Analyzing performance on AES implementations

Neng-Wen Wang
Department of Engineering Science
National Cheng Kung University
Taiwan,R.O.C.
nwwang@cc.kyit.edu.tw

Yueh-Min Huang
Department of Engineering Science
National Cheng Kung University
Taiwan, R.O.C.

Chi-Sung Laih
Department of Electrical Engineering
National Cheng Kung University
Taiwan, R.O.C.

Abstract

NIST(National Institute of standards and Technology) announced that Rijndael was selected as the proposed AES(Advanced Encryption Standard)[2] on Oct 2, 2000. Following the year, NIST approved the AES as a standard enumerated by FIPS 197(Federal Information Processing Standards Publication 197) [1] on Nov 26,2001. Several Versions of programs was also proposed for AES implementation [3]. However, there exists difference of more than hundred times in the efficiency of encryption/decryption among these versions. In this paper, we evaluate efficiency for these AES algorithms. We also elaborately analyzed these algorithms in their technical skills, especially for the latest 32-bit algorithm. We will describe some special methodology to expedite on the encryption and decryption. The result shows that the latest algorithm would largely promote the encryption/ decryption efficiency. To take advantage of the latest version, some restrictions should be taken in the hardware resource and code implementation. However, there is no clear explanation in the original proposed AES [2]. There are some ambiguous paragraphs that may confuse programmers. Thus, we suggested NIST to modify those paragraphs in [4]. The original AES proposal has been amended as a final official FIPS197 standard which had added some sections, including the ambiguous paragraphs.

Keywords: *AES(Advanced Encryption Standard), FIPS 197(Federal Information Processing Standards Publication 197)*

1. Introduction

AES has been chosen as the next generation encryption standard by NIST. It will replace the use of DES in the following 30 years. The original DEA (DES Algorithm) was designed for mid-1970s hardware implementation and does not produce efficient software code. TDEA (Triple DEA), which has three times as many rounds as DEA, is correspondingly slower. Both of them use a

64-bit block size encryption/decryption. For reasons of both efficiency and security, a large block size is desirable. Because of these drawbacks, TDEA is not a reasonable candidate for long term use[5]. In 1997, NIST issued a call for proposals for a final advanced encryption standard (AES), which should have a security strength equal to or better than TDEA and significantly improved efficiency. AES must be a symmetric block either with a block length of 128 bits and support for key lengths of 128,192, 256 bits. On Oct. 2,2000, Rijndael was finally chosen as the proposed AES. The AES standard was completed on the next year. NIST published the AES standard as FIPS197 on Nov 26,2001. The major concerns for AES are the efficiency and the security. There are theoretical proofs of the security issue in relative documents. However, there are few mentioned about the efficiency issues. In this paper, we evaluate these implementation algorithms. We dedicate on this efficiency issue. Currently, Computers are well designed in the hardware architecture. To achieve a better efficiency, one should design an algorithm that can utilize CPU power as high as possible. The 32-bit algorithm can fully utilize CPU power. We will describe these issues in detail on the following sections. The definition of the state and cyber key arrays in Rijndael Algorithm is described in Section 2. Some major amended sections in the final AES standard (FIPS197) are described in Section 3. We analyze AES Algorithms and elaborate its implementation issues in section 4. The performance is evaluated in section 5. Conclusion is finally summarized in section 6.

2. The state and the cipher key arrays in AES Algorithm

The Rijndael cipher is suited to be implemented efficiently on a wide range of processors and in dedicated hardware. We will define the terminology in following paragraph [2], then describe the Rijndael Algorithm.

2.1 The state and the cipher key [2]

The different transformations operated on the intermediate cipher result are called the state. The state can be pictured as a rectangular array of bytes. This array has four rows, the number of columns is denoted by Nb and is equal to the block length divided by 32.

The cipher key is similarly pictured as a rectangular array with four rows. The number of columns of the cipher key is denoted by Nk which is equal to the key length divided by 32. These representations are illustrated in Fig 1.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$	$k_{0,4}$	$k_{0,5}$	$k_{0,6}$	$k_{0,7}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$	$k_{1,4}$	$k_{1,5}$	$k_{1,6}$	$k_{1,7}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$	$k_{2,4}$	$k_{2,5}$	$k_{2,6}$	$k_{2,7}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$	$k_{3,4}$	$k_{3,5}$	$k_{3,6}$	$k_{3,7}$

Fig 1 Examples of state with Nb=6 and key with Nk=8

The input and output used by Rijndael at its external interface are considered to be one-dimensional arrays of 8-bit bytes numbered upwards from 0 to the 4*Nb-1. These blocks hence have lengths of 16,24 0r 32 bytes and array indices in the ranges 0..15, 0..23 or 0..31. The cipher key is considered to be one-dimensional arrays of 8-bit bytes numbered upwards from 0 to the 4*Nk-1. These blocks hence have lengths of 16,24 0r 32 bytes and array indices in the ranges 0..15, 0..23 or0..31.

