

---

From: maro@isl.ntt.co.jp  
To: AESFirstRound@nist.gov  
Subject: Optimized Software Implementations of E2  
Date: Fri, 16 Apr 1999 00:03:55 JST  
Sender: maro@sucaba.isl.ntt.co.jp

Dear AES Candidate Comments Secretariat,

I submit the report titled  
Optimized Software Implementations of E2  
using PostScript. The file is generated with gzip and  
uuencode. If you cannot print the file, I will help you.

Best regards,

/ NTT Laboratories  
/ AOKI, Kazumaro  
/ E-mail: maro@isl.ntt.co.jp

# Optimized Software Implementations of $E2$ <sup>\*\*\*</sup>

Kazumaro AOKI<sup>\*\*\*</sup> and Hiroki UEDA<sup>†</sup>

NTT Laboratories<sup>‡</sup>

**Abstract.** This paper describes many techniques for optimizing software implementations of  $E2$  on various platforms. It is relatively easy to implement a byte-oriented cipher such as  $E2$  on an 8-bit processor, but it is difficult to implement it efficiently on a 32-bit processor or a 64-bit processor. In particular, this paper shows several optimization techniques for SPN (Substitution-Permutation Network) on 32- or 64-bit processors. They are also applicable to other byte-oriented ciphers. As a result,  $E2$  achieves the encryption speeds of 100.5kb/s, 68.3Mb/s, 162.3Mb/s, and 130.8Mb/s for H8/300 (5MHz), Pentium Pro (200MHz), Pentium II (450MHz), and 21164A (600MHz).

*Keywords.*  $E2$ , SPN, optimization, 32-bit processor, 64-bit processor, inverse

## 1 Introduction

NTT submitted  $E2$  [N98a, N98b] as an AES candidate in response to the call issued by NIST in 1997 [U97].  $E2$  is a byte-oriented<sup>6</sup> cipher, and was designed to be fast on 8-bit processors as well as 32-bit processors, which are current standards, and 64-bit processors, which are considered to be the next generation standard. Since  $E2$  is byte-oriented, it is not obvious how to implement  $E2$  efficiently on 32- or 64-bit processors.

This paper describes some techniques for optimizing software implementations of  $E2$  on such processors. Optimization techniques are introduced for each part of  $E2$ . In particular, the optimization techniques for SPN (Substitution-Permutation Network) on a 32- or 64-bit processor are applicable to other byte-oriented ciphers as well.

## 2 Specification of $E2$

Figure 1 shows the outline of the  $E2$  encryption process.  $E2$  has a 12-round Feistel structure with a preprocess,  $IT$ -Function, and a postprocess,  $FT$ -Function.

---

\* Manuscript revised March 31, 1999

\*\* Manuscript revised April 15, 1999

\*\*\* Email: maro@isl.ntt.co.jp

† Email: ueda@isl.ntt.co.jp

‡ 1-1 Hikarinooka, Yokosuka-shi, Kanagawa-ken, 239-0847 Japan

<sup>6</sup> In this paper 1 *byte* is defined as 8 bits.

The decryption process is the same as the encryption process except for the order of the subkeys. Figure 2 outlines  $F$ -Function.  $F$ -Function consists of  $S$ -Functions,  $P$ -Function, and  $BRL$ -Function.

Refer to [N98a] for details of the specification and notations.

### 3 Optimization of Each Part of $E2$

Several optimization techniques were shown in [N98b]. However, this paper shows all known techniques including those described in [N98b].

#### 3.1 Setup

##### 3.1.1 $f(v_{-1})$

In the  $E2$  key scheduling part,  $G$ -Function shown in Fig. 4 is computed 9 times. In the first computation of  $G$ -Function,  $f(v_{-1})$  can be calculated in the setup stage, since  $U = v_{-1}$  holds and  $v_{-1}$  is a constant defined in the specification.

##### 3.1.2 128- and 192-bit Key

When the key is 128- or 192-bits long,  $E2$  performs the same key scheduling tasks as in the case of the 256-bit key after padding the shorter keys with some constant values. Thus,  $f$ -Function which depends on only constants can be calculated in the setup stage. 18 or 9  $f$ -Functions can be calculated for 128- or 192-bit keys, respectively, in the setup stage.

