &= \texttt{0x300c03c0} = 00110000\ 00001100\ 00000011\ 11000000_{(2)}.
\end{aligned}$$

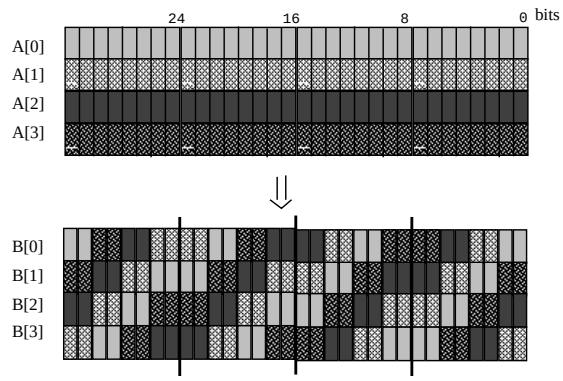

Figure 3: The bit permutation π_o (adding exclusive-or sum omitted)

As shown above, the bit permutation π_o (π_e, resp.) consists of column-wise permutations. Let A^i be the i-th byte column of A, i.e., $A^i = (a_{3i}, a_{2i}, a_{1i}, a_{0i})^t$. And Let π_i be the bit permutation of the i-th column induced by π. Then we can rewrite π_o and π_e as

$$\begin{aligned}
\pi_o(A) &= (\pi_3(A^3), \pi_2(A^2), \pi_1(A^1), \pi_0(A^0))^t, \\
\pi_e(A) &= (\pi_0(A^3), \pi_3(A^2), \pi_2(A^1), \pi_1(A^0))^t.
\end{aligned}$$

The component column-permutation π_i can be easily derived from the description of π. For example, $B^0 = \pi_0(A^0)$ is given by

$$\begin{aligned}
b_{00} &\leftarrow (a_{00} \wedge m_0) \oplus (a_{10} \wedge m_1) \oplus (a_{20} \wedge m_2) \oplus (a_{30} \wedge m_3), \\
b_{10} &\leftarrow (a_{00} \wedge m_1) \oplus (a_{10} \wedge m_2) \oplus (a_{20} \wedge m_3) \oplus (a_{30} \wedge m_0), \\
b_{20} &\leftarrow (a_{00} \wedge m_2) \oplus (a_{10} \wedge m_3) \oplus (a_{20} \wedge m_0) \oplus (a_{30} \wedge m_1), \\
b_{30} &\leftarrow (a_{00} \wedge m_3) \oplus (a_{10} \wedge m_0) \oplus (a_{20} \wedge m_1) \oplus (a_{30} \wedge m_2).
\end{aligned}$$

It is also easy to see that if $\pi_0([d, c, b, a]^t) = [h, g, f, e]^t$, then

$$\begin{aligned}
\pi_1([d, c, b, a]^t) &= [e, h, g, f]^t, \\
\pi_2([d, c, b, a]^t) &= [f, e, h, g]^t, \\
\pi_3([d, c, b, a]^t) &= [g, f, e, h]^t.
\end{aligned}$$

The byte transposition τ simply rearranges a 4×4 byte array by moving the byte at the (i, j)-th position to the (j, i)-th position.

- Byte transposition τ: $B = \tau(A)$, where $b_{ij} = a_{ji}$. That is,

  $B[0] \leftarrow a_{30} \parallel a_{20} \parallel a_{10} \parallel a_{00}$,

  $B[1] \leftarrow a_{31} \parallel a_{21} \parallel a_{11} \parallel a_{01}$,

  $B[2] \leftarrow a_{32} \parallel a_{22} \parallel a_{12} \parallel a_{02}$,

  $B[3] \leftarrow a_{33} \parallel a_{23} \parallel a_{13} \parallel a_{03}$.

$A[0]$	a_{03}	a_{02}	a_{01}	a_{00}		$B[0]$	a_{30}	a_{20}	a_{10}	a_{00}
$A[1]$	a_{13}	a_{12}	a_{11}	a_{10}	τ	$B[1]$	a_{31}	a_{21}	a_{11}	a_{01}
$A[2]$	a_{23}	a_{22}	a_{21}	a_{20}	$\Longrightarrow$	$B[2]$	a_{32}	a_{22}	a_{12}	a_{02}
$A[3]$	a_{33}	a_{32}	a_{31}	a_{30}		$B[3]$	a_{33}	a_{23}	a_{13}	a_{33}

Figure 4: The byte transposition τ

2.2.3 Key Addition σ

Key mixing is performed by simply exclusive-oring round keys with data words as follows:

- Key addition σ_K : $B = \sigma_K(A)$ is defined by

  $B[0] \leftarrow A[0] \oplus K[0]$,

  $B[1] \leftarrow A[1] \oplus K[1]$,

  $B[2] \leftarrow A[2] \oplus K[2]$,

  $B[3] \leftarrow A[3] \oplus K[3]$.

2.2.4 Round Transformation ρ

One round of CRYPTON consists of applying in sequence S-box transformation, bit permutation, byte transposition and key addition. That is, the encryption round functions used for odd and even rounds (the round number is denoted by r) are defined by

$$\begin{aligned} \rho_{oK}(A) &= (\sigma_K \circ \tau \circ \pi_o \circ \gamma_o)(A) \quad \text{for } r = 1, 3, \cdots \text{ etc.,} \\ \rho_{eK}(A) &= (\sigma_K \circ \tau \circ \pi_e \circ \gamma_e)(A) \quad \text{for } r = 2, 4, \cdots \text{ etc.,} \end{aligned}$$

where $K = (K[3], K[2], K[1], K[0])^t$ is a 4-word round key and $A = (A[3], A[2], A[1], A[0])^t$ is a 4-word input data (both key and data are regarded as 4×4 byte arrays).

Notice the inverse relations of each component functions. All component functions except for γ_o and γ_e are involutions. That is, $\gamma_o^{-1} = \gamma_e$ ($\gamma_e^{-1} = \gamma_o$), $\pi_o^{-1} = \pi_o$ ($\pi_e^{-1} = \pi_e$), $\tau^{-1} = \tau$ and $\sigma_K^{-1} = \sigma_K$. Therefore, the decryption round transformation is given by

$$\begin{aligned} \rho_{oK}^{-1}(A) &= (\gamma_e \circ \pi_o \circ \tau \circ \sigma_K)(A), \\ \rho_{eK}^{-1}(A) &= (\gamma_o \circ \pi_e \circ \tau \circ \sigma_K)(A). \end{aligned}$$

With this decryption round we can decrypt the ciphertext by applying encryption round keys in reverse order. However, it would be better to be able to decrypt ciphertexts by using the same encryption process. This is possible in CRYPTON, as can be seen below. Of course, for this we have to generate decryption round keys using a different key schedule.

2.3 Encryption and Decryption

Let K_e^i be the i-th encryption round key consisting of 4 words, derived from a user-supplied key K using the encryption key schedule described later. The encryption transformation E_K of r-round CRYPTON

under key K consists of an initial key addition and $r/2$ times repetitions of ρ_o and ρ_e and then a final output transformation (we assume r is even). E_K can be described as

$$E_K = \phi_e \circ \rho_{eK_e^r} \circ \rho_{oK_e^{r-1}} \circ \cdots \circ \rho_{eK_e^2} \circ \rho_{oK_e^1} \circ \sigma_{K_e^0}, \tag{1}$$

where ϕ_e is the output transformation to make the encryption and decryption processes identical and is given by

$$\phi_e = \tau \circ \pi_e \circ \tau.$$

Similarly we define ϕ_o as $\phi_o = \tau \circ \pi_o \circ \tau$. The corresponding decryption transformation D_K can be shown to have the same form as E_K, except for using suitably transformed round keys:

$$D_K = \phi_e \circ \rho_{eK_d^r} \circ \rho_{oK_d^{r-1}} \circ \cdots \circ \rho_{eK_d^2} \circ \rho_{oK_d^1} \circ \sigma_{K_d^0}, \tag{2}$$

where the decryption round keys are defined by

$$K_d^{r-i} = \begin{cases} \phi_e(K_e^i) & \text{for } i = 0, 2, 4, \cdots, \\ \phi_o(K_e^i) & \text{for } i = 1, 3, 5, \cdots. \end{cases} \tag{3}$$

This shows that encryption and decryption can be performed by the same code (logic) if different round keys are applied.

Notice that

$$\begin{aligned} \phi_e \circ \sigma_{K_e^i} &= \sigma_{\phi_e(K_e^i)} \circ \phi_e = \sigma_{K_d^{r-i}} \circ \phi_e \quad \text{for } i = 0, 2, 4, \cdots, \\ \phi_o \circ \sigma_{K_e^i} &= \sigma_{\phi_o(K_e^i)} \circ \phi_0 = \sigma_{K_d^{r-i}} \circ \phi_o \quad \text{for } i = 1, 3, 5, \cdots. \end{aligned}$$

Using this property, we can incorporate the output transformation ϕ_e into the final round as follows:

$$\begin{aligned} \phi_e \circ \rho_{eK_e^r} &= \phi_e \circ \sigma_{K_e^r} \circ \tau \circ \pi_e \circ \gamma_e \\ &= \sigma_{K_d^0} \circ \phi_e \circ \tau \circ \pi_e \circ \gamma_e \\ &= \sigma_{K_d^0} \circ \tau \circ \gamma_e. \end{aligned}$$

Also note that $\tau \circ \gamma = \gamma \circ \tau$.

We next explain how the decryption process is derived. For simplicity, we consider a 2-round version of CRYPTON ($r = 2$). The encryption process of this two round version can be rewritten as:

$$\begin{aligned} E_K &= \phi_e \circ \rho_{eK_e^2} \circ \rho_{oK_e^1} \circ \sigma_{K_e^0} \\ &= \phi_e \circ (\phi_e \circ \sigma_{K_d^0} \circ \phi_e \circ \tau \circ \pi_e \circ \gamma_e) \circ (\phi_o \circ \sigma_{K_d^1} \circ \phi_o \circ \tau \circ \pi_o \circ \gamma_o) \circ \sigma_{K_e^0} \\ &= (\sigma_{K_d^0} \circ \tau \circ \gamma_e) \circ (\phi_o \circ \sigma_{K_d^1} \circ \tau \circ \gamma_o) \circ \sigma_{K_e^0} \\ &= \sigma_{K_d^0} \circ (\gamma_e \circ \pi_o \circ \tau \circ \sigma_{K_d^1}) \circ (\gamma_o \circ \tau \circ \sigma_{K_e^0}) \\ &= \sigma_{K_d^0} \circ (\sigma_{K_d^1} \circ \tau \circ \pi_o \circ \gamma_o)^{-1} \circ (\sigma_{K_e^0} \circ \tau \circ \gamma_e)^{-1} \\ &= (\sigma_{K_d^0})^{-1} \circ (\rho_{oK_d^1})^{-1} \circ (\phi_e \circ \rho_{eK_d^2})^{-1}. \end{aligned}$$

Therefore, the decryption process of the reduced version is given by

$$D_K = E_K^{-1} = \phi_e \circ \rho_{eK_d^2} \circ \rho_{oK_d^1} \circ \sigma_{K_d^0}.$$

Thus we have shown that the decryption process can be the same as the encryption process, except that ϕ-transformed round keys are applied in reverse order.

