Efficient Software Implementation of AES on 32-bit Platforms

Guido Bertoni, Luca Breveglieri
Politecnico di Milano, Milano - Italy
Pasqualina “Lilli” Fragneto
AST-LAB of ST Microelectronics, Agrate B. - Italy
Marco Macchetti, Stefano Marchesin
ALARI - Università della Svizzera Italiana, Lugano - Switzerland

Table of Contents

Introduction

- A work for the efficient software implementation of AES.
- Optimised software implementation (in C) oriented to 32-bit platforms with low memory * (e.g. embedded systems).
- Evaluation of the time performances on various platforms: ARM, ST and Pentium.
- Comparison with the time performances of Gladman's C code.

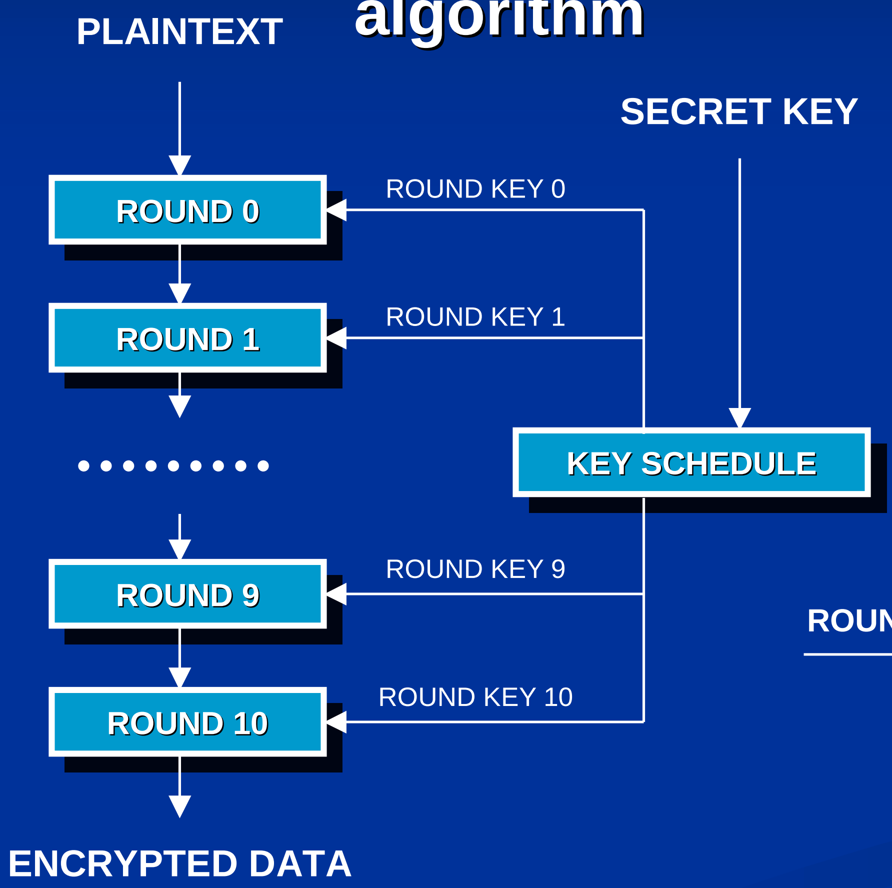
* The usage of look-up tables is limited: only the S-BOX and the inverse S-BOX transformations are tabularised (2×256 bytes).

Algorithm Description - General

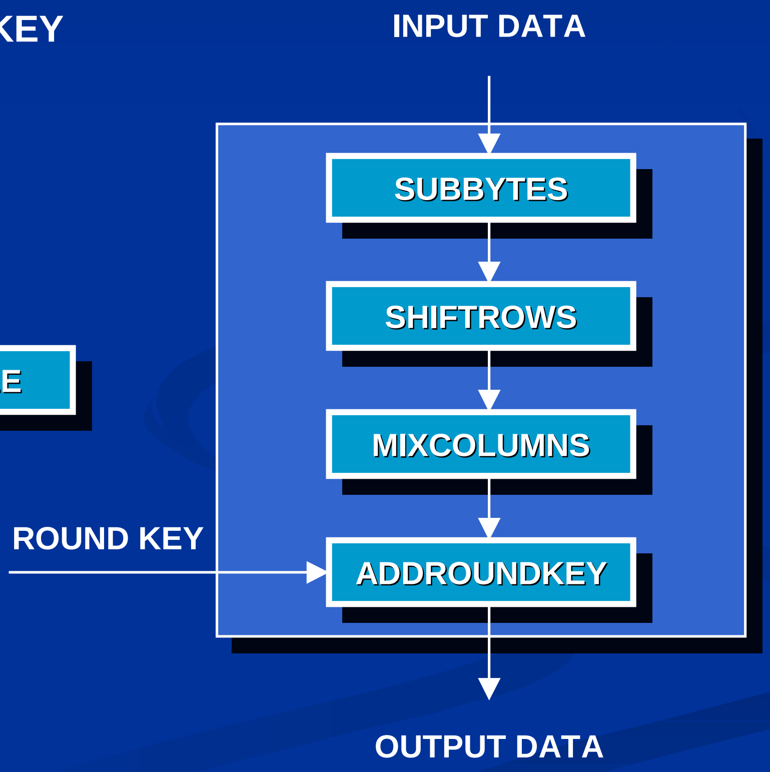
- Rijndael is the selected (NIST competition) algorithm for AES (Advanced Encryption Standard).
- It is a block cipher algorithm, operating on blocks of data.
- It needs a secret key, which is another block of data.
- Performs encryption and the inverse operation, decryption (using the same secret key).
- It reads an entire block of data, processes it in rounds and then outputs the encrypted (or decrypted) data.
- Each round is a sequence of four inner transformations.
- The AES standard specifies 128-bit data blocks and 128-bit, 192-bit or 256-bit secret keys.