##### 3.1.3 Inverse

The operation  $\oslash$  in  $FT$ -Function requires an inverse in mod  $2^{32}$ . This depends only on the key, i.e., it does not depend on plaintexts. Thus, the inverse can be calculated in the setup stage.

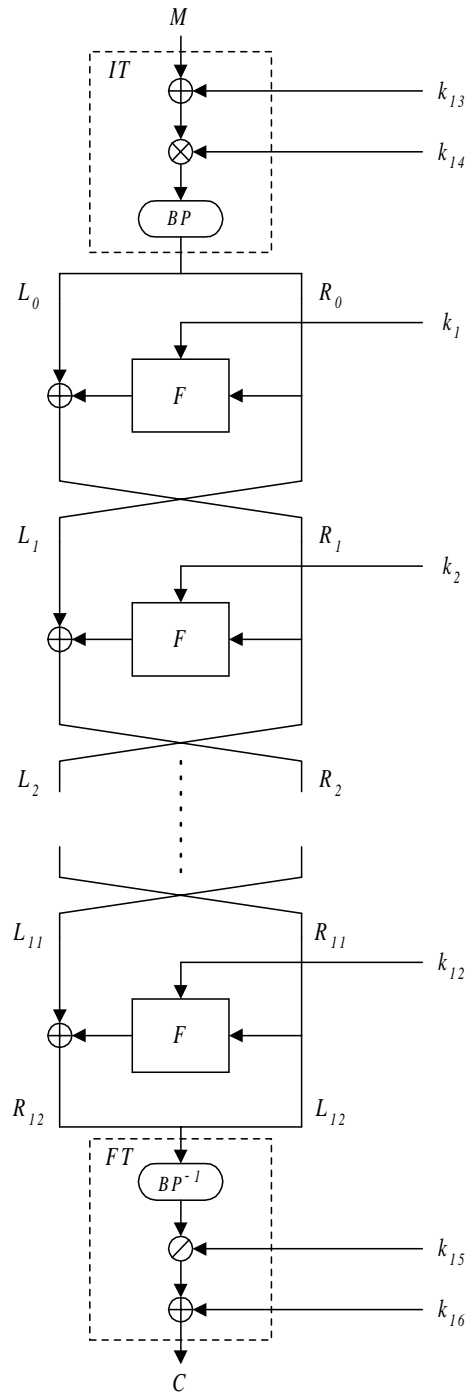
An inverse can be calculated by using the extended Euclidean algorithm. However, the extended binary GCD (ex., in [K97, Algorithm Y in p.646] and in [HKQ99, Figure 1 in p.101(p.7)]: the latter is optimized for mod  $2^n$ ) and Hensel lifting (ex., in [DK91, MODULAR-INVERSE algorithm in pp.235–236]) are more effective on a variety of platforms since the modulus has a special form.

Moreover, the Hensel lifting quadratic version proposed by Zassenhaus [Z69] is quite effective if the platform can use an effective 32-bit multiplier. We used Zassenhaus' algorithm to create Algorithm 1 for calculating inverses. It is useful for general processors whose word lengths are longer than 32 bits.

#### 3.2 Encryption Process

##### 3.2.1 $S$ -Function

$S$ -Function in  $F$ -Function consists of 8  $s$ -boxes whose input and output lengths are 8 bits. Figure 2 shows that 8  $s$ -boxes can be calculated in parallel. Preparing the table  $(x, y) \mapsto (s(x), s(y))$  halves the number of memory references. This