2.4 Key Scheduling

The purpose of key scheduling is to securely generate from a given short secret key as many subkeys as needed during the encryption/decryption process. R-round CRYPTON requires total $4 \times (r + 1)$ round keys of 32 bits. These round keys are generated from a user key of $64 + 32k (k = 0, 1, \cdots, 6)$ bits in two steps (we just restricted the key length to a multiple of 32 bits to avoid the inconvenience of handing the

key in bits or bytes): first nonlinear-transform the user key into 8 expanded keys and then generate the required number of round keys from these expanded keys using simple linear operations. This two-step generation of round keys may be advantageous in case where storage requirements do not allow to store the whole round keys (e.g., implementation in a portable device with restricted resources). In such a circumstance, we may store only the 8 expanded keys and generate the round keys from these expanded keys each time of encryption/decryption.

Generation of Expanded Keys

1. Let $K = k_{u-1} \cdots k_1 k_0$ be the user key of u bytes ($u = 8 + 4i$, $i = 0, 1, \cdots, 6$). prepend (left-extend) as many zeros as needed to make K to 256 bits.

2. convert the resulting user key into 8 32-bit words $U[i] (0 \leq i \leq 7)$: $U[i] = k_{4i+3} k_{4i+2} k_{4i+1} k_{4i}$.

3. compute 8 expanded keys $E_e[i]$ using the basic transformations described before as follows:

$$\begin{aligned}
(V_e[3], V_e[2], V_e[1], V_e[0])^t &= (\tau \circ \gamma_o \circ \sigma_P \circ \pi_o)((U[6], U[4], U[2], U[0])^t), \\
(V_e[7], V_e[6], V_e[5], V_e[4])^t &= (\tau \circ \gamma_e \circ \sigma_Q \circ \pi_e)((U[7], U[5], U[3], U[1])^t), \\
T_0 &= V_e[0] \oplus V_e[1] \oplus V_e[2] \oplus V_e[3], \\
T_1 &= V_e[4] \oplus V_e[5] \oplus V_e[6] \oplus V_e[7], \\
E_e[i] &= V_e[i] \oplus T_1 \quad \text{for } i = 0, 1, 2, 3, \\
E_e[i] &= V_e[i] \oplus T_0 \quad \text{for } i = 4, 5, 6, 7,
\end{aligned}$$

where $P = (P_3, P_2, P_1, P_0)^t$ and $Q = (Q_3, Q_2, Q_1, Q_0)^t$ are constants given by

$$\begin{array}{llll}
P_0 = \texttt{0xbb67ae85} & P_1 = \texttt{0x3c6ef372} & P_2 = \texttt{0xa54ff53a} & P_3 = \texttt{0x510e527f} \\
Q_0 = \texttt{0x9b05688c} & Q_1 = \texttt{0x1f83d9ab} & Q_2 = \texttt{0x5be0cd19} & Q_3 = \texttt{0xcbbb9d5d}
\end{array}$$

These constants were obtained from the fractional parts of the first 8 odd primes, e.g., P_0 is the integer part of $2^{32}(\sqrt{3} - 1)$ and Q_7 is the integer part of $2^{32}(\sqrt{23} - 4)$.

Generation of encryption round keys

1. Let $K_e^i = (K_e[4i+3], K_e[4i+2], K_e[4i+1], K_e[4i])^t$ be the i-th round keys (the initial key addition step is considered as the 0-th round). Set the first 4 expanded keys to K_e^0 and the second 4 expanded keys to K_e^1.

2. The round keys K_e^{2i+2} (K_e^{2i+3}, resp.) for $i \geq 0$ are successively derived from the round keys K_e^{2i} (K_e^{2i+1}, resp.) by constant additions and rotations. More specifically, two of the 4 round keys in K_e^{2i} are updated to the corresponding two keys in K_e^{2i+2} by left-rotation by some multiples of 8 and the remaining two keys by constant addition. Table 1 shows the rotation amounts and constants used in successive evolutions. The constants RC_i's are define by

$$\begin{array}{lll}
RC_0 = \texttt{0x01010101} & RC_1 = \texttt{0x02020202} & RC_2 = \texttt{0x04040404} \\
RC_3 = \texttt{0x08080808} & RC_4 = \texttt{0x10101010} & RC_5 = \texttt{0x20202020}.
\end{array}$$

For example, the evolution of K_e^0 to K_e^2 is:

$$\begin{aligned}
K_e[8] &= ROL(K_e[0], 8), \\
K_e[9] &= RC_0 \oplus K_e[1], \\
K_e[10] &= ROL(K_e[2], 16), \\
K_e[11] &= RC_0 \oplus K_e[3],
\end{aligned}$$

where $ROL(X, n)$ denotes left rotation of X by n-bit positions.

i	$K_e^{2i} \rightarrow K_e^{2i+2}$				$K_e^{2i+1} \rightarrow K_e^{2i+3}$			
	3	2	1	0	3	2	1	0
0	RC_0	16	RC_0	8	24	RC_0	16	RC_0
1	8	RC_1	24	RC_1	RC_1	16	RC_1	8
2	RC_2	24	RC_2	16	8	RC_2	24	RC_2
3	16	RC_3	8	RC_3	RC_3	24	RC_3	16
4	RC_4	8	RC_4	24	16	RC_4	8	RC_4
5	24	RC_5	16	RC_5				

Table 1: Rotation amounts and constants for updating encryption round keys

Table 11 in Appendix A shows the encryption round keys expressed in terms of expanded keys. This table will be useful for key schedule cryptanalysis.

Let $K_d^i = (K_d[4i+3], K_d[4i+2], K_d[4i+1], K_d[4i])^t$ be the i-th round decryption keys. The decryption round keys may be derived from encryption round keys by component-wise transformations under ϕ_e / ϕ_o, as mentioned in the previous section (see eq. 3). However, using some properties of ϕ, we can derive a more efficient version of decryption key schedule.

First observe that the transformations $\phi_o = \tau \circ \pi_o \circ \tau$ and $\phi_e = \tau \circ \pi_e \circ \tau$ are actually row-wise bit permutations and can be rewritten as

$$\begin{aligned} \phi_o(A) &= (\phi_3(A[3]), \phi_2(A[2]), \phi_1(A[1]), \phi_0(A[0]))^t, \\ \phi_e(A) &= (\phi_0(A[3], \phi_3(A[2]), \phi_2(A[1]), \phi_1(A[0]))^t, \end{aligned}$$

where the component transformation ϕ_i is actually the same as π_i, except that 4 input bytes are now arranged in row vector (see Sect.2.2.2). Also note the shift property of ϕ_i

$$\begin{aligned} \phi_i(ROL(X, 8k)) &= ROL(\phi_i(X), 32 - 8k) \text{ for } k = 1, 2, 3, \\ \phi_i(X) &= ROL(\phi_j(X), 8) \text{ for } j = i + 1 \bmod 4, \end{aligned}$$

and the linear property under exclusive-oring

$$\phi_i(A[j] \oplus C) = \phi_i(A[j]) \oplus \phi_i(C).$$

In particular, $\phi_i(C) = C$ if C consists of 4 identical bytes.

Generation of decryption round keys

1. compute the expanded key E_d for decryption using ϕ_o / ϕ_e as follows:

$$\begin{aligned} \{E_d[3], E_d[2], E_d[1], E_d[0]\} &= \{\phi_0(K_e[51]), \phi_3(K_e[50]), \phi_2(K_e[49]), \phi_1(K_e[48])\}, \\ \{E_d[7], E_d[6], E_d[5], E_d[4]\} &= \{\phi_3(K_e[47]), \phi_2(K_e[46]), \phi_1(K_e[45]), \phi_0(K_e[44])\}. \end{aligned}$$

   Since $K_e[j]$'s can be expressed in terms of $E_e[i]$'s (see Table 11 in Appendix A), one can transform E_e into E_d using the shift and linear properties of ϕ_i as follows:

$$\begin{aligned} E_d[0] &= \phi_3(E_e[0]) \oplus RC_1 \oplus RC_3 \oplus RC_5, \\ E_d[1] &= \phi_0(E_e[1]) \oplus RC_0 \oplus RC_2 \oplus RC_4, \\ E_d[2] &= \phi_1(E_e[2]) \oplus RC_1 \oplus RC_3 \oplus RC_5, \\ E_d[3] &= \phi_2(E_e[3]) \oplus RC_0 \oplus RC_2 \oplus RC_4, \\ E_d[4] &= \phi_3(E_e[4]) \oplus RC_0 \oplus RC_2 \oplus RC_4, \\ E_d[5] &= \phi_3(E_e[5]) \oplus RC_1 \oplus RC_3, \\ E_d[6] &= \phi_3(E_e[6]) \oplus RC_0 \oplus RC_2 \oplus RC_4, \\ E_d[7] &= \phi_1(E_e[7]) \oplus RC_1 \oplus RC_3. \end{aligned}$$

2. Set the first 4 expanded keys to K_d^0 and the second 4 expanded keys to K_d^1. Then, derive the decryption round keys $K_d^i = \{K_d[4i+3], K_d[4i+2], K_d[4i+1], K_d[4i]\}$ $(i = 0, 1, \cdots, 12)$ successively, as in the encryption key schedule, using the constants and rotation amounts shown in Table 2.

Table 12 in Appendix A shows how the decryption round keys can be expressed in terms of expanded keys.

	$K_d^{2i} \rightarrow K_d^{2i+2}$				$K_d^{2i+1} \rightarrow K_d^{2i+3}$			
i	3	2	1	0	3	2	1	0
0	24	RC_5	16	RC_5	16	RC_4	8	RC_4
1	RC_4	8	RC_4	24	RC_3	24	RC_3	16
2	16	RC_3	8	RC_3	8	RC_2	24	RC_2
3	RC_2	24	RC_2	16	RC_1	16	RC_1	8
4	8	RC_1	24	RC_1	24	RC_0	16	RC_0
5	RC_0	16	RC_0	8				

Table 2: Rotation amounts and constants for updating decryption round keys

3 Security Analysis

We first investigate the diffusion property of linear transformations and the differential and linear characteristics of S-boxes together with their construction method. We then analyze the security of CRYPTON against various possible attacks.

