Study of AES and its Efficient Software Implementation

Eashwar Thiagarajan and Madhuri Gourishetty
Department of Electrical Engineering & Computer Science,
Oregon State University, Corvallis, Oregon 97331 -USA.
E-mail: *teashwar@engr.orst.edu & gourisma@engr.orst.edu*

***Abstract*— This work aims to familiarize the reader with the concepts behind AES or Advanced Encryption Standard. This work shall serve as a useful starting point for those who are interested in thinking along lines of software implementation of AES. To do this we shall go over some basic definitions in the context of AES and then explain AES at large from the perspective of various authors and papers. After this a brief discussion of an efficient software implementation of AES shall follow.**

INTRODUCTION

Section 1 of this paper shall introduce the reader to AES. Section 2 shall discuss the implementation of the AES, with concepts like finite field operations - Multiplication, Modulo-2 Addition. Section 3 sheds light on the Softfare implementations of AES and briefly discusses some performance aspects. Section 4 reviews an already presented matter on Efficient implementation of AES.

I. WHAT IS AES ?

The Advanced Encryption Standard (AES) is an encryption algorithm for securing sensitive but unclassified material by U.S. Government agencies. The Rijndael algorithm proposed as an accepted standard is discussed in the sections that follow. The AES algorithm is aimed at replacing the DES. The figure 1 elucidates the main steps comprising the AES algorithms.The four major functions that comprise the AES are Add Round Key, Substitute bytes, Shift Rows and Mix Columns. Rijndael is a substitution-linear transformation network with 10, 12 or 14 rounds, depending on the key size. A data block to be processed using Rijndael is partitioned into an array of bytes, and each of the cipher operations is byte-oriented.

II. AES IMPLEMENTATION DETAILS

A byte in Rijndael is a group of 8 bits and is the basic data unit for all cipher operations. Such bytes are interpreted as finite field elements using polynomial representation, where a byte b with bits b0, b1,...b7 represents the finite field elements, as shown below.

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x + b_0 = \sum_{i=0}^{7} b_i x^i$$

Internally Rijndael operates on a two dimensional array of bytes called the state that contains 4 rows and Nc columns, where Nc is the input sequence length divided by 32. In this state array, denoted by the symbol s, each individual byte has two indexes: its row number r, in the range 0 ??r ¡ 4, and its column number c, in the range 0 ??c ¡ Nc, hence allowing it to be referred to as s[r, c]. For AES the range for c is 0 ??c ¡ 4 since Nc has a fixed value of 4. The conversion from plain text to State Matrix is shown in Figure 1.

As discussed, bytes are can be represented as polynomials. Finite field operations like addition and multiplication are required for key scheduling and rounding.The addition of two finite field elements is achieved by adding the coefficients for corresponding powers in their polynomial representations, this addition being performed in GF(2), that is, modulo 2, so that 1 + 1 = 0. Addition is nothing but performing XOR between two expressions.

Finite field multiplication is more difficult than addition and is achieved by multiplying the polynomials for the two elements concerned and collecting like powers of x in the result. Since each polynomial can have powers of x up to 7, the result can have powers of x up to 14 and will no longer fit within a single byte. This is overcome by using a field generator. The product is divided by this field generator and the remainder is taken as the result. Since there are 256 possible polynomials, a look up table can be created for a specific field generator. So the look up table will have 256 * 256 entries.

Key scheduling: The round keys are derived from the cipher key by means of a key schedule with each round requiring Nc words of key data. For 128-bits keys the key scheduling operates intrinsically on blocks of 4 32-bits words; we can calculate one new round key from the previous one.We denote the ith word of the actual round key with K[i], where 0¡i¡4, and the ith word of the next round key with K[i]. K[0] is computed by an XOR between K[0], a constant r (field generator) and K[3], the latter being pre rotated and transformed. The other three words K[1], K[2] and K[3] are calculated as K[i] = K[i] . K[i - 1].