The cipher input bytes are mapped onto the state bytes in the order $a_{0,0}$, $a_{1,0}$, $a_{2,0}$, $a_{3,0}$, $a_{0,1}$,$a_{1,1}$,$a_{2,1}$,$a_{3,1}$, $a_{0,2}$,$a_{1,2}$,$a_{2,2}$,$a_{3,2}$, …, and the cipher key input bytes are mapped onto the state bytes in the order $k_{0,0}$,$k_{1,0}$,$k_{2,0}$,$k_{3,0}$, $k_{0,1}$,$k_{1,1}$,$k_{2,1}$,$k_{3,1}$, $k_{0,2}$,$k_{1,2}$,$k_{2,2}$,$k_{3,2}$, …. At the end of the cipher operation, the cipher output is extracted from the state by taking the state bytes in the same order.

2.2 Using Transpose Matrix to enhance the efficiency of data access

The Rijdael algorithm use matrices to store the state and the cipher key. All the encryption/decryption operation will perform on these two matrices. During the implementation, if we use the original matrix orientation as Fig 1, sequential words accesses may become slower. To enhance the efficiency in data access, we would like to transpose the matrix while implementing it as Fig 2. The reason will be explained in the next paragraph.

$a_{0,0}$	$a_{1,0}$	$a_{2,0}$	$a_{3,0}$
$a_{0,1}$	$a_{1,1}$	$a_{2,1}$	$a_{3,1}$
$a_{0,2}$	$a_{1,2}$	$a_{2,2}$	$a_{3,2}$
$a_{0,3}$	$a_{1,3}$	$a_{2,3}$	$a_{3,3}$
$a_{0,4}$	$a_{1,4}$	$a_{2,4}$	$a_{3,4}$
$a_{0,5}$	$a_{1,5}$	$a_{2,5}$	$a_{3,5}$

Fig. 2 Transpose matrix

For most of the currently high-level programming language, the alignment of array elements is row major. That is, array elements are placed in the memory with a sequential order of row by row. If the CPU manipulates data in another fashion of column by column, the column's elements (such as a00, a10, a20, a30) are not in the consecutive location. The access time for nonconsecutive data is always much slower than the consecutive data. Fig 3 shows the architecture of classical DRAM[7]. To access data in a location, the row address should be decoded before the column address. Fig 4 and Fig. 5 show the difference of time required between consecutive-byte access and nonconsecutive one [6][7]. For consecutive-byte access, the row address is decoded only one time, which followed by four- column address. Nevertheless, the row address should be decoded for each nonconsecutive-byte. Because of this drawback, the 32-bit algorithm will arrange data access in fashion of row by row.