3.1 Diffusion Property of Linear Transformations

Due to memory requirements, small size S-boxes are commonly used in most block cipher designs and thus the diffusion of S-box outputs by linear transformations plays a great role in providing resistance against various attacks such as the differential and linear attacks.

From Sect.2.2.2, we can see that it suffices to consider any one component transformation π_i of π to examine the diffusion property of π, since π acts on each byte column independently. Consider π_0, for example. It is easy to see that any column vector with n ($n < 4$) nonzero bytes is transformed by π_0 into a column vector with at least $4 - n$ nonzero bytes (this number 4 is called a diffusion order). This is due to the operation of exclusive-or sum in π. More important is that the number of such input vectors giving minimal diffusion is very limited. This is due to the masked bit permutation. Exhaustive search shows that there are only 204 values among 2^{32} possible values that achieve the minimum diffusion order 4. Furthermore, the nonzero bytes in input vector should have the same value to achieve minimal diffusion (see Tables 13 - 15 in Appendix B).

Input: $(d, c, b, a)^t$		Round								Possible values
Class	Type	1	2	3	4	5	6	7	8	of x, y (in hexa)
Class 1	$(0,0,0,x)$ $(0,0,x,0)$ $(0,x,0,0)$ $(x,0,0,0)$	1	3	9	3	1	3	9	3	x : 01,02,03; 04,08,0c 10,20,30; 40,80,c0
Class 2	$(0,0,x,x)$ $(0,x,x,0)$ $(x,x,0,0)$ $(x,0,0,x)$ $(0,y,0,y)$ $(y,0,y,0)$	2	2	6	6	2	2	6	6	y : 01,02,03; 04,08,0c 10,20,30; 40,80,c0 11,12,13; 21,22,23 31,32,33; 44,48,4c 84,88,8c; c4,c8,cc
Class 3	$(x,x,x,0)$ $(x,x,0,x)$ $(x,0,x,x)$ $(0,x,x,x)$	3	1	3	9	3	1	3	9	

Table 3: Number of nonzero bytes in S-box inputs in each round

Let us examine the diffusion effect of π through consecutive rounds. This analysis can be done by assuming that in each round the S-box output can take any desired value, irrespective of the input value. This assumption is to take into account the probabilistic nature of S-box transformation combined with unknown round keys. Since it suffices to consider worst-case propagations, we only examine the inputs with 1, 2, or 3 nonzero bytes in any one byte column, say the first byte column (see Appendix B). These input values can be divided into 3 classes as shown in Table 3. The sum of the number of nonzero bytes throughout the evolution is of great importance to ensure resistance against differential and linear

cryptanalysis. Table 3 shows that the number of nonzero bytes per round is repeated with period 4 and their sum up to 8 rounds is at least 32.

3.2 S-boxes Construction and their Property

The S-boxes for a block cipher should be chosen to have two important requirements: differential uniformity and nonlinearity. Combined with the diffusion effect of linear transformations used, they directly affect the security of the block cipher against differential and linear cryptanalysis (DC and LC, for short) [1, 15].

Following the formalization by Matsui [16], we define the differential and linear approximation probabilities of an S-box S (DP_S and LP_S for short) as follows. Let X and Y be a set of possible 2^n inputs/outputs of S, respectively. Then, DP_S and LP_S, respectively, are defined by

$$DP_S \stackrel{\text{def}}{=} \max_{\Delta x \neq 0, \Delta y} \frac{\#\{x \in X | S(x) \oplus S(x \oplus \Delta x) = \Delta y\}}{2^n}, \tag{4}$$

$$LP_S \stackrel{\text{def}}{=} \max_{\Gamma x, \Gamma y \neq 0} \left(\frac{\#\{x \in X | x \bullet \Gamma x = S(x) \bullet \Gamma y\} - 2^{n-1}}{2^{n-1}} \right)^2, \tag{5}$$

where $a \bullet b$ denotes the parity of bit-wise product of a and b.

The nonlinear transformation adopted in CRYPTON is byte-wise substitution using two 8×8 S-boxes, S_0 and S_1. We first constructed an S-box S_0 from three 4×4 S-boxes, P_0, P_1, P_2, using a 3-round Feistel cipher. That is, $y = S_0(x)$ is obtained by

$$\begin{aligned} x_l \| x_r &= x, \text{ where } |x_r| = |x_l| = 4, \\ y_r &= x_r \oplus P_1(x_l \oplus P_0(x_r)), \\ y_l &= x_l \oplus P_0(x_r) \oplus P_2(y_r), \\ y &= y_l \| y_r. \end{aligned}$$

Then the S-box S_1 is derived from S_0 as (see Figure 5) $S_1(x) = S_0(x)^{-1}$ for $x = 0, 1, \cdots, 255$. This technique to generate a larger S-box from smaller S-boxes was first introduced in MISTY [17] and also used in CS-cipher [21]. According to Nyberg and Knudsen [19], the S-boxes constructed as above will have $DP_{S_i} \leq 2p^2$ ($LP_{S_i} \leq 2p^2$, resp.) if each P_i is bijective with $DP_{P_i} \leq p$ ($DP_{P_i} \leq p$, resp.).

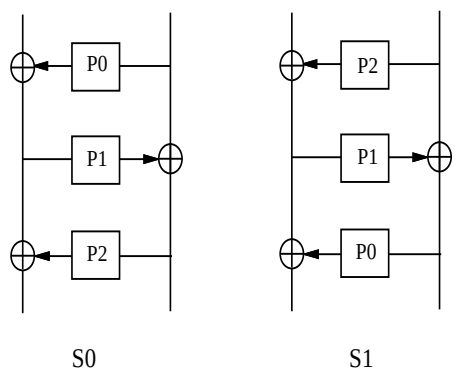

Figure 5: Construction of 8×8 S-boxes S_0 and S_1 from 4×4 S-boxes P_j ($j = 0, 1, 2$)

The 4×4 S-boxes shown in Table 4 were found in some restricted search space of 4-bit functions according to the following criteria:

- The S-boxes should have good differential and linear characteristics. More specifically, $DP_{P_i} = LP_{P_i} \leq 2^{-2}$. Furthermore, the number of difference pairs (selection patterns, resp.) with the best differential (linear approximation, resp.) probability in the resulting 8×8 S-boxes should be as small as possible when the input is restricted to the minimal diffusion set under π (see Table 3).

- The S-boxes P_0 and P_2 need not be one-to-one mappings. However, P_1 should be one-to-one for S_i's constructed from P_i's to achieve good differential and linear characteristics. We further required P_1 to satisfy the strict avalanche criterion (SAC).
- The S-boxes should be implemented in hardware using as small gates as possible. The algebraic degree of the component functions for P_0 and P_2 is restricted to two and that for P_1 to three for hardware efficiency.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
P_0	15	9	6	8	9	9	4	12	6	2	6	10	1	3	5	15
P_1	10	15	4	7	5	2	14	6	9	3	12	8	13	1	11	0
P_2	0	4	8	4	2	15	8	13	1	1	15	7	2	11	14	15

Table 4: 4×4 S-boxes P_0, P_1 and P_2

P_0	$y_0 = (x_2 \wedge x_3) \vee (\overline{x_1} \wedge \overline{x_3})$ $y_1 = (x_0 \wedge x_3) \vee (\overline{x_0} \wedge \overline{x_2})$ $y_2 = (x_1 \wedge x_2) \vee (\overline{x_0} \wedge \overline{x_2})$ $y_3 = (x_0 \wedge x_1) \vee (\overline{x_1} \wedge \overline{x_3})$
P_1	$y_0 = (x_3 \wedge (\overline{x_1} \vee (\overline{x_0} \wedge x_2))) \vee ((\overline{\overline{x_0} \wedge \overline{x_2}}) \wedge ((\overline{x_0} \wedge \overline{x_1}) \vee (\overline{x_2} \wedge \overline{x_3})))$ $y_1 = (x_0 \wedge (\overline{x_3} \vee (\overline{x_1} \wedge \overline{x_2}))) \vee ((\overline{x_1 \wedge \overline{x_2}}) \wedge ((\overline{x_0} \wedge x_1) \vee (\overline{x_2} \wedge \overline{x_3})))$ $y_2 = (x_1 \wedge (\overline{x_3} \vee (\overline{x_0} \wedge \overline{x_2}))) \vee ((\overline{\overline{x_0} \wedge \overline{x_2}}) \wedge ((\overline{x_0} \wedge \overline{x_1}) \vee (\overline{x_2} \wedge \overline{x_3})))$ $y_3 = (x_3 \wedge (\overline{x_0} \vee (x_1 \wedge \overline{x_2}))) \vee ((\overline{x_1 \wedge \overline{x_2}}) \wedge ((\overline{x_0} \wedge x_1) \vee (\overline{x_2} \wedge \overline{x_3})))$
P_2	$y_0 = (x_0 \wedge x_2) \vee (\overline{x_2} \wedge x_3)$ $y_1 = (x_1 \wedge x_3) \vee (\overline{x_1} \wedge x_2)$ $y_2 = (x_1 \wedge x_3) \vee (x_0 \wedge \overline{x_3})$ $y_3 = (x_0 \wedge x_2) \vee (\overline{x_0} \wedge x_1)$

Table 5: Boolean expressions for $y = P_i(x)$ ($y = y_3y_2y_1y_0, x = x_3x_2x_1x_0$)

The selected 4-bit S-boxes P_i's have the following DC and LC characteristics:

$$\begin{aligned} DP_{P_0} &= DP_{P_2} = 2^{-3}, \\ LP_{P_0} &= LP_{P_2} = 2^{-2}, \\ DP_{P_1} &= LP_{P_1} = 2^{-2}. \end{aligned}$$

The boolean expressions for implementing P_i's are given in Table 5. It is easy to see that P_0 and P_1 can be implemented using 16 nand gates and P_1 using 27 nand gates.

The two 8×8 S-boxes constructed from the selected 4-bit S-boxes are presented in Appendix C. The S-box S_0 has one fixed point and S_1 has two. Table 6 shows their statistics on the distribution of input-output difference/linear approximation pairs, where the entry value is the number computed by the numerator of equation (4) (equation (5) in the case of linear approximations).