Rounding: At the start of the cipher the cipher input is copied into the internal state using the conventions described before (Figure 1). An initial round key is then added and the state is then transformed by iterating a round function in a number of

Authors are graduate students at the Department of Electrical Engineering & Computer Science, Oregon State University, Corvallis, Oregon 97331. E-mail: *teashwar@engr.orst.edu & gourisma@engr.orst.edu*

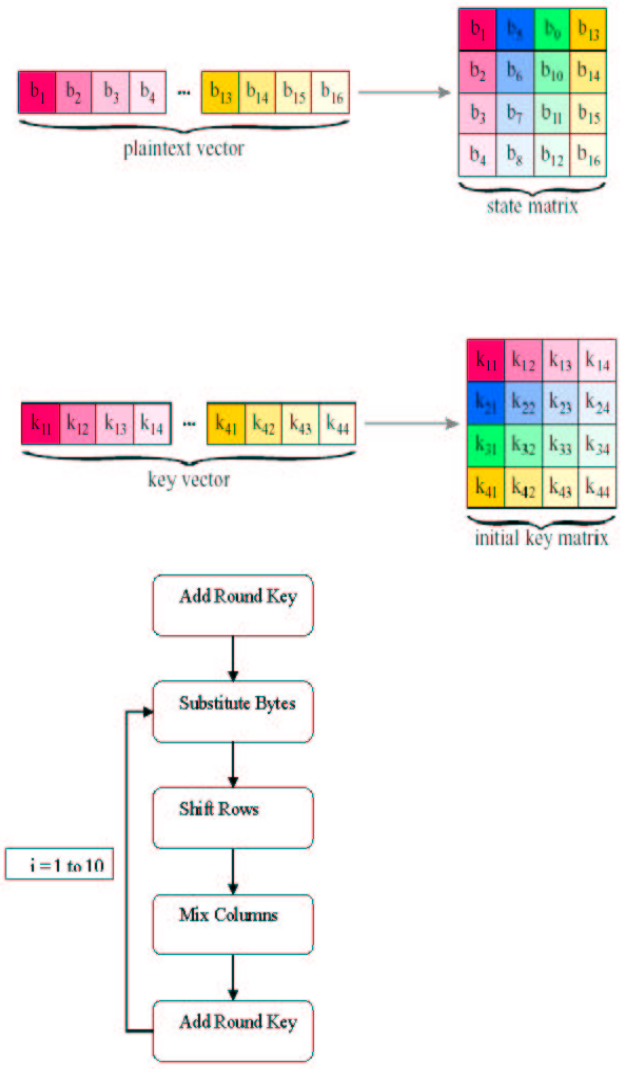

Fig. 1. Four Major functions of AES

cycles. The number of cycles Nn varies with the key length and block size. On completion the final state is copied into the cipher output using the same conventions. The steps involved in AES are shown in Illustration1. It is the algorithm for the Rijndael algorithm.

Round Function: One round is termed as a cycle and each cycle has four steps. Each step of transformation is described below.

Add Round Key: This is the first step of transformation. The XorRoundKey function declared in Illustration 1 has to perform a bitwise xor of the state matrix and the round key matrix.

Sub-Bytes Transformation: Inverse of the state matrix is found here and affine transformation is performed.

Shift Rows: The ShiftRows transformation operates individually on each of the last three rows of the state matrix by cyclically shifting the bytes in the row. The second row is shifted one time to the left, third row shifted two times and fourth row shifted three times.

Mix Columns: The mix columns transformation computes the new state matrix S by left-multiplying the current state matrix S by the polynomial matrix P:S = P o S where P is a fixed matrix as shown below.

```
Cipher(byte in[4*Nc], byte out[4*Nc], word k[Nn+1,Nc], Nc, Nn)
Begin
  byte state[4,Nc]   // The notation k[Nn+1,Nc] above indicates that
                     // the array k contains Nn + 1 individual round
  state = in         // keys that are each arrays of Nc words

  XorRoundKey(state, k[0,-], Nc)        // k[0,-] = k[0..Nc-1]

  for round = 1 step 1 to Nn - 1
    SubBytes(state, Nc)
    ShiftRows(state, Nc)
    MixColumns(state, Nc)
    XorRoundKey(state, k[round,-], Nc) // k[round*Nc..(round+1)*Nc-1]
  end for

  SubBytes(state, Nc)
  ShiftRows(state, Nc)
  XorRoundKey(state, k[Nn,-], Nc)       // k[Nn*Nc..(Nn+1)*Nc-1]

  out = state
end
```

Fig. 2. Illustration: AES Algorithm

$$\begin{bmatrix} s'_{11} & s'_{12} & s'_{13} & s'_{14} \\ s'_{21} & s'_{22} & s'_{23} & s'_{24} \\ s'_{31} & s'_{32} & s'_{33} & s'_{34} \\ s'_{41} & s'_{42} & s'_{43} & s'_{44} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \bullet \begin{bmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \\ s_{31} & s_{32} & s_{33} & s_{34} \\ s_{41} & s_{42} & s_{43} & s_{44} \end{bmatrix}$$