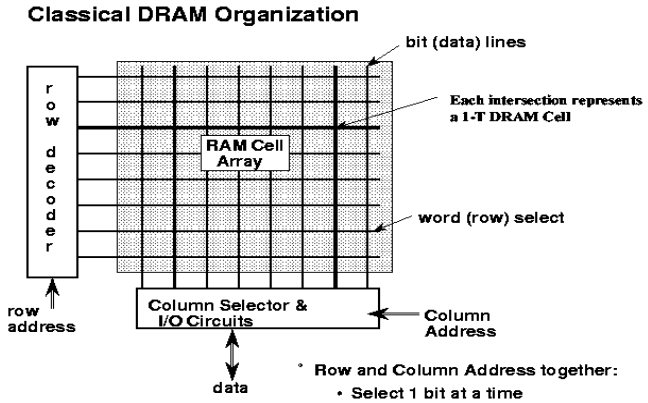

Fig 3 Classical DRAM architecture

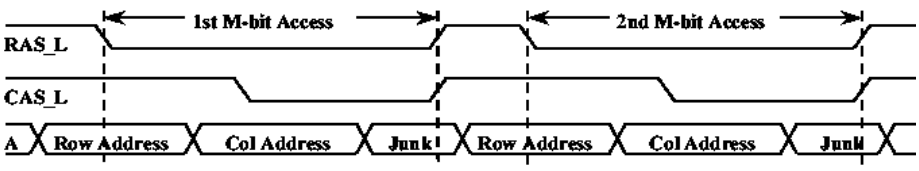

Fig. 4 Nonconsecutive bytes access

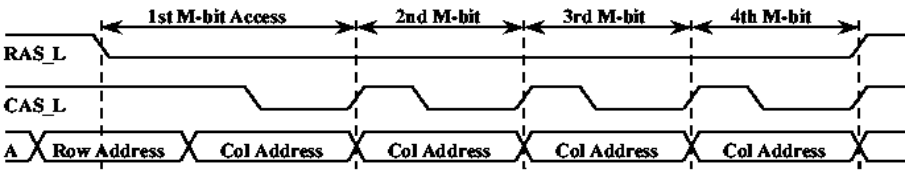

Fig. 5 Consecutive bytes access

2.3 Some ambiguity in the AES proposal and our suggestion for AES standard

In the section 4.1 of AES proposal, there are some ambiguous descriptions for the state matrix and key matrix. First, we list these paragraphs as follows:

*The input and output used by Rijndael at its external interface are considered to be one-dimensional arrays of 8-bit bytes numbered upwards from 0 to the 4*Nb-1. These blocks hence have lengths of 16,24 0r 32 bytes and array indices in the ranges 0..15, 0..23 or0..31. The cipher key is considered to be one-dimensional arrays of 8-bit bytes numbered upwards from 0 to the 4*Nk-1. These blocks hence have lengths of 16,24 0r 32 bytes and array indices in the ranges 0..15, 0..23 or0..31.*

The cipher input bytes are mapped onto the state bytes in the order a0,0, a1,0, a2,0, a3,0, a0,1,a1,1,a2,1,a3,1, …, and The cipher key input bytes are mapped onto the state bytes in the order k0,0,k1,0,k2,0,k3,0, k0,1,k1,1,k2,1,k3,1, …. At the end of the cipher operation, the cipher output is extracted from the state by taking the state bytes in the same order.

However, there is inconsistent description in another paragraph as follows:

In some instances, these blocks are also considered as one-dimensional arrays of 4-byte vectors, where each vector consists of the corresponding column in the rectangular array representation. These arrays hence have lengths of 4,6 or 8 respectively and indices in the ranges 0..3,0..5 or 0..7. 4-byte vectors will sometimes be referred as words.

If the matrix blocks are considered as one-dimensional arrays of 4-byte (words) vectors, which each vector consists of the corresponding column. These arrays hence have indices in the ranges 0..3, 0..5, 0..7 for 128,192,256 bits block respectively. For C programming language, the column vectors

($[a_{00} \quad a_{10} \quad a_{20} \quad a_{30}]^t$, $[a_{01} \quad a_{11} \quad a_{21} \quad a_{31}]^t$, etc.…) are never allocated in the consecutive location. These vectors can not be directly treated as words. Except that there are other temporary variables or arrays used for storing the transposed column vectors. The transposed vectors are directed from the column vectors. The transposed vectors will be $[a_{00} \quad a_{10} \quad a_{20} \quad a_{30}]$, $[a_{01} \quad a_{11} \quad a_{21} \quad a_{31}]$, and etc…. This additional procedure needs to be handled for the implementation in row-major programming languages such as C. There will be another overhead in transposing vectors and additional variables or arrays. It will be better that the AES standard should provide another technique note to explain this additional procedure for programmers.

Because of this drawback, we suggest that the AES proposal adopt the transposed matrices as Fig 2. In addition to benefitting the efficiency of data access (as explained in section 2.2), it also resolve the ambigious problem. If the transposed matrices are adopted in the AES proposal, no additional transposing procedure is needed.

3. Some modified sections in the final offical AES standard[1]

3.1 Related modifications in the Matrices

If the matrix is transposed, the row vectors and column vectors are actually interchanged.Hence, some modifications should be needed in the Rijndael Algorithm. First, Shiftrow should be modified as Shiftcolumn, Second, Mixcolumn should be modified as Mixrow. After this modification, the Rijndael Algorithm will be clearly understood both by the system designer and the programmer. Unfortunately, the orientations of states and keys are not modified in the final AES standard. The original rightward orientation of matrices which have been used for a long period of time, due to some other reasons of the AES work group, is continuously been used. However, NIST did modify the ambiguity sections in the original AES proposal section 4.1. The new definitions of the matrices are described in Sections 3.3: Arrays of Bytes, Section 3.4: The State and Section 3.5: The state as array of columns.

The sequence of element and the orientation in arrays are much clearly defined than before. There should be no more ambiguity in the final FIPS197 standard. However, programmers are still needed to

take notice of the definition in the standard may be inconsistency with their chosen programming language in aspect of orientation of arrays.