Difference distribution					
entry value	0	2	4	6	8
no of entries	41503	16592	6576	560	304
Linear approx. distribution					
entry value	0	8	16	24	32
no of entries	26655	31200	7232	288	160

Table 6: Distribution of nonzero entries in the difference/linear approx. tables of S_i

From the table, we can see that for $i = 0, 1$

$$p_d \stackrel{\text{def}}{=} DP_{S_i} = \frac{8}{256} = 2^{-5}, \tag{6}$$

$$p_l \stackrel{\text{def}}{=} LP_{S_i} = (\frac{2 \cdot 32}{256})^2 = 2^{-4}, \tag{7}$$

and that there are 304 input-output difference pairs achieving the best characteristic probability p_d (160 input-output selection patterns achieving the best linear approximation probability p_l). However, if the input is restricted to the minimal diffusion set shown in Table 3, there are only 2 such pairs (8 such pairs in the case of linear approx.). Table 7 shows such pairs in each S-box.

S_0	DC	(80, 88) (80, c8)
	LC	(2, 80) (4, 80) (8, 10) (12, 8) (12, 11) (88, 2) (88, 84) (8c, 4)
S_1	DC	(88, 80) (c8, 80)
	LC	(2, 88) (4, 8c) (8, 12) (10, 8) (80, 2) (80, 4) (11, 12) (84, 88)

Table 7: Selected I/O pairs with maximum entry value in the difference/linear approx. tables of S_i

3.3 Differential Cryptanalysis

Differential cryptanalysis is a chosen plaintext attack introduced by Biham and Shamir [1], which tries to find the subkey of the last round by examining changes in the ciphertext bits in response to controlled changes in the plaintext bits. Differential cryptanalysis relies on the existence of highly probable characteristics/differentials. An *r-round characteristic* consists of a sequence of (input difference, output difference) pairs in each round up to round r, while an *r-round differential* only considers an input difference at the 1st round and an output difference at the r-th round [13]. Thus the probability of an r-round differential with input difference α and output difference β is equal to the sum of the probabilities of all r-round characteristics with input difference α and output difference β. Therefore, it would be right to consider differentials, rather than characteristics, to prove security against differential attacks.

Let us first evaluate the best r-round characteristic probability for CRYPTON. It can be shown (e.g., see [3]) that the probability of any characteristic in CRYPTON is completely determined by the number of active S-boxes and their characteristic probabilities. For worst-case analysis, suppose that the smallest active S-boxes involved in an r-round characteristic is α and all the S-boxes involved have the best characteristic probability p_d. Then, under the assumption of independent and uniform distribution for plaintexts and round keys, the probability p_{C_r} for the r-round characteristic is given by

$$p_{C_r} = p_d^{\alpha}.$$

From Table 3, we know that the best 8-round characteristic should involve at least 32 active S-boxes. (In the context of differential cryptanalysis the nonzero bytes in Table 3 should be thought of as input/output differences.) Therefore, the probability p_{C_8} that the best 8-round characteristic could take is at most

$$p_{C_8} = 2^{-160},$$

since our S-boxes have $p_d = 2^{-5}$. This figure shows that any differential attack based on the characteristic would require at least 2^{160} known plaintexts (more than available) and thus impossible. Furthermore, we can easily see from Table 7 and Tables in Appendix B that there exist no such characteristic.

Next, consider the best r-round differential probability p_{D_r}. Given a pair of input and output differences, there may be a relatively large number of characteristics starting with the input difference and ending with the output difference. With a little close examination of the diffusion property of linear transformations π and τ, we can obtain a very loose bound on the number of possible characteristics that can reside in any differential involving the smallest possible S-boxes. In fact, the number of such characteristics for 8-round CRYPTON can be bounded by $15^8 \times 4^2 \times 3^6 \times 2^3 \approx 2^{47.8}$. This figure was

obtained by assuming that each S-box can take any desired output difference for a given input difference. Even though we assume the existence of such number of best characteristics, the best 8-round differential probability is sufficiently small, i.e., $p_{D_8} = 2^{-112.2}$.

Note that the above figures were obtained by assuming that all S-boxes involved can take maximum possible values for differential and linear approximation probabilities (i.e., 2^{-5} and 2^{-4}, resp.). Actual figures will be much lower, since our S-boxes each have only 2 input/output difference pairs giving $p_d = 2^{-5}$ when difference values are limited to those with minimum diffusion (see Table 7) and linear transformations between rounds would make it impossible to form a chain only with best S-box characteristics.

The above analysis shows that we can get strong enough resistance against DC only with 10 rounds (assuming a 2R-attack). Though it is assumed in this analysis that the plaintexts and round keys are independent and uniformly random, which does not hold in practice, such an analysis has been found to provide a reasonable estimation on the security against DC.

3.4 Linear Cryptanalysis

Linear cryptanalysis is a known plaintext attack introduced by Matsui [15]. The key point in this attack is to find an effective linear approximation involving some bits of the plaintext (selected by input selection patterns), some bits of the ciphertext (selected by output selection patterns) and the associated key bits. If this linear approximation holds with some probability biased from $\frac{1}{2}$, then the correct value of the combination of key bits, thus equivalent one-bit key information, can be extracted by testing the satisfiability of the linear approximation with sufficiently many plaintext-ciphertext pairs. Obviously, the greater the bias, the fewer the number of plaintext-ciphertext pairs required to determine the correct key bit value.

In the context of linear cryptanalysis the nonzero bytes in Table 3 should be thought of as input/output selection patterns. Note that the S-boxes chosen for CRYPTON has the best linear approximation probability of $p_l = 2^{-4}$. The overall linear approximation involves a number of S-box linear approximations and the number of such S-boxes (i.e., active S-boxes) determines the complexity of linear cryptanalysis. As in differential cryptanalysis, it is known that the probability p_{L_r} for the best r-round linear approximation can be approximated by

$$p_{L_r} = p_l^{\alpha},$$

under the assumption that the plaintext and key bits are distributed independently and uniformly at random, where α denotes the total number of active S-boxes involved. Therefore, the best linear approximation probability for 8-round CRYPTON is approximated by

$$p_{L_8} = 2^{-128}.$$

Again this value is a very loose upper bound. Actually there will be no linear approximation achieving this probability, considering the linear characteristic of S-boxes and the linear transformations involved (see Table 7 and the related tables in Appendix B).

As in the differential attack, we may use multiple linear approximations to improve the basic linear attack [10, 11]. Suppose that one can derive N linear approximations involving the same key bits with the same probability. Then the complexity of a linear attack can be reduced by a factor of N, compared to a linear attack based on a single linear approximation [10]. However, a large number of linear approximations involving the same key bits are unlikely to be found in most ciphers, in particular in CRYPTON. Multiple linear approximations involving different key bits may be used to derive the different key bits in the different linear approximations simultaneously with almost the same complexity [11]. However, this will be of little help to improve the basic linear attack, since we already have a linear approximation probability far beyond a practical attack. Therefore, we believe that there will be no linear attack on 10-round CRYPTON with a complexity significantly lower than 2^{128}.

3.5 Security against Other Possible Attacks

There are some variants to the basic differential attack discussed above. Knudsen introduced the idea of *truncated differentials* [7]. A truncated (or partial) differential is a differential that predicts only part of the difference (not the entire value of difference). The existence of good truncated differentials and their usefulness depend on specific cipher algorithms. Knudsen demonstrated that this variant of DC may be

more effective against some kind of ciphers than the basic differential attack and may be independent of the S-boxes used [9]. Our preliminary analysis shows that truncated differentials are not much useful for differential cryptanalysis of CRYPTON compared to ordinary differentials.

Another variant of DC is the *higher order differential* attack considered by Lai [12]. This variant is also quite effective for some ciphers [7, 4]. Let d be the polynomial degree of $(r-1)$-round output bits expressed as polynomials of plaintext bits. Then the higher order differential attack allows us to find some key bits of the last round key for an r-round cipher using about 2^{d+1} chosen plaintexts [4]. Obviously the success of this attack depends on the nonlinear order of S-box outputs. Since CRYPTON uses S-boxes with nonlinear order 5, the polynomial degree of output bits after 4 rounds increases to $5^4 \gg 128$. Therefore, the higher order differential attack on CRYPTON will be completely infeasible after 4 rounds.

On the other hand, Jacobsen and Knudsen presented an interesting algebraic attack on block ciphers called the *interpolation attack* [4]. This attack exploits the fact that the ciphertext can be expressed as a polynomial of the plaintext with a fixed number (say n) of unknown coefficients and thus the encryption polynomial can be reconstructed given n pairs of plaintexts/ciphertexts encrypted under a fixed key K. This polynomial should then be equivalent to an encryption algorithm under the key K. Clearly the complexity of this attack depends upon the number of S-boxes applied throughout encryption and/or the polynomial degree of S-box outputs. The S-boxes used in CRYPTON do not allow a simple algebraic description. Furthermore, the bit permutation π in each round destroys any potential algebraic structure through the bit-wise mixing of the S-box outputs and encryption involves a large enough number of S-box applications. Therefore, we believe that CRYPTON also provides strong resistance against algebraic cryptanalysis, such as the interpolation attack.

3.6 Key Schedule Cryptanalysis

Key schedule cryptanalysis is another important category of attacks on block ciphers. Typical weaknesses exploited in key schedule cryptanalysis include weak keys or semi-weak keys, key collisions (equivalent keys), linear factors, simple relations such as the complemetation property existing in DES, etc. (for details, see e.g. [8, 5, 6]). These weaknesses can be exploited to speed up an exhaustive key search or to mount related key attacks. Though most of these attacks on key schedules are not practical in normal use, they may be a serious flaw in certain circumstances (e.g., when a block cipher is used a basic building block for hash functions).

The key schedule for CRYPTON is designed with the above known weaknesses in mind. There is no weak keys or semi-weak keys. Weak keys exist in CRYPTON if there exist round keys such that

$$K_e[4k+j] = K_e[48-4k+j] \ (0 \le k \le 5,\ 0 \le j \le 3).$$

Referring to Table 11, we can easily check that there are no weak keys (This is mainly due to the use of different round constants in each round). There are no semi-weak keys either, since no weak keys exist whatever values the expanded keys take.

The 8 expanded keys are derived from a user key via an invertible transformation and thus no different user keys can produce the same expanded keys. This guarantees that there is no equivalent keys.

There is no complementation property either, since both key expansion and encryption processes involve parallel nonlinear substitutions. The same reason ensures that there will be no other simple relations between different user keys.