**Fig. 1.** Encryption process

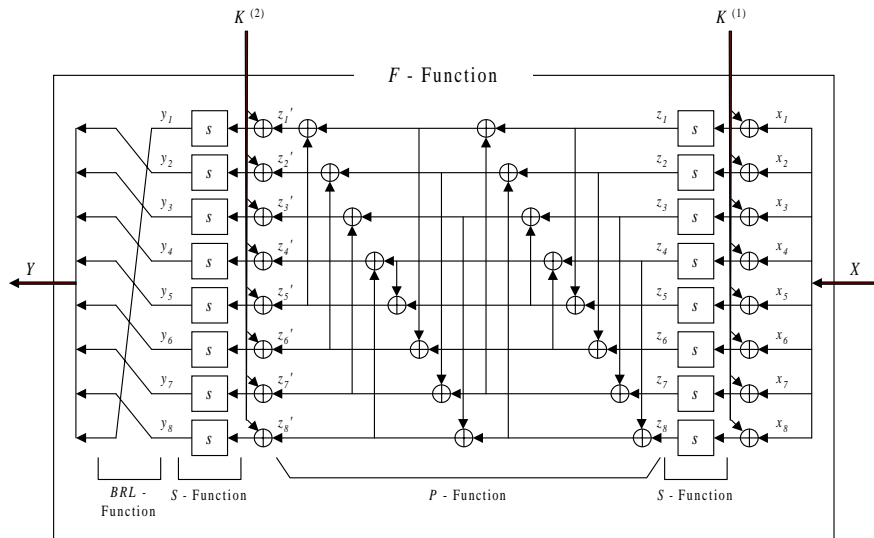


Fig. 2.  $F$ -Function

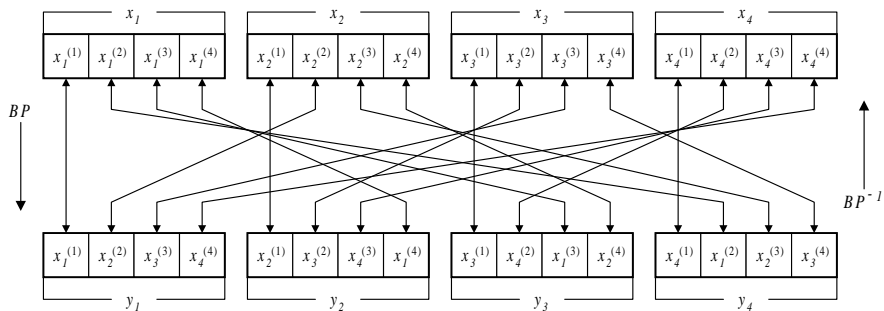


Fig. 3.  $BP$ -Function and  $BP^{-1}$ -Function

technique requires as much as 128KB memory for the table, however, it is effective in the following cases.

1. The table can be stored in fast memory such as the 1st cache.
2. The 1st cache is hard to control such as in Java.

Referring to each  $s$ -box table is preferred if the size of the 1st cache is less than 128KB. Note that recent processors can cause a penalty when data that are not aligned on word boundary are accessed. For example, prepare table

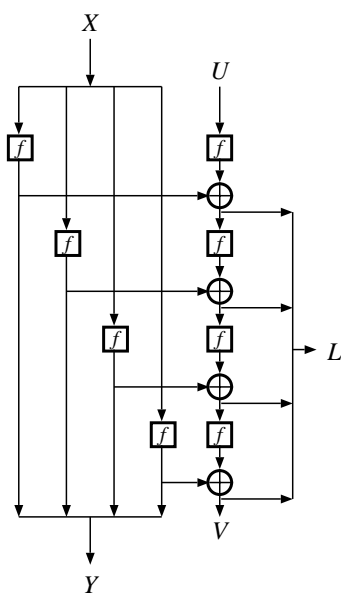


Fig. 4. G-Function

**Algorithm 1.** Calculation of  $y = x^{-1} \bmod 2^{2^n}$ . Let  $a$  and  $b$  be temporary variables, and  $[z]$  be Gauss' symbol (which denotes maximum integer which does not exceed  $z$ ), and the bit lengths of  $x$ ,  $y$ ,  $a$ , and  $b$  be  $2^n$ .