3.2 Modified the size of the state arrays

The most significant amendment in the final FIPS197 standard is in the size of the state array. The original arrays allow three different size of arrays with Nb(numbers of columns)= 4 , Nb=5 or Nb=6. To achieve the maximum efficiency of the 32-bit processors, the final AES standard does merely allow Nb=4(that is with a cyber block of 128-bit). The reason can be easily understood by reading our explanations in the next section.

4. The analysis of AES Algorithms and its implementation issues[1][2]

The Rijndael cipher is suited to be implemented efficiently on a wide range of processors and in dedicated hardware. Its algorithm is based on the byte (8-bit)-manipulation. However, most of contemporary computer platforms have a 32-bit (or above) processor. Although the Rijndael algorithm is suitable for all range of processors, this algorithm will be less efficient while executing in a 32-bit (or above) processor. The latest version of 32-bit algorithm is provided and suited for processors over 32 bits.

4.1 Using word (4 bytes)-manipulation in Transposed Matrix

The original matrix suffers the efficiency problem as described in section 2.2, hence the modified 32-bit algorithm will always uses the transposed matrix. This will also benefit the word manipulation in the row since there is always 4 bytes in each row on the transposed matrix. The original state matrix and cipher key matrix are not uniform in the row. In the original matrix, there are 4, 6, and 8 elements in a row each for 128, 192 and 256 bit-data, hence we can not handle the row by word manipulation for all cases.

In the original matrix, there is always four elements in column for all cases (128, 192 and 256 bits data). After transposing, the new matrix will always has 4 elements in the row. Instead of four

continuous byte-access in the row, the 32-bit algorithm manipulates it as one word-access. For most of current computers with 32-bit (or above) CPU, it takes only one access for one word in the row. In C programming language, we can treat the word (four-byte) data as the type of long integer. The word-manipulation (such as additions) can be performed in this data of long-integer type. It will take less than one-fourth of time in comparing with byte-manipulation.

4.2 Using lookup table to enhance substitution and multiplication

In the proposed Rijndael Algorithm, substitutions and multiplications always waste plenty of time. The 32-bit algorithm replaces those mathematical operations by looking up table. It merely takes a index lookup array of 256 elements for byte substitution. This will be much faster than substitution operation. All mapping results are calculated in advance, then stored these results in a array [256] for all the 256 input conditions. Of course, the index lookup will be much faster than mathematical substitution operation which is composed of array multiplication and array addition.

The 32-bit algorithm also speeds up the mixcolumn operation by lookup table. The original mixcolumn operation is as Fig 6, where $\vec{a}=[\,a_0 \quad a_1 \quad a_2 \quad a_3\,]^t$ vector is from the state.

$$\begin{bmatrix} b0 \\ b1 \\ b2 \\ b3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} a0 \\ a1 \\ a2 \\ a3 \end{bmatrix}$$

Fig. 6

By using byte-multiplication, it takes 16 multiplications to acquire the result vector of $\vec{b}$. The 32-bit algorithm will replace these multiplications by the lookup table. The original mixcolumn could be modified as:

$$\begin{bmatrix} 2 \\ 1 \\ 1 \\ 3 \end{bmatrix}^t a_0 + \begin{bmatrix} 3 \\ 2 \\ 1 \\ 1 \end{bmatrix}^t a_1 + \begin{bmatrix} 1 \\ 3 \\ 2 \\ 1 \end{bmatrix}^t a_2 + \begin{bmatrix} 1 \\ 1 \\ 3 \\ 2 \end{bmatrix}^t a_3 = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}^t$$

rewritten as:

$$\vec{p}_0 + \vec{p}_1 + \vec{p}_2 + \vec{p}_3 = \vec{b}$$

where $\vec{p}_0 = \begin{bmatrix} 2 \\ 1 \\ 1 \\ 3 \end{bmatrix}^t a_0$, $\vec{p}_1 = \begin{bmatrix} 3 \\ 2 \\ 1 \\ 1 \end{bmatrix}^t a_1$, $\vec{p}_2 = \begin{bmatrix} 1 \\ 3 \\ 2 \\ 1 \end{bmatrix}^t a_2$, $\vec{p}_3 = \begin{bmatrix} 1 \\ 1 \\ 3 \\ 2 \end{bmatrix}^t a_3$, $\vec{b} = \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}^t$

The column vectors are transposed. We would treat it as 4-byte row vector. The partial result of $\vec{p}_0, \vec{p}_1, \vec{p}_2, \vec{p}_3$ can be treated as one word and can be placed in the consecutive memory location. In high-level language such as C, we can read those vectors by a type of long integer. For 32 bits computer it takes only one time for data access.