We also believe that there are no related keys that can be used to mount related-key differential attacks. First, a user-supplied key is transformed into expanded keys by a nonlinear bijective transformation, ensuring that any controlled change in the user key should result in at least one-byte change in one of the expanded keys. Second, the same 8 expanded keys are used six or seven times throughout encryption, each time at least two round keys being rotated and thus applied to different locations of data array. Finally, parallel nonlinear substitutions in each round make it difficult for an attacker to exploit any controlled change in an expanded key.

The CRYPTON key schedule is designed to be very efficient in both software and hardware implementation. In hardware the 8 expanded keys can be retained in registers and updated at each round by rotations and constant additions. In software the whole round keys can be generated and stored for use in multiple blocks of encryption.

4 Implementation and Efficiency

The cipher CRYPTON is designed to be highly parallelizable, considering the current trend of microprocessor technology and the efficiency of hardware implementation. Today's most microprocessors are adopting multiple levels of pipelining and a certain amount of parallelism to maximize their performances. Thus parallelizability has been one of important design criteria in modern algorithm designs. The round function of CRYPTON actually consists of three steps of parallelizable operations. Therefore we can expect that CRYPTON will be extremely fast in both software and hardware implementation.

4.1 Implementation on 32-bit Microprocessors

The round transformation of CRYPTON can be efficiently implemented on a 32-bit microprocessor using table lookups, if we use 4 KBytes of storage in addition. The idea is to precompute and store 4 tables of 256 words ($0 \le j \le 255$) as follows:

$$\begin{aligned}
SS_0[j] &= S_0[j] \wedge m_3 \parallel S_0[j] \wedge m_2 \parallel S_0[j] \wedge m_1 \parallel S_0[j] \wedge m_0, \\
SS_1[j] &= S_1[j] \wedge m_0 \parallel S_1[j] \wedge m_3 \parallel S_1[j] \wedge m_2 \parallel S_1[j] \wedge m_1, \\
SS_2[j] &= S_0[j] \wedge m_1 \parallel S_0[j] \wedge m_0 \parallel S_0[j] \wedge m_3 \parallel S_0[j] \wedge m_2, \\
SS_3[j] &= S_1[j] \wedge m_2 \parallel S_1[j] \wedge m_1 \parallel S_1[j] \wedge m_0 \parallel S_1[j] \wedge m_3.
\end{aligned}$$

Then it is easy to see that the odd round function $B = \rho_{oK}(A)$ can be implemented by

$$\begin{aligned}
B_0 &= SS_0[a_{00}] \oplus SS_1[a_{10}] \oplus SS_2[a_{20}] \oplus SS_3[a_{30}] \oplus K[0], \\
B_1 &= SS_1[a_{01}] \oplus SS_2[a_{11}] \oplus SS_3[a_{21}] \oplus SS_0[a_{31}] \oplus K[1], \\
B_2 &= SS_2[a_{02}] \oplus SS_3[a_{12}] \oplus SS_0[a_{22}] \oplus SS_1[a_{32}] \oplus K[2], \\
B_3 &= SS_3[a_{03}] \oplus SS_0[a_{13}] \oplus SS_1[a_{23}] \oplus SS_2[a_{33}] \oplus K[3].
\end{aligned}$$

Similarly, the even round $B = \rho_{eK}(A)$ can be implemented by

$$\begin{aligned}
B_0 &= SS_1[a_{00}] \oplus SS_2[a_{10}] \oplus SS_3[a_{20}] \oplus SS_0[a_{30}] \oplus K[0], \\
B_1 &= SS_2[a_{01}] \oplus SS_3[a_{11}] \oplus SS_0[a_{21}] \oplus SS_1[a_{31}] \oplus K[1], \\
B_2 &= SS_3[a_{02}] \oplus SS_0[a_{12}] \oplus SS_1[a_{22}] \oplus SS_2[a_{32}] \oplus K[2], \\
B_3 &= SS_0[a_{03}] \oplus SS_1[a_{13}] \oplus SS_2[a_{23}] \oplus SS_3[a_{33}] \oplus K[3].
\end{aligned}$$

Therefore, one round of CRYPTON can be performed using 20 table lookups (16 to SS-boxes, 4 to round keys), 16 XORs, 12 shifts and 16 ANDs (for byte extraction). (If there are a number of registers available, as in most RISC machines, it may be more efficient to use the 8 expanded keys during encryption/decryption instead of round keys, since we only need 2 XORs and 2 rotates for round key computation from expanded keys. This method of partial key schedule is also expected to yield a better performance in the case of frequent key change, e.g., when CRYPTON is used for hashing.) Due to the parallel processing of data, 8 intermediate data variables cannot be managed by registers on most PCs and thus some of these variables should be loaded/stored from/to memory.

We have implemented CRYPTON on 200 MHz Pentium Pro running Windows 95 (with 32 Mbytes of RAM) and on 167 MHz UltraSparc running Solaris 2.5 (the two C codes are a little different.) The result is shown in Table 8 (These timings were obtained using our core routines without including AES-API overheads. The codes submitted to NIST were not fully optimized due to our tight time schedule. The presented timings were obtained with the latest version of our optimized codes. The updated codes are available from my home page at *http://crypt.future.co.kr/~chlim*). Our optimized C code runs quite fast, giving an encryption rate of about 6.4 Mbytes/sec on Pentium Pro. The partial assembly code can encrypt/decrypt about 7.8 Mbytes per second, running about 20 % faster than the optimized C code. (Only the encryption routine is implemented in-line assembly.) We expect that a full assembly language implementation will be a little faster. On UltraSparc, CRYPTON runs somewhat slower, achieving an encryption rate of about 4.1 Mbytes/sec.

The key setup time of CRYPTON is different for encryption and decryption. Decryption key setup requires a little more computations, i.e., transformation of expanded keys. Our encryption key schedule

is very fast, taking much less time than one-block encryption. As a result, CRYPTON is very efficient even in the case of encrypting/decrypting only a few blocks of data. Note that the key setup time remains almost the same for different sizes of user keys.

CRYPTON should be initialized with a table for S-boxes at the first time of use. Once generated, these S-boxes are embedded into the code. The table generation ($P_j(j = 0, 1, 2) \rightarrow S_0, S_1 \rightarrow SS_i(0 \leq i \leq 3)$) takes a relatively large amount of time compared to key scheduling. But this will cause no problem in practice, since it is required only once at the algorithm setup time. (Actually we need not generate SS-boxes. In most cases, it suffices to write into a file (to be included in the encryption routine) appropriately masked versions of S-boxes, as described above.)

Language\Clocks	Alg. setup	Key setup (enc/dec)	Enc/Dec
In-line Asm (PC)	N/A	N/A	390 / 390
MSVC 5.0 (PC)	9740	325 / 360	475 / 475
GNU C (UltraSparc)	11460	470 / 520	615 / 615

Table 8: Speed of CRYPTON on Pentium Pro and UltraSparc

4.2 Implementation on 8-bit Microprocessors

Since CRYPTON essentially processes data byte by byte, it is very simple and efficient to implement on 8-bit microprocessors. It uses only simple operations, such as logical ANDs, XORs and table lookups. The 8-bit S-boxes S_i $(i = 0, 1)$ take 512 bytes of EEPROM and intermediate data variables require only 20 bytes of RAM. Furthermore, it is reasonable to assume that 32-byte expanded keys can reside within RAM during encryption, so there may be no need to generate round keys. This will be of great advantage when implemented on small portable devices such as smart cards.

If a target microprocessor cannot accommodate even 512 bytes of storage, each S-box table entry can be directly computed from three 4×4 S-boxes P_i $(i = 0, 1, 2)$. In this case the table needs only 48 bytes of memory (or even can be packed to 24 bytes if a few more operations are used to access the table entry). Then, we can obtain one S-box table entry by using 10 operations (3 table lookups, 3 XORs, 2 AND/OR and 2 shifts (to extract/combine two 4-bit values from/to a byte value)).

We do not have access to an 8-bit microprocessor, so we simply estimate the number of required instructions (cycles) under the following computing model:

- The microprocessor has two Accumulator registers and sufficient memory (EEPROM) to hold the two 8×8 S-boxes.
- Destination of any instruction must be Accumulator. Therefore, data must be first loaded into Accumulator, processed there using other data in a register or memory and then stored back into memory.
- We only use three kinds of instructions, each of which is assumed to take 3 clock cycles: register-to-memory/memory-to-register MOV, register-to-register/memory-to-register XOR and immediate-to-register/memory-to-register AND. Note that some instruction (e.g., memory access by indexed addressing) may require more clock cycles, while some other instructions (e.g., AND/XOR with immediate data) take less cycles. By averaging, we made a uniform 3 cycle assumption to simplify the speed estimation.

We estimated two cases of encryption/decryption time. If a microprocessor has a sufficient RAM space for intermediate data variables and round keys, and if many data blocks need to be encrypted/decrypted, then we can generate and store the encryption/decryption round keys from a user-supplied key. After then, data can be encrypted/decrypted using the round keys stored in RAM. On the other hand, if only a small amount of RAM is available, then we have to encrypt/decrypt data together with in-line key scheduling. Table 9 shows the resource requirements and the estimated speeds for each scenario. We did not try any optimization for this estimation. Note that component functions of encryption/decryption are re-grouped into different round transformations to facilitate the estimation. The difference in the number

of instructions (416) between one-time encryption and decryption exactly corresponds to the operations needed to transform expanded keys (two times of $\tau \circ \pi \circ \tau$ and constant additions, see Sect.2.4).

RAM	20 (data) + 208 (round keys) = 228 bytes		
EEPROM	512 bytes for S-boxes		
	# of instructions	# of cycles	notes
$\gamma \circ \sigma$	64	192	round 1
$\gamma \circ \sigma \circ \tau \circ \pi$	224	672	round 2 - 11
$\sigma \circ \tau \circ \gamma \circ \sigma \circ \tau \circ \pi$	240	720	round 12
Total Enc/Dec	2544	7632	Round keys in RAM
Key Expansion	576	1728	from 32-byte UserKey
Enc Round key Gen.	520	1560	from 32-byte ExpKey
Dec Round key Gen.	670	2010	from 32-byte ExpKey
Total Enc Key Setup	1096	3288	
Total Dec Key Setup	1246	3738	

RAM	20 (data) + 32 (UserKey) = 52 bytes		
EEPROM	512 bytes for S-boxes		
	# of instructions	# of cycles	notes
Key Expansion	576	1728	
Trans. of ExpKeys	416	1248	$E_e \rightarrow E_d$
RoundKey Update	40	120	11 times
$\tau \circ \pi \circ \tau$	192	576	for final round keys
Total Encryption	3752	11256	with In-line Key Scheduling
Total Decryption	4168	12504	with In-line Key Scheduling

Table 9: Estimated speed of CRYPTON on an 8-bit microprocessor

4.3 Software Implementations on Other Platforms

CRYPTON can also be efficiently implemented on other platforms using the table lookup method described above, for example, on 16-bit or 64-bit microprocessors or digital signal processors. It can be expected that CRYPTON will run a little faster on a 64-bit microprocessor than on a 32-bit microprocessor, since eight 32-bit data variables can be managed with four 64-bit registers. The number of required operations is almost the same if we use 12 Kbytes of storage. Also, CRYPTON will be ideal to be implemented on DSPs which have multiple execution units such as TMS320C6x. We are working on implementation on TMS320C6x.