III. Software Implementation

Software implementations cover a wide range. In some cases, space is essentially unrestricted; in other cases, RAM and/or ROM may be severely restricted. In some cases,large quantities of data are encrypted or decrypted with a single key. In the case of AES, the key changes frequently, perhaps with each block of data. One issue that arises in software implementations is the basic underlying architectures, whether the architecture is 32 bit processor, 8 bit etc. It should be noted that performance cannot be classified by word size alone. The performance of AES or any other encryption algorithm depends on a particular high-level language used (like assembler, compiler or interpreter). In most of the cases, the software plays an important role and strongly affects the performance figures. Hand-coded assembly code will generally produce better performance results than an optimizing compiler. Interpreted languages are, in general, poorly adapted to the task of optimizing performance. Compilers are in between. Some compilers perform better because of the underlying architecture. This increases the difficulty of measuring performance across a variety of platforms. It is found that Rijndael performed better on some platforms when hand-coded assembler was used as opposed to compilers. For Rijndael, key setup or encryption/decryption is noticeably slower for 192-bit keys than for 128-bit keys, and slower still for 256-bit keys. Rijndael specifies more rounds for the larger key sizes, affecting the speed of both encryption/decryption and key setup.

So from the above discussion it is clear that the performance of the implementation depends on the underlying architecture. C and assembler (64 bit processor) have better performance over Java (32 bit processor). Java and C codes are studied for comparisons, but it is found difficult to find the clock cycles, because Java supports only 32 bits. A Matlab implementation of AES is also found. The main advantage of implementing with Matlab is the understandability, more than speed of execution.

In the next section, an efficient way of AES is described which improves the performance of software implementation.

IV. Efficient Software Implementation of AES

This section illustrates optimized version of the Rijndael AES algorithm. The details of this section can be found in r eference [3]. Both the encryption and the decryption algorithms are optimized. The task is divided in two parts: optimization of the algorithm working on the State matrix, and modification of the key scheduling. In both cases, the base line consists in the transposition of the State matrix, and the consequent rearranging of the various transformations. In fact, as a consequence of the transposition of the State matrix, also the key scheduling is rearranged in a suited way.

A. The Transposed State Matrix Primitives

It is possible to enhance the throughput of the implementation of AES by changing the way in which data are represented by the software. For this study, look up tables were used for Sub-Byte transformations. A little amount of space was given to S-Box and inverse S-Box. All the remaining operations were computed dynamically.All the primitives considered in the study behave in a peculiar way, operating on a transposed version of the State matrix. All the steps of the algorithm must be modified in order to preserve global functionality while operating on the transposed State matrix. Only the encryption is described here.

Steps of AES:

Add Round Key: This remains unchanged. It is just XOR between state matrix and the round keys. But round key generation is changed.

SubBytes transformation is not changed, because it is not dependant on the State Matrix.

Shift Rows is changed - This transformation does not shift rows but it operates in the same way on columns.

The MixColumns transformation is deeply revised. As shown in the above section, the Mix columns transformation is nothing but multiplying the State Matrix with a fixed polynomial matrix.

$$\begin{bmatrix} s'_{11} & s'_{12} & s'_{13} & s'_{14} \\ s'_{21} & s'_{22} & s'_{23} & s'_{24} \\ s'_{31} & s'_{32} & s'_{33} & s'_{34} \\ s'_{41} & s'_{42} & s'_{43} & s'_{41} \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \bullet \begin{bmatrix} s_{11} & s_{12} & s_{13} & s_{14} \\ s_{21} & s_{22} & s_{23} & s_{24} \\ s_{31} & s_{32} & s_{33} & s_{34} \\ s_{41} & s_{42} & s_{43} & s_{41} \end{bmatrix}$$

In this study, Mix columns transformation is completely changed.Instead of using a 4 * 4 matrix, here only one column is used which has 32 bits.