Multiplication is the most time-wasting operators in Rijndael algorithm. Instead of using multiplication, we use lookup tables in acquiring the partial result of $\vec{p}_0$, $\vec{p}_1$, $\vec{p}_2$, $\vec{p}_3$. The multiplication result can be calculated in advance and store in the array. Therefore, every partial result can be acquired by looking up method. The lookup table for mixcolumn needs to be arranged as an array of 4*256, since each partial result $\vec{p}$ is composed of four elements. We can design 4 kinds of lookup array. The first lookup array is for vector $[2\ 1\ 1\ 3]^t$ which is multiplied by a_0. The second one for vector $[3\ 2\ 1\ 1]^t$ which is multiplied by a_1, and so on.

Instead of 4 multiplication needed for each partial result, the 32-bit algorithm uses only one index lookup. In comparing with byte-manipulation, it takes much less than one-fourth of time in acquiring the result $\vec{b}$ vector. The original byte-manipulation method needs 16 additions. The 32-bit algorithm needs only 4 additions to sum up 4 partial result of $\vec{p}_0, \vec{p}_1, \vec{p}_2, \vec{p}_3$. It will take only one-fourth of time in comparing with byte-addition.

For the decryption, the 32-bit algorithm will use the same technique in the inverse operation. All the lookup tables have been calculated in advance. All the substitutions and multiplications are replaced by lookup. The encryption of 32-bit algorithm will be at the same speed as the encryption. Because we use lookup table to replace the inverse of multiplication, the looking time will be almost

as the same as the one in encryption.

4.3 Speedup key schedule

The efficiency of key schedule is a major concern of encryption/decryption. All the intermediate round for encryption/decryption will perform only after the intermediate key was expanded. To explain the 32-bit algorithm of key schedule, we take an example of key with Nk=4. The original Rijndael key schedule for Nk=4 is illustrated in Fig 7. Other cases for Nk=6 and Nk=8 will be also modified by this method.

Step 1 : W4=W0+W3t where W3t=SubByte(RotByte(W3)) + Rcon[4/Nk]
Step 2 : W5=W1+W4
Step 3 : W6=W2+W5
Step 4 : W7=W3+W6
……

Fig 7 Key expansion example for Nk=4

To speed up the key expansion, all additions use the word-manipulation as the same reason as we described in previous section. After the key matrix being transposed, there are 4 elements in each row. The 4-element byte-data in the first row can be represented as word-data W0, which is followed by word data W1, W2 and W3. To expanse the next round key, W4 can be acquired by word-addition of W0+W3, as illustrated in Figure 7 step 1.As the same method, W5, W6 and W7 can be acquired in step 2, 3 and 4.

4.4 Expediting the Decryption

The decryption for the block cipher is generally to manipulate the cipher in the reverse steps and to use the inverse operations as in the encryption. Since the operations in encryption are not symmetrical, we need to develop another procedure to handle the decryption. If there exists another equivalent decryption procedure whose operations order is the same as the one in the encryption, we can use the same procedure (exception for using reverse operations during decrypting) to manipulate both encryption and decryption. This will benefit on the speedup of decryption. There is another benefit that

is particular useful when we implement both encryption and decryption in the hardware. It means we need only one set of process logic circuit. It will save both cost and time in the decryption. We will prove the equivalent decryption procedure in the following paragraph.

Before going on further proof, we need to define some notations. The encryption on AES Rijadael composes of four district operations. [8]

(a) Substitution γ:

γ: $b = \gamma(a) \Leftrightarrow b_{i,j} = S_\gamma(a_{i,j})$

γ is a nonlinear byte substitution with an invertible 8-bit S-Box. The inverse of γ is the inverse substitution S_γ^{-1}(by using inverse lookup table) to all bytes of a state.

(b) ShiftRow π:

The inverse of ShiftRow π is to shift the row in the reversed direction.

(c) Mixcolumn θ:

θ: $b = \theta(a) \Leftrightarrow b_{i,j} = c_j\, a_{i,0} \oplus c_{j-1}\, a_{i,0} \oplus c_{j-2}\, a_{i,2} \oplus c_{j-3}\, a_{i,3}$

Mixcolumn is a linear operation that operations separately on each of the four rows of a state. Where the multiplication is in $GF(2^8)$ and the indices of c must be taken modulo 4. The polynominal corresponding to row i of a state a is given by

$a_i(x) = a_{i,0} \oplus a_{i,0}\, x \oplus a_{i,2}\, x^2 \oplus a_{i,3}\, x^3$

Using this notation, we define $c(x) = \oplus_j c_j x^j$. The Mixcolumn can be rewritten as