4.4 VLSI Implementation

CRYPTON is designed by taking into account efficient hardware implementations. The 4×4 S-boxes P_0 and P_2 can be implemented using 16 nand gates of depth 3 and P_1 needs 27 nand gates of depth 5 (see Table 5). Thus, each S-box can be implemented using 107 nand gates of depth 20. Therefore, we can see that one round of CRYPTON can be implemented using 3248 nand gates with depth 26. The initial key addition and the special final round together can be implemented using 2736 nand gates. We also need two 128-bit registers as input and output data buffers.

Generation of 8 expanded keys (for encryption) requires the circuit equivalent in gate count to two round encryptions plus 14 XORs of 32-bit words. This amounts to 8288 nand gates. We also need 2048 nand gates for conversion of expanded keys. Round key generation (from round 2) only needs 32 nand gates per round. There also should be two 128-bit registers to buffer 256-bit expanded keys and one 32-bit register to update a round constant. Therefore, the whole key scheduling part can be implemented using 10688 nand gates and a 288-bit register.

With a cheap technology we may implement just two rounds of encryption and iterate this circuit 6 times to encrypt one block. In this case, we also need to implement the initial key addition (512 nand gates) and the final output transformation ($\tau \circ \pi_e \circ \tau$; 1024 nand gates). This circuit can be implemented using 8032 nand gates and a 384-bit register, where we assume that pre-computed round keys are applied to the circuit and thus only one 128-bit register is counted for key scheduling.

On the other hand, we can implement the full 12-round CRYPTON, including the complete key scheduling part, for high speed applications such as ATM, HDTV, B-ISDN and PCI bus, etc. This circuit can be implemented using 49152 nand gates and a 544-bit register. Table 10 summarizes the number of nand gates required for this full implementation. Note that the time to pass the key expansion part corresponds to an initial key setup delay in hardware implementation. For multiple blocks of encryption/decryption, the expanded keys can be retained in registers for later use. We have not yet carried out any simulation to estimate the required number of clocks.

	# of nand gates	depth	note
Initial KeyAdd	512	3	
Round Trans	3248	26	round 1 - 11
Final round	2224	23	
Total Enc/Dec	38464	312	+ 256-bit register
Key Expansion	8288	35	
RoundKey Update	32	3	round 2-12
ExpKey Trans	2048	6	
Total Key Schedule	10688	doesn't matter	+ 288-bit register

Table 10: Estimated gate count for a full parallel implementation of CRYPTON

5 Other Considerations and Discussions

5.1 The Number of Rounds and Possible Variants

Our preliminary analysis shows that the complexity of differential and linear attacks on 10-round CRYPTON would require more ciphertexts than available. We thus propose to use 12 rounds for CRYPTON, with a margin of two more rounds. We believe that this number of rounds will be far sufficient to thwart any known attack against block ciphers. Nevertheless, if desired, we may increase the number of rounds to, say, 16 by extending the key scheduling process. Also, the number of rounds may be increased as the size of a user key increases. However, in this version of proposal, we did not consider such a variable number of rounds.

Though we designed CRYPTON with a fixed 128-bit block size, it is quite easy to modify it into a cipher with other block size. In fact, the bit permutation π is better suited for use in a 64-bit or 512-bit block cipher. For example, we can design a 64-bit block cipher using four 4-bit S-boxes and a variant of π in which a new 4-bit nibble is formed by extracting just one bit from each XORed nibble in the same column. This variant will have better differential and linear characteristics, since there are smaller number of characteristics constituting a differential. For fast bulk encryption on a 64-bit microprocessor, we may use a variant of 512-bit block size. This variant processes a 512-bit data block by representing it in 8×8 byte array.

5.2 Advantages and Limitations

Simplicity and Easy Analysis: The simplicity of CRYPTON allows easy analysis against various known attacks, including differential and linear cryptanalysis. This simple analysis in turn enables us to determine the number of rounds required to guarantee high enough security against those attacks. Though we couldn't carry out a complete analysis against all possible attacks, its simple design will make it easier to perform various security evaluations.

Efficiency in both HW and SW: CRYPTON uses only very simple instructions (AND, XOR, rotate by a multiple of 8 and table lookups), which allows very efficient implementations in both large microprocessors and small microprocessors. Various tradeoffs are possible between speed and memory (e.g., with memory of 48, 512, 1K, 2K, 4K, 12K, 512K bytes etc.). The S-boxes were carefully designed to enable efficient implementation with a simple hardware logic. All the other components of CRYPTON can be implemented in hardware only using XOR gates (1 XOR gate = 4 nand gates). Our estimation on the gate count shows that CRYPTON can be implemented very efficiently in hardware. We also made much effort to make the encryption and decryption processes identical. This greatly reduces the code size (in software) and the gate count (in hardware).

Fast Key Scheduling: The key scheduling algorithm is also very efficient and appears to be secure against various attacks on key schedules. It is designed by taking into account various applications: Two step generation of round keys will be very useful in the environment with limited resources. Fast key setup time makes CRYPTON very advantageous when used for a few blocks of encryption (in particular, for hashing). The CRYPTON key schedule allows to use any size of a user key (Key size is restricted to minimum 64 bits for security) and can be easily extended as the number of rounds increases.

CRYPTON has a fixed block size of 128 bits. However, it is possible to modify CRYPTON to operate in other block sizes as mentioned before. We may also double the block length by using the Luby-Rackoff construction [14].

5.3 Mode of Operations

CRYPTON can be used as a building block for various applications. These include collision-resistant hash functions, pseudo-random number generators, stream ciphers and message authentication codes (MACs).

Hash Functions: There have been proposed a lot of methods to construct hash functions from block ciphers (e.g., see [20]). Some well-known constructions include the Matyas-Meyer-Oseas construction and the Davies-Meyer construction. A hash function based on a block cipher is in general much slower than the underlying block cipher due to the key schedule overhead. One block hashing typically requires one-time key schedule and one block encryption. Our key schedule runs very fast, taking much less time than one block encryption. Actually we could achieve a hashing speed of about 60 % of the encryption speed with our C implementation. For a longer hash code we may use double-length hash constructions (for 256-bit hash code in the case of CRYPTON), such as the one described in ISO/IEC 10118-2.

Message Authentication Codes: Block ciphers are widely used to generate a MAC. The most popular method is to encrypt a message in CBC mode and take the last ciphertext (or part of it) as a MAC (CBC-MAC). CRYPTON can also be used for this purpose.

Stream Ciphers: CRYPTON can be used to generate a keystream for a stream cipher. For example, we can use CRYPTON as a synchronous stream cipher by running it in OFB (output feedback) mode, or as a self-synchronizing stream cipher by running it in CFB (cipher feedback) mode.

Pseudo-random Number generators: Block ciphers are often used for pseudo-random number generation. The simplest such generator is to run a block cipher in counter mode or in OFB mode. There also exist cryptographically more strong PRN generators, such as the PRN generator suggested in ANSI X.9.17. CRYPTON can be used for this purpose as well.

5.4 Historical Remarks

The overall structure of CRYPTON is borrowed from Square [2]. However, CRYPTON completely differs from Square in its bit-permutation and S-box construction. These two components are essential parts in block cipher designs. We decided to start with the parallel structure of Square, considering its efficiency in modern microprocessors implementing more and more parallelisms. One disadvantage of this structure is that more registers are needed for handling intermediate variables.

The bit permutation π has diffusion order 4, while the MDS matrix multiplication used in Square has diffusion order 5. However, our bit permutation is very simple and efficient (it can be implemented only using XORs) and, since it is involution (i.e., $\pi = \pi^{-1}$), we could make the encryption and decryption processes identical. There are a lot of choices for a masking vector M for π. We did not test all possible values for masking bytes m_i's. Our choice is made based on the ease of analysis (systematic diffusion property when rotated) and the number of minimal diffusion elements.

There were some changes in S-box construction. At a first time, we tried to use just one 8×8 S-box constructed from an inverse polynomial over GF(2^8), since such an S-box is self-invertible and has very good differential and linear characteristics [18]. However, such S-boxes can only be implemented in hardware using ROM or EEPROM. Then, the speed of a cipher will be limited by the memory access time. The speed limitation may be even worse, considering parallel accesses to the S-boxes in CRYPTON. Therefore, we wanted the S-boxes to be efficiently implemented with a simple hardware logic. The final S-box S_0 was constructed so that it can be implemented using as small number of nand gates as possible.

5.5 Future Directions

We are currently working toward some improvements in security with minor changes in the algorithm. One direction is to find better S-boxes. For example, we have a little uneasy feeling for the bit/byte permutation used in CRYPTON. The bit permutation only mixes the bits in the same bit column (i.e., unchanges the bit positions in bytes). Though the byte substitution nonlinearly transforms the resulting bytes into (presumably) new random bytes (combined with key addition), this fact might be a weak point if the S-boxes would have some relevant, unidentified weakness. We have found no weakness yet, but in the next revision we will reflect the following change in S-boxes.

The present version uses just two S-boxes, i.e., S_0 and $S_1 = S_0^{-1}$. In the revised version we will add two more variants of S_0 to make it more difficult to form a chain of good characteristics. The four S-boxes are constructed as: S_0, $S_1(x) = S_0(x)^{\lll 6}$ for $x \in [0, 256)$, $S_2 = S_0^{-1}$, and $S_3 = S_1^{-1}$. We decided to use variants of one S-box, instead of independent S-boxes, to allow greater flexibility in memory requirements (in particular, considering the environment with limited resources, such as smart cards). Actually we have already updated this change in both the code and document. (This revised version is available from my home page at *http://crypt.future.co.kr/~chlim.*) This change increases the storage requirement for 8-bit microprocessors (512 bytes to 1024 bytes). With the same 512 bytes of EEPROM we would need 8 rotations per round in addition. However, it does not affect the efficiency on large microprocessors or in hardware.