Here xi, for i = 0 to 3, is the 32-bits words (or columns) of the transposed State matrix before applying the MixColumns transformation. yi, for j = 0 to 3, is the 32-bits words (or columns) of the transposed State matrix after applying the MixColumns transformation. The revised version of MixColumns is then represented by the following set of equations:

y0 = 02 * x0 (+) 03 * x1 (+) x2 (+) x3
y1 = x0 * 02 (+) x1 (+) 03 * x2 (+) x3
y2 = x0 (+) x1 (+) 02 * x2 (+) 03 * x3
y3 = 03 * x0 (+) x1 (+) x2 (+) 02 * x3

As described, the variables xi and yi contain the 4 bytes at position i of the columns of the State matrix in the normal, non-transposed, version of the transformation. These variables are 32 bits long. 02 * x0 means multiplying the polynomial with x0. x0 is 32 bits. So multiplication does not mean the normal GF multiplication. But 4 multiplications are performed simultaneously to be precise parallely on 4 bytes of the 32 bit word. The generator polynomial i.e. the field generator remains the same. These calculations can be done in a simple way. Use the yi variable as an accumulator, and to use the xi variable for storing the product of the initial values of xi and of the 4 partial products: xi, 2 * xi, 4 * xi and 8 * xi. Therefore the MixColumns transformation is computed in only 3 steps: a sum step, a doubling step and a final sum step. The three steps are shown below

First	Second	Third
y0 = x1 (+) x2 (+) x3	x0 = 02 (+) x0	y0 (+) = x0 (+) x1
y1 = x0 (+) x2 (+) x3	x1 = 02 (+) x1	y1(+) = x1 (+) x2
y2 = x0 (+) x1 (+) x3	x2 = 02 (+) x2	y2(+) = x2 (+) x3
y3 = x0 (+) x1 (+) x2	x3 = 02 (+) x3	y3(+) = x0 (+) x3

B. The Transposed State Matrix Key Scheduling

As described in the above sections, we need to transpose the keys before using them. This can be done by first calculating the round keys and then transposing them. But this involves a big overhead. So a better way is suggested in the paper (reference [3]). Key scheduling is redesigned to get the transposed one directly. We will here see the details of the implementation. See reference [3] for more details.

For 128-bits keys the key scheduling operates intrinsically on blocks of 4 32-bits words; we can calculate one new round key from the previous one.We denote the ith word of the actual round key with K[i], and the ith word of the next round key with K^1[i]. K^1[0] is computed by an XOR between K[0], a constant field generator and K[3], the latter being pre rotated and transformed using the S-BOX. The other three words K^1[1], K^1[2] and K^1[3] are calculated as K^1[i] = K[i] . K^1[i - 1].

The new way of representing the round keys is shown below:-

$$\mathbf{K_T}[0] = \begin{bmatrix} k_0 \\ k_4 \\ k_8 \\ k_{12} \end{bmatrix} \mathbf{K_T}[1] = \begin{bmatrix} k_1 \\ k_5 \\ k_8 \\ k_{13} \end{bmatrix} \mathbf{K_T}[2] = \begin{bmatrix} k_2 \\ k_6 \\ k_9 \\ k_{14} \end{bmatrix} \mathbf{K_T}[3] = \begin{bmatrix} k_3 \\ k_7 \\ k_{10} \\ k_{15} \end{bmatrix}$$

The transposed key schedule is made up of the following transformations:

$$\begin{aligned} K'_T[0] &= K_T[0] \oplus (pad(Sbox(k_{13})) << 24) \oplus rcon \\ K'_T[1] &= K_T[1] \oplus (pad(Sbox(k_{14})) << 24) \\ K'_T[2] &= K_T[2] \oplus (pad(Sbox(k_{15})) << 24) \\ K'_T[3] &= K_T[3] \oplus (pad(Sbox(k_{12})) << 24) \\ K'_T[0] \oplus &= (K_T[0] >> 8) \oplus (K_T[0] >> 16) \oplus (K_T[0] >> 24) \\ K'_T[1] \oplus &= (K_T[1] >> 8) \oplus (K_T[1] >> 16) \oplus (K_T[1] >> 24) \\ K'_T[2] \oplus &= (K_T[2] >> 8) \oplus (K_T[2] >> 16) \oplus (K_T[2] >> 24) \\ K'_T[3] \oplus &= (K_T[3] >> 8) \oplus (K_T[3] >> 16) \oplus (K_T[3] >> 24) \end{aligned}$$

”pad” means zero-padding of the 24 most significant bits of the word since SBox returns an 8 bits value. We can note that the computation overhead with respect to the normal, non-transposed key scheduling is just few shift operations.

C. Performance results

It is found that this implementation has given good results. The implementation is done in C. There are two kinds of implementation - key unrolling and key on the fly. As discussed above round keys can be found at the beginning of the encryption or on the fly, because they are not dependant on any other data. Finding keys on the fly requires less memory but the latter requires more because all the keys should be stored. Results for both cases are shown below. The results are compared with an equivalent version of AES by Dr. Brian Gladman, who has been involved in the definition of the AES standard and his version is well referenced. The code is run on different processors for comparing the results. The results are shown in table 1 and table 2.