θ: $b = \theta(a) \Leftrightarrow b_i(x) = c(x)\, a_i(x) \bmod (1 \oplus x^4)$

The inverse of θ corresponds to a polynominal d(x) is given by

$d(x)\, c(x) = 1 \bmod (1 \oplus x^4)$

c(x) is chosen as $2 \oplus 1\, x \oplus 1\, x^2 \oplus 3\, x^3$

d(x) is chosen as $E \oplus 9\, x \oplus D\, x^2 \oplus B\, x^3$

Using this notation, we define

θ^{-1}: $a = \theta(b) \Leftrightarrow a_i(x) = d(x)\, b_i(x) \bmod (1 \oplus x^4)$

(d) Keyaddition σ:

$\sigma[K^t]$: $b = \sigma[K^t](a) \Leftrightarrow b = a \oplus K^t$

$\sigma^{-1}[K^t] = \sigma[K^t]$: $b = \sigma^{-1}[K^t](a) \Leftrightarrow b = a \oplus K^t$

$\sigma[K^t]$ consists of the bitwise addition of a round key K^t. The inverse of $\sigma[K^t]$ is itself.

The encryption in the AES complete round consists of the round transformation denoted by $\rho[K^t]$.

$\rho[K^t] := \rho[K^t] \circ \theta \circ \pi \circ \gamma$

If we encrypt the state as 10 rounds proceeded by an initial key addition $\sigma[K^0]$, then the encryption procedure is as

$$\text{Encrypt}(k) = (\rho[K^{10}] \circ \pi \circ \gamma) \circ \rho[K^9] \circ \rho[K^8] \circ \rho[K^7] \circ \rho[K^6] \circ \rho[K^5] \circ \rho[K^4] \circ \rho[K^3] \circ \rho[K^2] \circ \rho[K^1] \circ \sigma[K^0] \quad (1)$$

The decryption procedure should be

$$\text{Decrypt}(k) = \sigma[K^0] \circ \rho^{-1}[K^1] \circ \rho^{-1}[K^2] \circ \rho^{-1}[K^3] \circ \rho^{-1}[K^4] \circ \rho^{-1}[K^5] \circ \rho^{-1}[K^6] \circ \rho^{-1}[K^7] \circ \rho^{-1}[K^8] \circ \rho^{-1}[K^9] \circ (\gamma^{-1} \circ \pi^{-1} \circ \rho[K^{10}]) \quad (2)$$

Where

$$\rho^{-1}[K^t] := \gamma^{-1} \circ \pi^{-1} \circ \theta^{-1} \circ \sigma^{-1}[K^t] \quad (3)$$

Since γ^{-1} only operates on the individual bytes, we have $\gamma^{-1} \circ \pi^{-1} = \pi^{-1} \circ \gamma^{-1}$. Equation (3) can be rewritten as

$$\rho^{-1}[K^t] := \pi^{-1} \circ \gamma^{-1} \circ \theta^{-1} \circ \sigma[K^t] \quad (4)$$

Moreover, since

$\theta^{-1}(a) \oplus K^t = \theta^{-1}(a \oplus \theta(K^t))$, we have

$$\sigma[K^t] \circ \theta^{-1} = \theta^{-1} \circ \sigma[\theta(K^t)] \quad (5)$$

We can also derive

$$\theta^{-1} \circ \sigma[K^t](a) = \theta^{-1}(a \oplus K^t) = \theta^{-1}(a) \oplus \theta^{-1}(K^t) = \sigma[\theta^{-1}(K^t)] \circ \theta^{-1}(a)$$

$$\Rightarrow \theta^{-1} \circ \sigma[K^t] = \sigma[\theta^{-1}(K^t)] \circ \theta^{-1} \quad (6)$$

Now, we define the complete round transformation of the decryption as

$$\rho'[K^t] := \sigma[K^t] \circ \theta^{-1} \circ \pi^{-1} \circ \gamma^{-1} \quad (7)$$

The decryption of round 10 and round 9 can be written as

$$\rho^{-1}[K^9] \circ \gamma^{-1} \circ \pi^{-1} \circ \sigma^{-1}[K^{10}] = \pi^{-1} \circ \gamma^{-1} \circ \theta^{-1} \circ \sigma[K^9] \circ \pi^{-1} \circ \gamma^{-1} \circ \sigma[K^{10}] = \pi^{-1} \circ \gamma^{-1} \circ \sigma^{-1}[\theta^{-1}(K^9)] \circ \theta^{-1} \circ \pi^{-1} \circ \gamma^{-1} \circ \sigma[K^{10}]$$

$$= \pi^{-1} \circ \gamma^{-1} \circ \rho'[\theta^{-1}(K^9)] \circ \sigma[K^{10}] \quad (8)$$