**Step 1:** Input  $x$ . ( $x$  is assumed as an odd integer.)

**Step 2:** Do the initial process as follows.

1.  $b := \lfloor \frac{x}{2} \rfloor$
2.  $a :=$  least significant bit of  $b$
3.  $b := \lfloor \frac{ax + b}{2} \rfloor$
4.  $y :=$  least significant 2 bits of  $x$

**Step 3:** Do the following for  $i = 1, 2, \dots, n - 1$ .

1.  $a := -by$
2.  $b := \lfloor \frac{b + ax}{2^{2^i}} \rfloor$
3.  $y := y + a \times 2^{2^i}$

**Step 4:** Output  $y$ .

$x \mapsto (0, 0, 0, s(x))$  for a 32-bit processor instead of a simple 256 byte  $s$ -box table. Moreover, preparing the tables

$$\begin{aligned}
 x &\mapsto (0, 0, 0, s(x)) \\
 x &\mapsto (0, 0, s(x), 0) \\
 x &\mapsto (0, s(x), 0, 0) \\
 x &\mapsto (s(x), 0, 0, 0)
 \end{aligned} \tag{1}$$

eliminates the data position adjustment processes. However, we should ensure that the size of these tables does not exceed the size of the 1st cache.

### 3.2.2 *BP*-Function

*BP*-Function shown in Fig. 3 changes the order of bytes in *IT*-Function. A 32-bit or a 64-bit processor requires a large number of instructions if *BP*-Function is implemented in a straightforward manner since the number of instructions needed to handle byte operations is very large. Considering the processors requirements, we usually divide the input of *F*-Function into bytes for  $s$ -box input as described in Sect. 3.2.1. Thus, it is not necessary to follow the specification in terms of the byte order of an *F*-Function input, because no additional costs are incurred even if the byte-order is changed. When 16 bytes are divided into 2 eight bytes for input to the Feistel structure, we should efficiently extract 2 sets of 8 bytes which are outputs of *BP*-Function, which are left or right halves defined in the specification, and put them into registers. Since *F*-Function requires byte operations, the transformed *F*-Function which differs only in input byte order is not slower than the original *F*-Function.

To achieve this purpose, if we change the byte order

$$01234567 \ 89ABCDEF \mapsto 05AF49E3 \ 8D27C16B$$

into

$$01234567 \ 89ABCDEF \mapsto 09A345EF \ 812BCD67,$$

then the number of masking operations etc. is reduced to about a half. Note that each letter represents 1 byte.

To get the correct ciphertext as defined in the specification, apply a similar technique to  $BP^{-1}$ -Function in *FT*-Function.

### 3.2.3 *BRL*-Function

*BRL*-Function is at the end of *F*-Function. If *BRL*-Function and *S*-Function are calculated at the same time, no time is required for *BRL*-Function. That is, we should put the output bytes from  $s$ -boxes into the correct positions considering the effect of *BRL*-Function using (1), when bytes are changed to words.

### 3.2.4 P-Function

$P$ -Function, which realizes linear transformation layer in  $F$ -Function, is represented as multiplication using an  $8 \times 8$  matrix. If we consider the operation unit as a byte, the calculation requires 36 XORs, however, if we follow Fig. 2, only 16 XORs are required.

Algorithm 2 requires only 4 cycles if the algorithm is implemented on recent processors which offer pipelining, parallel execution, and 32-bit rotation. The byte order of the output does not match the specification, however, suitable coding may prevent a speed decrease, since each  $s$ -box is processed individually in  $S$ -Function.

<b>Algorithm 2.</b> Calculation of $Z' = P(Z)$ . Let $RL_b(X)$ mean $b$ -byte left rotation of $X$ .				
<b>Step 1:</b> Input $(H, L) = ((z_1, z_2, z_3, z_4), (z_5, z_6, z_7, z_8))$ .				
<b>Step 2:</b> Do the operations as the following order.				
cycle	Operation	order of $H$	order of $L$	
1	$L := H \oplus L$	1234	5678	
1	$H := RL_2(H)$	3412	5678	
2	$H := H \oplus L$	3412	5678	
2	$L := RL_3(L)$	3412	8567	
3	$L := H \oplus L$	3412	8567	
3	$H := RL_1(H)$	4123	8567	
4	$H := H \oplus L$	4123	8567	
<b>Step 3:</b> Output $(H, L) = ((z'_4, z'_1, z'_2, z'_3), (z'_8, z'_5, z'_6, z'_7))$ .				

### 3.2.5 Substitution and Permutation

This section uses the notation

$$sb_1b_2 \cdots b_n : x \mapsto (b_1s(x), b_2s(x), \dots, b_ns(x)),$$

where  $b_i \in \{0, 1\}$ . For example,  $s0010$  means  $x \mapsto (0, 0, s(x), 0)$ .