Another direction is to find a better masking vector for bit permutation. For example, we may use different bit permutations in odd and even rounds. Finally, the use of essentially the same round keys in the initial and final key addition steps may be exploited to speed up some variants of differential cryptanalysis, as is often the case in the existing algorithms. The key scheduling algorithm will also be carefully reviewed in the next revision. We will continue to improve/ analyze the security and efficiency of CRYPTON and reflect any improvement found in the next revision.

6 Conclusion

We have described a new 128-bit block cipher CRYPTON proposed as a candidate algorithm for AES and analyzed its security. CRYPTON uses the same algorithm for encryption and decryption with different key schedules, and supports variable key-length up to 256 bits. Its high parallelism allows fast implementation in both software and hardware. Our analysis shows that 12-round CRYPTON is secure against most known attacks. At present the best attack on CRYPTON appears to be exhaustive key search. However, as usual, more extensive analysis should be done before practical applications of a newly introduced cipher, so we strongly encourage the reader to further investigate CRYPTON with various ways of attack. We would greatly appreciate reports of any weakness found.

References

[1] E. Biham and A. Shamir, Differential cryptanalysis of DES-like cryptosystems, *Journal of Cryptology*, v. 4, 1991, pp. 3-72.

[2] J.Daemen, L.Knudsen and V.Rijmen, The block cipher Square, In *Fast Software Encryption*, Lecture Notes in Computer Science (LNCS) 1267, Springer-Verlag, 1997, pp.149-171.

[3] H.M.Heys and S.E.Tavares, Substitution-permutation networks resistant to differential and linear cryptanalysis, *J. Cryptology*, 9(1), 1996, pp.1-19.

[4] T.Jakobsen and L.R.Knudsen, The interpolation attack on block ciphers, In *Fast Software Encryption*, LNCS 1267, Springer-Verlag, 1997, pp.28-40.

[5] J.Kelsey, B.Schneier and D.Wagner, Key-schedule cryptanalysis of IDEA, DES, GOST, SAFER, and triple-DES, In *Advances in Cryptology-Crypto'96*, LNCS 1109, Springer-Verlag, 1996, pp.237-252.

[6] J.Kelsey, B.Schneier and D.Wagner, Related-key cryptanalysis of 3-WAY, Biham-DES, CAST, DES-X, NewDES, RC2, and TEA, In *Proc. of ICICS'97*, Beijing, 1997.

[7] L.R.Knudsen, Truncated and higher order differentials, In *Fast Software Encryption*, LNCS 1008, Springer-Verlag, 1995, pp.196-211.

[8] L.R.Knudsen, A key schedule weakness in SAFER-K64, In *Advances in Cryptology-Crypto'95*, LNCS 963, Springer-Verlag, 1995, pp.274-286.

[9] L.R.Knudsen and T.A.Berson, Truncated differentials of SAFER, In *Fast Software Encryption*, LNCS 1039, Springer-Verlag, 1996, pp.15-26.

[10] B.S.Kaliski Jr. and M.J.B.Robshaw, Linear cryptanalysis using multiple linear approximations, In *Advances in Cryptology-Crypto'94*, LNCS 839, Springer-Verlag, 1994, pp.26-39.

[11] B.S.Kaliski Jr. and M.J.B.Robshaw, Linear cryptanalysis Using multiple linear approximations and FEAL, In *Fast Software Encryption*, LNCS 1008, Springer-Verlag, 1995, pp.249-264.

[12] X.Lai, *On the design and security of block ciphers*, PhD thesis, ETH, Zurich, 1992.

[13] X.Lai and J.L.Massey, Markov ciphers and differential cryptanalysis, In *Advances in Cryptology-Eurocrypt'91*, LNCS 547, Springer-Verlag, 1991, pp.17-38.

[14] M.Luby and C.Rackoff, How to construct pseudo-random permutations from pseudo-random functions, *SIAM J.Compt.*, Apr.1988, pp.373-386.

[15] M.Matsui, Linear cryptanalysis method for DES cipher, In *Advances in Cryptology-Eurocrypt'93*, LNCS 765, Springer-Verlag, 1994, pp.386-397.

[16] M.Matsui, New structure of block ciphers with provable security against differential and linear cryptanalysis, In *Fast Software Encryption*, LNCS 1039, Springer-Verlag, 1996, pp.205-218.

[17] M.Matsui, New block encryption algorithm MISTY, In *Fast Software Encryption*, LNCS 1267, Springer-Verlag, 1997, pp.54-68.

[18] K. Nyberg, Differentially uniform mappings for cryptography, In *Advances in Cryptology-Eurocrypt'93*, LNCS 765, Springer-Verlag, 1994, pp. 55-64.

[19] K. Nyberg and L.Knudsen, Provable security against differential cryptanalysis, *J. Cryptology*, Vol.8, No. 1, 1995, pp.27-37.

[20] B.Schneier, *Applied Cryptography*, 2nd edition, Sect.18.11, John Wiley & Sons, 1996.

[21] J.Stern and S.Vaudenay, CS-cipher, In *Fast Software Encryption, 5th International Workshop Proceedings*, Springer-Verlag, 1998, pp.189-205.

A Encryption/Decryption Round Keys in Terms of Expanded Keys

Tables 11 and 12 show the encryption/decryption round keys expressed in terms of 8 expanded keys.

$K_e[0] = E_e[0]$ $K_e[1] = E_e[1]$ $K_e[2] = E_e[2]$ $K_e[3] = E_e[3]$	$K_e[4] = E_e[4]$ $K_e[5] = E_e[5]$ $K_e[6] = E_e[6]$ $K_e[7] = E_e[7]$
$K_e[8] = ROL(E_e[0], 8)$ $K_e[9] = E_e[1] \oplus RC_0$ $K_e[10] = ROL(E_e[2], 16)$ $K_e[11] = E_2[3] \oplus RC_0$	$K_e[12] = E_e[4] \oplus RC_0$ $K_e[13] = ROL(E_e[5], 16)$ $K_e[14] = E_e[6] \oplus RC_0$ $K_e[15] = ROL(E_e[7], 24)$
$K_e[16] = ROL(E_e[0], 8) \oplus RC_1$ $K_e[17] = ROL(E_e[1], 24) \oplus RC_0$ $K_e[18] = ROL(E_e[2], 16) \oplus RC_1$ $K_e[19] = ROL(E_e[3], 8) \oplus RC_0$	$K_e[20] = ROL(E_e[4], 8) \oplus RC_0$ $K_e[21] = ROL(E_e[5], 16) \oplus RC_1$ $K_e[22] = ROL(E_e[6], 16) \oplus RC_0$ $K_e[23] = ROL(E_e[7], 24) \oplus RC_1$
$K_e[24] = ROL(E_e[0], 24) \oplus RC_1$ $K_e[25] = ROL(E_e[1], 24) \oplus RC_{02}$ $K_e[26] = ROL(E_e[2], 8) \oplus RC_1$ $K_e[27] = ROL(E_e[3], 8) \oplus RC_{02}$	$K_e[28] = ROL(E_e[4], 8) \oplus RC_{02}$ $K_e[29] = ROL(E_e[5], 8) \oplus RC_1$ $K_e[30] = ROL(E_e[6], 16) \oplus RC_{02}$ $K_e[31] = E_e[7] \oplus RC_1$
$K_e[32] = ROL(E_e[0], 24) \oplus RC_{13}$ $K_e[33] = E_e[1] \oplus RC_{02}$ $K_e[34] = ROL(E_e[2], 8) \oplus RC_{13}$ $K_e[35] = ROL(E_e[3], 24) \oplus RC_{02}$	$K_e[36] = ROL(E_e[4], 24) \oplus RC_{02}$ $K_e[37] = ROL(E_e[5], 8) \oplus RC_{13}$ $K_e[38] = ROL(E_e[6], 8) \oplus RC_{02}$ $K_e[39] = E_e[7] \oplus RC_{13}$
$K_e[40] = ROL(E_e[0], 16) \oplus RC_{13}$ $K_e[41] = E_e[1] \oplus RC_{04}$ $K_e[42] = ROL(E_e[2], 16) \oplus RC_{13}$ $K_e[43] = ROL(E_e[3], 24) \oplus RC_{04}$	$K_e[44] = ROL(E_e[4], 24) \oplus RC_{04}$ $K_e[45] = ROL(E_e[5], 16) \oplus RC_{13}$ $K_e[46] = ROL(E_e[6], 8) \oplus RC_{04}$ $K_e[47] = ROL(E_e[7], 16) \oplus RC_{13}$
$K_e[48] = ROL(E_e[0], 16) \oplus RC_{15}$ $K_e[49] = ROL(E_e[1], 16) \oplus RC_{04}$ $K_e[50] = ROL(E_e[2], 16) \oplus RC_{15}$ $K_e[51] = ROL(E_e[3], 16) \oplus RC_{04}$	

Notes :

$RC_{02} = RC_0 \oplus RC_2$, $RC_{13} = RC_1 \oplus RC_3$,
$RC_{04} = RC_0 \oplus RC_2 \oplus RC_4$, $RC_{15} = RC_1 \oplus RC_3 \oplus RC_5$.

Table 11: Encryption round keys in terms of expanded keys

B The Minimal Diffusion Set Under π_0

The bit permutation π_i (acting on a 4-byte column vector) has the property that the sum of the number of nonzero bytes in input and output is at least 4. (We call this sum as the diffusion order of the permutation.) Below are given all possible input/output vectors achieving such minimal diffusion under π_0. There are 48 values giving 1-to-3 (or 3-to-1) propagation (see Table 13) and 108 values giving 2-to-2 propagation (see Table 15). Note that in each of these input/output pairs the value of nonzero bytes is always the same. Any input vector with nonzero bytes of different values has a diffusion order greater than 4.

Note that there are 30 possible nonzero byte values in the case of 2-to-2 propagation with separated nonzero bytes. All the other cases have only 12 possible values. Thus, it would be better to choose an input/output difference pair from the former case in order to maximize the number of characteristics residing in a particular differential.

C CRYPTON S-boxes

The two 8×8 S-boxes used in CRYPTON are given in Tables 16 and 17.