We can continue the rest of rounds, thus the decryption of ten rounds can be rewritten as

$$\text{Decrypt}(k) = (\sigma[K^0] \circ \pi^{-1} \circ \gamma^{-1}) \circ \rho'[\theta^{-1}(K^1)] \circ \rho'[\theta^{-1}(K^2)] \circ \rho'[\theta^{-1}(K^3)] \circ \rho'[\theta^{-1}(K^4)] \circ \rho'[\theta^{-1}(K^5)] \circ \rho'[\theta^{-1}(K^6)] \circ \rho'[\theta^{-1}(K^7)] \circ \rho'[\theta^{-1}(K^8)] \circ \rho'[\theta^{-1}(K^9)] \circ \sigma[K^{10}] \quad (9)$$

The decryption procedure in equation (9) is equivalent to decryption procedure in equation (2). From equation (9), we can clearly find that the inverse decryption have the same structure as the one in encryption equation (1). We use equation (9) in the decryption since it will allow us to save both cost and time in the implementation. In equation (9), the internal complete round (from round 1 to round 9) takes round key $\theta^{-1}(K^t)$ rather than K^t. This is the reason why we list two different time for key scheduling in the performance result of 32-bit algorithm. The first one is for encrypting and the second one is for decrypting.

5. Performance evaluation

To show the enhanced performance in the 32-bit algorithm, we compared with the old version of algorithm which use byte-manipulations. Table 1 through table 3 is the test results run at PC platform of 450MHz CPU, 128M DRAM. We also executed all programs (written in C language) by using NIST API in different platforms (table 4 through table 6). Table 1 shows the performance for original algorithm of byte-manipulations. To speed up byte-manipulation, we replace byte-multiplication by lookup table. The performance is showed in table 2.

Table 1. Original algorithm (Byte-Manipulation)

key length	**Key schedule**	**Encrypt**	**Decrypt**
bits	Mbps	Mbps	Mbps
128	4.4	1.42	0.85
192	5.5	1.13	0.71
256	6.2	0.99	0.56

Table 2. Speedup in enhanced original algorithm where multiplication is replaced by Lookup table (in comparison with the original 8-bit algorithm)

key length	**Key schedule**	**Encrypt**		**Decrypt**	
bits	Mbps	Mbps	Speedup	Mbps	Speedup
128	4.4	8.1	5.7	7.2	8.5
192	5.5	6.7	5.9	6.0	8.5
256	6.2	5.8	5.8	5.2	9.3

As the result showed, the lookup table can speed up the encryption/decryption more than five times. Table 3 shows the enhanced performance of the 32-bit algorithm which speeds up the encryption/decryption more than sixty times in comparison with the old algorithm of byte-manipulations.

Table 3. Performance of the 32-bit algorithm and its speedup (in comparison with the original 8-bit algorithm)

key length	**Key schedule for Encryption**		**Encrypt**		**Key schedule for Decryption**		**Decrypt**	
bits	Mbps	Speedup	Mbps	Speedup	Mbps	Speedup	Mbps	Speedup
128	50.7	11.5	87	61.2	40.1	9.1	86.5	101.7
192	51.5	9.4	73.9	65.4	42.3	7.7	73.4	103.3
256	52.7	8.5	64.2	64.8	44.1	7.1	64.1	114.5

Transpose matrix, word-manipulation and lookup table in our modified algorithm contributes the enhancement in encryption/decryption. The enhancement in the key schedule is also showed in table 3. The 32-bit algorithm speeds up the key schedule about nine times, which is contributed by transpose matrix and word-manipulation.

We also executed our program in different platforms. Table 4 shows the performance in PC platform of 233MHz CPU, 64M DRAM. Table 5 shows the performance in PC platform of 450MHz CPU, 128M DRAM. Table 6 shows the performance in PC platform of 900MHz CPU, 256M DRAM. Some AES test results are also provided in the AES Proposal [2]. We choose their related test results of C programs run on 200MHz Pentium to compare. Their performance on encryption is listed as:

27.0Mbps for 128-bit key, 22.8Mbps for 192-bit key and 19.8Mbps for 256-bit. There is another test result provided by Brian Gladman, which have a better efficiency on performance. We would not compare with this set of data since they replace some part of program by assembly code (the core of encryption and decryption) in this version. Our codes are all implemented in high level C language. Our test result run on PC with 233Mhz CPU is listed as table 4: 44.0Mbps for 128-bit key, 37.9Mbps for 192-bit key and 32.3Mbps for 256-bit key. We have a better test result on the performance in comparison with their public announcement in the AES proposal.