The substitution and the permutation in  $F$ -Function of **E2** is represented as

$${}^T[z'_1 \ z'_2 \ \cdots \ z'_8] = P^T[s(x'_1) \ s(x'_2) \ \cdots \ s(x'_8)]$$

using the matrix

$$P = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix},$$



where  $x'_i$  is the XORed value of  $x_i$  and  $K^{(1)}$  in Fig. 2, and the superscript  $T$  means matrix transposition.

Rijmen et al. proposed an effective implementation of the substitution and permutation in SHARK [RDP<sup>+</sup>96]. This section studies the implementation of substitution and permutation for 64- and 32-bit processors based on the implementation of SHARK.

64-bit processor Using the implementation technique of SHARK directly means that tables

$s01111101, s10111110, s11010111, s11101011,$   
 $s10111001, s11011100, s11100110, s01110011$

are required. The computation cost of this technique is summarized as follows.

Required memory	16KB
Number of table references	8
Number of XORs	7

When the size of the 1st cache is less than 16KB, 8 tables described above may be generated from just  $s11111111$  using masks. This case is summarized as follows.

Required memory	2KB
Number of table references	8
Number of masks	8
Number of XORs	7

32-bit processor The previous section describing the implementation for 64-bit processors only discussed the implementation of  $P(S(\cdot))$ , since no effective implementation of  $BRL(S(\cdot))$  has been found. This section considers the memory required for implementing the 2nd non-linear layer (substitution) in  $F$ -Function on a 32-bit processor, which causes good results.

Suppose that tables

$s1000, s0100, s0010, s0001$

as described in Sect. 3.2.1 are prepared for implementing the 2nd non-linear layer. They occupy a total of 4KB. Note that for time complexity we consider only  $P(S(\cdot))$ ; we do not consider the 2nd non-linear layer.

Following the SHARK implementation technique directly, similarly to the case of 64-bit processors, tables

$s0111, s1011, s1101, s1110,$   
 $s0011, s1001, s1100, s0110,$   
 $s1000, s0100, s0010, s0001$

are required for 32-bit processors. This case is summarized as follows.

Required memory	12KB
Number of table references	16
Number of XORs	14

If 4 bytes are stored in a 32-bit register in any order, the speed is the same, since the implemented process unit is a byte as described in Sect. 3.2.1. For example, changing the order of calculation as follows:

$$\begin{bmatrix} z'_1 \\ z'_8 \\ z'_5 \\ z'_4 \\ z'_7 \\ z'_2 \\ z'_3 \\ z'_6 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 1 & | & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & | & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & | & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & | & 1 & 1 & 0 & 1 \\ \hline 0 & 1 & 1 & 1 & | & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & | & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & | & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & | & 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \\ z_8 \end{bmatrix},$$

means that tables

$s0111, s1011, s1101, s1110, s0101, s1111,$   
 $s1000, s0100, s0010, s0001$

are required, and memory references and XOR operations of high and low words corresponding to  $z_1, z_2, z_3, z_4$  are the same. This improved case is summarized as follows.

Required memory	10KB
Number of table references	12
Number of XORs	11

Consider the case that required memory exceeds the size of the cache or the case that the latency<sup>7</sup> of memory references is problematic. For example, if we change the order of calculation to

$$\begin{bmatrix} z'_1 \\ z'_2 \\ z'_4 \\ z'_3 \\ z'_7 \\ z'_8 \\ z'_6 \\ z'_5 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 1 & | & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & | & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & | & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & | & 1 & 0 & 1 & 1 \\ \hline 0 & 1 & 1 & 1 & | & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & | & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & | & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & | & 1 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \\ z_8 \end{bmatrix}$$

and prepare tables

$s0111, s1011, s1101, s1110,$   
 $s1000, s0100, s0010, s0001,$

<sup>7</sup> Cycles after issuing an instruction before being able to access the result.

memory references corresponding to  $z_5, z_6, z_7, z_8$  are directly used for  $z'_1, z'_2, z'_4, z'_3$ , and  $z'_7, z'_8, z'_6, z'_5$  are calculated as right 1 byte, right 2 bytes, left 1 byte, and left 2 bytes logical shifted from  $z'_1, z'_2, z'_4, z'_3$ , respectively. This case is summarized as follows.