$K_d[0] = E_d[0]$ $K_d[1] = E_d[1]$ $K_d[2] = E_d[2]$ $K_d[3] = E_d[3]$	$K_d[4] = E_d[4]$ $K_d[5] = E_d[5]$ $K_d[6] = E_d[6]$ $K_d[7] = E_d[7]$
$K_d[8] = E_d[0] \oplus RC_5$ $K_d[9] = ROL(E_d[1], 16)$ $K_d[10] = E_d[2] \oplus RC_5$ $K_d[11] = ROL(E_2[3], 24)$	$K_d[12] = E_d[4] \oplus RC_4$ $K_d[13] = ROL(E_d[5], 8)$ $K_d[14] = E_d[6] \oplus RC_4$ $K_d[15] = ROL(E_d[7], 16)$
$K_d[16] = ROL(E_d[0], 24) \oplus RC_5$ $K_d[17] = ROL(E_d[1], 16) \oplus RC_4$ $K_d[18] = ROL(E_d[2], 8) \oplus RC_5$ $K_d[19] = ROL(E_d[3], 24) \oplus RC_4$	$K_d[20] = ROL(E_d[4], 16) \oplus RC_4$ $K_d[21] = ROL(E_d[5], 8) \oplus RC_3$ $K_d[22] = ROL(E_d[6], 24) \oplus RC_4$ $K_d[23] = ROL(E_d[7], 16) \oplus RC_3$
$K_d[24] = ROL(E_d[0], 24) \oplus RC_{35}$ $K_d[25] = ROL(E_d[1], 24) \oplus RC_4$ $K_d[26] = ROL(E_d[2], 8) \oplus RC_{35}$ $K_d[27] = ROL(E_d[3], 8) \oplus RC_4$	$K_d[28] = ROL(E_d[4], 16) \oplus RC_{24}$ $K_d[29] = E_d[5] \oplus RC_3$ $K_d[30] = ROL(E_d[6], 24) \oplus RC_{24}$ $K_d[31] = ROL(E_d[7], 24) \oplus RC_3$
$K_d[32] = ROL(E_d[0], 8) \oplus RC_{35}$ $K_d[33] = ROL(E_d[1], 24) \oplus RC_{24}$ $K_d[34] = E_d[2] \oplus RC_{35}$ $K_d[35] = ROL(E_d[3], 8) \oplus RC_{24}$	$K_d[36] = ROL(E_d[4], 24) \oplus RC_{24}$ $K_d[37] = E_d[5] \oplus RC_{13}$ $K_d[38] = ROL(E_d[6], 8) \oplus RC_{24}$ $K_d[39] = ROL(E_d[7], 24) \oplus RC_{13}$
$K_d[40] = ROL(E_d[0], 8) \oplus RC_{15}$ $K_d[41] = ROL(E_d[1], 16) \oplus RC_{24}$ $K_d[42] = E_d[2] \oplus RC_{15}$ $K_d[43] = ROL(E_d[3], 16) \oplus RC_{24}$	$K_d[44] = ROL(E_d[4], 24) \oplus RC_{04}$ $K_d[45] = ROL(E_d[5], 16) \oplus RC_{13}$ $K_d[46] = ROL(E_d[6], 8) \oplus RC_{04}$ $K_d[47] = ROL(E_d[7], 16) \oplus RC_{13}$
$K_d[48] = ROL(E_d[0], 16) \oplus RC_{15}$ $K_d[49] = ROL(E_d[1], 16) \oplus RC_{04}$ $K_d[50] = ROL(E_d[2], 16) \oplus RC_{15}$ $K_d[51] = ROL(E_d[3], 16) \oplus RC_{04}$	

Notes :

$RC_{13} = RC_1 \oplus RC_3$, $RC_{24} = RC_2 \oplus RC_4$,
$RC_{35} = RC_3 \oplus RC_5$, $RC_{04} = RC_0 \oplus RC_2 \oplus RC_4$,
$RC_{15} = RC_1 \oplus RC_3 \oplus RC_5$.

Table 12: Decryption round keys in terms of expanded keys

inputs				outputs				inputs				outputs			
0	0	0	1	1	1	1	0	0	0	1	0	0	1	1	1
0	0	0	2	2	2	2	0	0	0	2	0	0	2	2	2
0	0	0	3	3	3	3	0	0	0	3	0	0	3	3	3
0	0	0	4	4	4	0	4	0	0	4	0	4	4	4	0
0	0	0	8	8	8	0	8	0	0	8	0	8	8	8	0
0	0	0	c	c	c	0	c	0	0	c	0	c	c	c	0
0	0	0	10	10	0	10	10	0	0	10	0	10	10	0	10
0	0	0	20	20	0	20	20	0	0	20	0	20	20	0	20
0	0	0	30	30	0	30	30	0	0	30	0	30	30	0	30
0	0	0	40	0	40	40	40	0	0	40	0	40	0	40	40
0	0	0	80	0	80	80	80	0	0	80	0	80	0	80	80
0	0	0	c0	0	c0	c0	c0	0	0	c0	0	c0	0	c0	c0
0	1	0	0	1	0	1	1	1	0	0	0	1	1	0	1
0	2	0	0	2	0	2	2	2	0	0	0	2	2	0	2
0	3	0	0	3	0	3	3	3	0	0	0	3	3	0	3
0	4	0	0	0	4	4	4	4	0	0	0	4	0	4	4
0	8	0	0	0	8	8	8	8	0	0	0	8	0	8	8
0	c	0	0	0	c	c	c	c	0	0	0	c	0	c	c
0	10	0	0	10	10	10	0	10	0	0	0	0	10	10	10
0	20	0	0	20	20	20	0	20	0	0	0	0	20	20	20
0	30	0	0	30	30	30	0	30	0	0	0	0	30	30	30
0	40	0	0	40	40	0	40	40	0	0	0	40	40	40	0
0	80	0	0	80	80	0	80	80	0	0	0	80	80	80	0
0	c0	0	0	c0	c0	0	c0	c0	0	0	0	c0	c0	c0	0

Table 13: 1-to-3 / 3-to-1 propagations by π_0

inputs				outputs				inputs				outputs			
0	0	1	1	1	0	0	1	0	1	1	0	1	1	0	0
0	0	2	2	2	0	0	2	0	2	2	0	2	2	0	0
0	0	3	3	3	0	0	3	0	3	3	0	3	3	0	0
0	0	4	4	0	0	4	4	0	4	4	0	4	0	0	4
0	0	8	8	0	0	8	8	0	8	8	0	8	0	0	8
0	0	c	c	0	0	c	c	0	c	c	0	c	0	0	c
0	0	10	10	0	10	10	0	0	10	10	0	0	0	10	10
0	0	20	20	0	20	20	0	0	20	20	0	0	0	20	20
0	0	30	30	0	30	30	0	0	30	30	0	0	0	30	30
0	0	40	40	40	40	0	0	0	40	40	0	0	40	40	0
0	0	80	80	80	80	0	0	0	80	80	0	0	80	80	0
0	0	c0	c0	c0	c0	0	0	0	c0	c0	0	0	c0	c0	0
1	1	0	0	0	1	1	0	1	0	0	1	0	0	1	1
2	2	0	0	0	2	2	0	2	0	0	2	0	0	2	2
3	3	0	0	0	3	3	0	3	0	0	3	0	0	3	3
4	4	0	0	4	4	0	0	4	0	0	4	0	4	4	0
8	8	0	0	8	8	0	0	8	0	0	8	0	8	8	0
c	c	0	0	c	c	0	0	c	0	0	c	0	c	c	0
10	10	0	0	10	0	0	10	10	0	0	10	10	10	0	0
20	20	0	0	20	0	0	20	20	0	0	20	20	20	0	0
30	30	0	0	30	0	0	30	30	0	0	30	30	30	0	0
40	40	0	0	0	0	40	40	40	0	0	40	40	0	0	40
80	80	0	0	0	0	80	80	80	0	0	80	80	0	0	80
c0	c0	0	0	0	0	c0	c0	c0	0	0	c0	c0	0	0	c0

Table 14: 2-to-2 propagations by π_0: consecutive nonzero bytes

inputs				outputs				inputs				outputs			
0	1	0	1	0	1	0	1	1	0	1	0	1	0	1	0
0	2	0	2	0	2	0	2	2	0	2	0	2	0	2	0
0	3	0	3	0	3	0	3	3	0	3	0	3	0	3	0
0	4	0	4	4	0	4	0	4	0	4	0	0	4	0	4
0	8	0	8	8	0	8	0	8	0	8	0	0	8	0	8
0	c	0	c	c	0	c	0	c	0	c	0	0	c	0	c
0	10	0	10	0	10	0	10	10	0	10	0	10	0	10	0
0	11	0	11	0	11	0	11	11	0	11	0	11	0	11	0
0	12	0	12	0	12	0	12	12	0	12	0	12	0	12	0
0	13	0	13	0	13	0	13	13	0	13	0	13	0	13	0
0	20	0	20	0	20	0	20	20	0	20	0	20	0	20	0
0	21	0	21	0	21	0	21	21	0	21	0	21	0	21	0
0	22	0	22	0	22	0	22	22	0	22	0	22	0	22	0
0	23	0	23	0	23	0	23	23	0	23	0	23	0	23	0
0	30	0	30	0	30	0	30	30	0	30	0	30	0	30	0
0	31	0	31	0	31	0	31	31	0	31	0	31	0	31	0
0	32	0	32	0	32	0	32	32	0	32	0	32	0	32	0
0	33	0	33	0	33	0	33	33	0	33	0	33	0	33	0
0	40	0	40	40	0	40	0	40	0	40	0	0	40	0	40
0	44	0	44	44	0	44	0	44	0	44	0	0	44	0	44
0	48	0	48	48	0	48	0	48	0	48	0	0	48	0	48
0	4c	0	4c	4c	0	4c	0	4c	0	4c	0	0	4c	0	4c
0	80	0	80	80	0	80	0	80	0	80	0	0	80	0	80
0	84	0	84	84	0	84	0	84	0	84	0	0	84	0	84
0	88	0	88	88	0	88	0	88	0	88	0	0	88	0	88
0	8c	0	8c	8c	0	8c	0	8c	0	8c	0	0	8c	0	8c
0	c0	0	c0	c0	0	c0	0	c0	0	c0	0	0	c0	0	c0
0	c4	0	c4	c4	0	c4	0	c4	0	c4	0	0	c4	0	c4
0	c8	0	c8	c8	0	c8	0	c8	0	c8	0	0	c8	0	c8
0	cc	0	cc	cc	0	cc	0	cc	0	cc	0	0	cc	0	cc

Table 15: 2-to-2 propagations by π_0: separate nonzero bytes

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

Table 16: The S-box S_0

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

Table 17: The S-box S_1