Table 4 Performance in PC platform of 233MHz CPU, 64M DRAM

key length	**Key schedule for Encryption**	**Encrypt**	**Key schedule for Decryption**	**Decrypt**
bits	Mbps	Mbps	Mbps	Mbps
128	23.2	44.0	16.6	44.9
192	25.7	37.6	18.2	37.5
256	27.8	32.3	19.9	32.4

Table 5 Performance in PC platform of 450MHz CPU, 128M DRAM

key length	**Key schedule for Encryption**	**Encrypt**	**Key schedule for Decryption**	**Decrypt**
bits	Mbps	Mbps	Mbps	Mbps
128	48.7	87.0	40.1	86.5
192	51.5	73.9	42.3	73.4
256	52.7	64.2	44.1	64.1

Table 6 Performance in PC platform of 900MHz CPU, 256M DRAM

key length	**Key schedule for Encryption**	**Encrypt**	**Key schedule for Decryption**	**Decrypt**
bits	Mbps	Mbps	Mbps	Mbps
128	89.2	138.8	40.1	138.6
192	93.2	118.3	42.3	116.2
256	97.5	103.1	44.1	102.8

The results show the platform of 450MHz CPU which can speed up the operations of key scheduling and encryption/decryption about two times. Nevertheless, the platform of 900MHz CPU can merely speed up the operations of key scheduling and encryption/decryption less than two times. When the CPU clock rate is quite fast to the extension, the bottleneck of performance will not on the CPU but on the data-bus.

6. Conclusion

AES is the next generation secret encryption standard of NIST. The original DEA uses a 64-bit block size encryption/decryption. It is not secure enough for current computer technology. TDEA (Triple DEA) has a 192-bit key, which has three times as many rounds as DEA, is correspondingly slower. Both use a 64-bit block size encryption/decryption. A large block size is desirable for reasons of both efficiency and security. Because of these drawbacks, AES is called and proposed as the next generation standard. It is more efficient than DEA. The AES algorithm is designed for general type of computers. However, current microprocessors are well designed in the hardware architecture. Many microprocessors are over 32 bits. There is no reason to use the general AES algorithm which uses byte-manipulations in encryption/decryption. In this paper we analyze the modified 32-bit AES algorithm, which can fully utilize the CPU power for computers with 32-Bit (or above) processors.

According to the performance evaluation in section 5, the modified 32-bit AES algorithm can speed up the encryption/decryption more than sixty times. For the key schedule, it speeds up about eight times. The modified AES algorithm can largely enhance the efficiency of encryption/decryption if 32-bit processor (or above) was selected as the encrypting tool. Some skills of the modified algorithm can be also applied in 8-bit processors, which include transposed matrix to enhance the efficiency of data access and lookup tables to enhance the byte-multiplication.

The Rijndael Algorithm becomes the standard of AES in Nov 26, 2001. Since most of the computer languages are row-major in the matrices, we remind that programmers should take notice of the definition of the standard which may be inconsistent with their chosen programming language in aspect of orientation of arrays. In these cases, programmers should use transposed matrix on their implementations to acquire a better efficiency. According to our evaluation in this paper, the 32-bit algorithm can be a very nice choice in the AES implementation on the target system with 32-bit processor. Programmers should always take advantage of the 32-bit algorithm if 32-bit (or above) processors were supported by their target systems.

Reference

1. FIPS197(Federal Information Processing Standards Publication 197), Nov 26, 2001,Federal Register Announcement.
2. AES Proposal: Rijndael, Joan Daemen,Vincent Rijmen,http://csrs.nist.gov/encryption/aes/rijndael/Rijndael.pdf
3. http://www.nist.gov/aes
4. Neng-Wen Wang, Yueh-Min Huang, Chi-Sung Laih, “Concerns about the efficiency of data access and ambiguity in the AES Proposal”, Public Comments on the Draft Federal Information Processing Standards (FIPS) for the Advanced Encryption Standard
5. William Stallings, Network Security Essentials-applications and standards, Prentice Hall,2000
6. Carl Hamacher etc., Computer Orgranazation, fifth ed.,McGraw Hill, p296-p307.
7. Sao-Jie Chen, http://www.cc.ee.ntu.edu.tw, Computer Architecture, Ch. 7 Memory System
8. Joan Daemen, L.Knudsen, V.Rijmen,”The Block Cipher Square”, Fast Software Encryption 1997,Spinger LNCE 1267, pp.149-165.