Required memory	8KB
Number of table references	8
Number of XORs	11
Number of shifts	4

$P$ -Function of  $E2$  has an interesting property. First, we change the order of the calculations to

$$\begin{bmatrix} z'_1 \\ z'_2 \\ z'_3 \\ z'_4 \\ z'_7 \\ z'_8 \\ z'_5 \\ z'_6 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 1 & 1 & | & 1 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & | & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & | & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & | & 1 & 1 & 0 & 1 \\ \hline 0 & 1 & 1 & 1 & | & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & | & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 & | & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & | & 0 & 1 & 1 & 0 \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \\ z_8 \end{bmatrix}.$$

We focus on the top right and bottom right submatrices:

$$Q_U = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{bmatrix}, \quad Q_D = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

Letting  $SD_b(X)$  be  $b$ -byte down-shift of  $X$ , and  $SU_b(X)$  be  $b$ -byte up-shift of  $X$  yields

$$\begin{aligned} Q_U \oplus Q_D &= \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \\ &= SD_1(Q_U) \oplus SU_3(Q_U). \end{aligned}$$

This means that we can calculate  $P(S(\cdot))$  using tables

$$\begin{aligned} &s0111, s1011, s1101, s1110, \\ &s1000, s0100, s0010, s0001. \end{aligned}$$

This case is summarized as follows.

Required memory	8KB
Number of table references	8
Number of XORs	9
Number of shifts	2

If the rotation is available, we can reduce the computation cost to

Required memory	8KB
Number of table references	8
Number of XORs	8
Number of rotations	1

## 4 Implementation Results

We optimized *E2* implementations for several processors. Table 1 shows the results for the key scheduling part, and Table 2 shows the results for data randomizing part.

To achieve high performance on recent processors, it is important to consider instruction scheduling as well as decreasing the number of instructions. We achieved 2.45 [ $\mu$ ops/cycle] parallel execution on a Pentium II and 1.73 [instructions/cycle] parallel execution on an Alpha processor (average values) using the implementations described in Table 2. We think that these implementations realize parallel execution efficiently.

**Table 1.** Key Scheduling Part

Processor	Key length (bits)	Speed (cycles/key)
Pentium Pro <sup>a</sup>	128	1868
	192	2031
	256	2294
Pentium II <sup>b</sup>	128	1804
	192	1991
	256	2228
H8/300 <sup>c</sup>	128	14041
	192	15284
	256	16518

<sup>a</sup> IBM PC/AT compatible, Pentium Pro(200MHz), 64MB RAM, MS-Windows95, Microsoft Visual C++ 5.0 Enterprise Edition

<sup>b</sup> IBM PC/AT compatible, Pentium II(450MHz), 256MB RAM, MS-Windows95, Microsoft Visual C++ 5.0 Enterprise Edition

<sup>c</sup> H8/300(5MHz) emulator on FreeBSD, assembly

## 5 Conclusion

We analyzed each part of *E2* and studied how to implement them efficiently on various platforms. As a result, we achieved faster implementation on 32-bit processors, which are the current standard, and a 64-bit processor, which is

**Table 2.** Data Randomizing Part

Processor	Speed	
	(cycles/block)	(bits/second)
Pentium Pro <sup>a</sup>	375	68.3M
Pentium II <sup>b</sup>	355	162.3M
Java VM <sup>c</sup>	2370	10.8M
Java VM <sup>d</sup>	28800	0.9M
Alpha <sup>e</sup>	587	130.8M
H8/300 <sup>f</sup>	6374	100.5k

<sup>a</sup> IBM PC/AT compatible, Pentium Pro(200MHz), 64MB RAM, assembly

<sup>b</sup> IBM PC/AT compatible, Pentium II(450MHz), 256MB RAM, assembly

<sup>c</sup> IBM PC/AT compatible, Pentium Pro(200MHz), 64MB RAM, JDK 1.1.6 with JIT

<sup>d</sup> IBM PC/AT compatible, Pentium Pro(200MHz), 64MB RAM, JDK 1.1.6 without JIT

<sup>e</sup> Alpha AXP 21164A (600MHz), 8MB 3rd cache, 256MB RAM, Digital Unix 4.0, assembly

<sup>f</sup> H8/300(5MHz) emulator on FreeBSD, assembly

considered to be the next generation standard, even though **E2** is a byte-oriented cipher.