CPU	Implementation	Key Schedule	Encryption	Decryption
ARM7TDMI	Proposal in [3]	634	1675	2074
	Gladman	449	1641	2763
ARM9TDMI	Proposal in [3]	499	1384	1764
	Gladman	333	1374	2439
ST22	Proposal in [3]	0.22	0.51	0.60
	Gladman	0.13	0.61	1
Pentium III	Proposal in [3]	370	1119	1395
	Gladman	396	1404	2152
	Gladman with tables	202 (encrypt.) 306 (decrypt.)	362	381

Fig. 3. Table 1: Clock cycles required for AES on different platforms (using key unrolling)

CPU	Implementation	Encryption	Decryption
ARM7TDMI	Proposal in [3]	2074	2378
	Gladman	1950	3221
ARM	cAESar	2889	N.A.
	cAESar with tables	1467	N.A.
ARM9TDMI	Proposal in [3]	1755	1976
	Gladman	1623	2796
ST22	Proposal in [3]	0.72	0.82
	Gladman	0.75	1.13

Fig. 4. Table 2: Clock cycles required for AES on different platforms (using key on-the-fly)

This algorithm is written in C and has been compiled and evaluated on some 32-bits architectures, including the ARM7TDMI and ARM9TDMI processors [4], the ST22 smart card processor by ST Microelectronics [5], and also on a general purpose Intel PentiumIII platform. These three platforms represent rather different architectures used in various systems and environments: embedded system, smart cards and PC, respectively. Please see the references to get more information on the processors. See the references for ARM and cAESar.

As explained before, the application of the Rijndael algorithm consists of 3 parts, key scheduling, encryption and decryption. This efficient algorithm provided with a speed gain in the MixColumns (during encryption) transformations, requiring only few changes to the key scheduling. A brief description of how these two things work is given in the following lines. A single Mix-Columns is a composition of sums and doublings in the field GF(28), plus some rotations of the elements of the column. A sum in GF(28) is a bitwise XOR of bytes and a doubling is a composition of a masking, a shift and a conditional bitwise XOR of bytes [3]. Since a column is composed by 4 elements of the field GF(28), some operations can be applied in parallel to the entire column, as the whole column can be accommodated in a single register of the CPU [3]. On a 32 bits platform a single MixColums requires 4 bitwise XORs plus one doubling of the four GF(28) elements and 3 rotations. The MixColumns must be applied to the 4 columns giving a total of 16 XORs, 4 doublings and 12 rotations. In the efficient algorithm no rotations are required for Mix Columns, so the speed is greatly increased.In general it is found that this efficient algorithm is much better than the Gladman decryption. The tables 1 and 2 show the results.

V. Conclusion

This paper surveyed several publications on AES, brought out the essential concepts that make the algorithm, some software implementations and issues were visited. AES algorithms implementation has been done with 128 bits in most of the observed cases. Implementations at 192 and 256 bits would make an interesting case study, even as the encryption levels are up a notch or two. The usage of look-up tables ends up being an memory intensive approach, and this is evident in the case of Smart Card systems. A novel approach to this issue of reducing the implementation complexity needs to be addressed.

References

[1] J. Daemen, V. Rijmen "AES Proposal: Rijndael," in *http://csrc.nist.gov/encryption/aes/* 1999.

[2] NIST - FIPS Standard "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," in *Federal Information Processing Standards Publication, n. 197*, 2001, November 26.

[3] Guido Bertoni1, Luca Breveglieri1, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin "Efficient Software Implementation of AES on 32-Bit Platforms," in *Cryptographic Hardware and Embedded Systems - CHES 2002*,, pp. 159-171, B.S. Kaliski Jr., .K. Ko, C. Paar.

[4] Kazumaro Aoki, Helger Lipmaa "Fast Implementations of AES Candidates," in *Submitted for publication - Third AES Candidate Conference, New York City, USA*, 2001, August 13-15.

[5] ARM Ltd. Website "www.arm.com,"

[6] STMicroelectronics website "www.st.com,"

[7] B. Gladman "A Specification for Rijndael, the AES Algorithm," in *http://fp.gladman.plus.com/*, 2001.

[8] Jif org J. Buchholz "Matlab Implementation of the Advanced Encryption Standard ," in *http://buchholz.hs-bremen.de*, 2001, December 19.