NTT will continue to optimize **E2** implementation. The latest implementation results are available at <http://info.isl.ntt.co.jp/e2/>.

## References

- [DK91] S. R. Dussé and B. S. Kaliski Jr. A Cryptographic Library for the Motorola DSP56000. In I. B. Damgård, editor, *Advances in Cryptology — EURO-CRYPT'90*, Volume 473 of *Lecture Notes in Computer Science*, pp. 230–244. Springer-Verlag, Berlin, Heidelberg, New York, 1991.
- [HKQ99] G. Hachez, F. Koeune, and J.-J. Quisquater. cAESar results: Implementation of Four AES Candidates on Two Smart Cards. In *Second Advanced Encryption Standard Candidate Conference*, pp. 95–108, Hotel Quirinale, Rome, Italy, 1999. Information Technology Laboratory, National Institute of Standards and Technology.
- [K97] D. E. Knuth. *Seminumerical Algorithms*, Volume 2 of *The Art of Computer Programming*. Addison Wesley, third edition, 1997.
- [N98a] Nippon Telegraph and Telephone Corporation. *Specification of E2 — a 128-bit Block Cipher*, 1998. (<http://info.isl.ntt.co.jp/e2/>).
- [N98b] Nippon Telegraph and Telephone Corporation. *Supporting Document on E2*, 1998. (<http://info.isl.ntt.co.jp/e2/>).
- [RDP<sup>+</sup>96] V. Rijmen, J. Daemen, B. Preneel, A. Bosselaers, and E. De Win. The Cipher SHARK. In D. Gollmann, editor, *Fast Software Encryption — Third*

- International Workshop*, Volume 1039 of *Lecture Notes in Computer Science*, pp. 99–111. Springer-Verlag, Berlin, Heidelberg, New York, 1996.
- [U97] U.S. Department of Commerce, National Institute of Standards and Technology. *Announcing Request for Candidate Algorithm Nominations for the Advanced Encryption Standard (AES)*, 1997. Federal Register: Volume 62, Number 177, pp.48051–48058, (<http://csrc.nist.gov/encryption/aes/aes-9709.htm>).
- [Z69] H. Zassenhaus. On Hensel Factorization, I. *Journal of number theory*, Vol. 1, pp. 291–311, 1969.

## Appendix: Implementation for Low Memory Environment

**E2** can be implemented even in low memory environments such as low-end smart cards where not all subkeys can be stored in RAM. In this case, each subkey should be generated *on-the-fly* by calling the key scheduling part several times during the data randomizing.

First, we estimate the required RAM size by the implementation for 128-bit key. The minimum required RAM size is 56 bytes: 16 bytes for a plaintext or ciphertext we call ‘PC area,’ 16 bytes for a master key we call ‘M area,’ and 24 bytes for subkey generation we call ‘SG area.’ The PC area is also used for storing the intermediate data in data randomizing part, and the SG area is also used for storing the working data required for computing  $F$ -,  $IT$ -, and  $FT$ -Functions<sup>8</sup>.

Second, we estimate how many times key scheduling calls are required. Let  $A$  be the available RAM size. Because **E2** requires sixteen 16 bytes for subkeys generation, the implementation requires  $\left\lceil 16 / \left\lfloor \frac{A - 56}{16} \right\rfloor \right\rceil$  key scheduling calls. We summarize the results in Table 3.

**Table 3.** Number of key scheduling calls required for encryption

Available RAM (bytes)	$\geq 72$	$\geq 88$	$\geq 104$	$\geq 120$	$\geq 152$	$\geq 184$	$\geq 312$
# of calls	16	8	6	4	3	2	1

This article was processed using the  $\LaTeX$  macro package with LLNCS style

<sup>8</sup> Moreover, because some intermediate values required by key scheduling depend on constant values, the intermediate values can be computed before coding as stated in Sect. 3.1.2. These 144 bytes can be stored in ROM.