Algorithm Description – Encrypt.

encryption algorithm

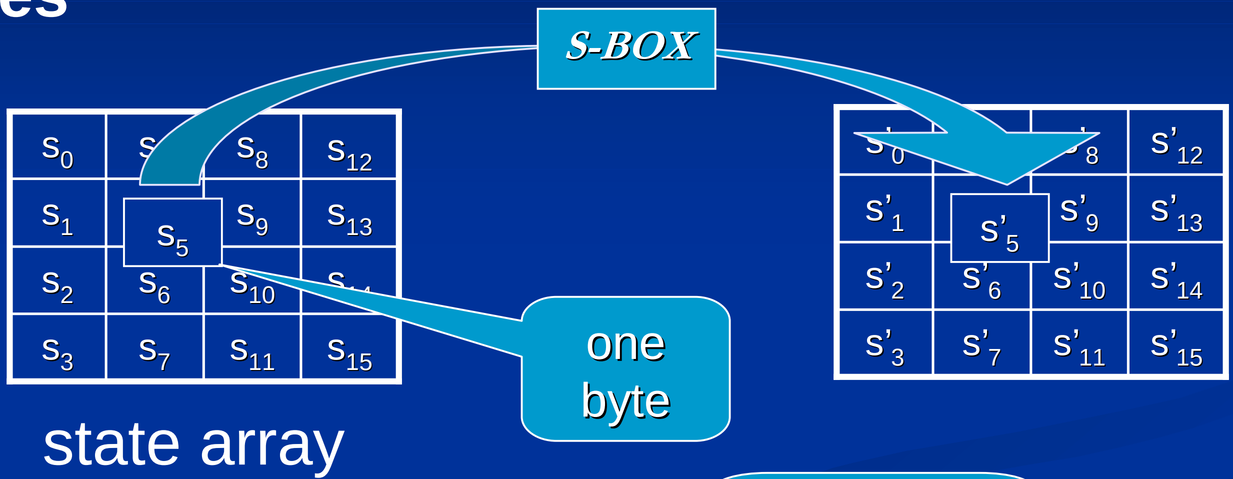


structure of a generic round

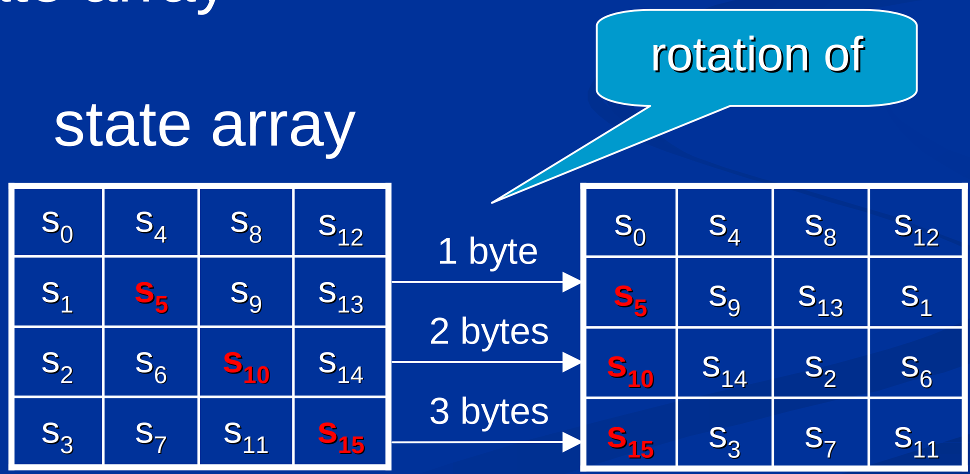


Algorithm Description – Encrypt.

SubBytes



ShiftRows



Algorithm Description – Encrypt.

MixColumns coeff.s matrix state array

$$\begin{array}{|c|c|c|c|}\hline s'_0 & s'_4 & s'_8 & s'_{12} \\ \hline s'_1 & s'_5 & s'_9 & s'_{13} \\ \hline s'_2 & s'_6 & s'_{10} & s'_{14} \\ \hline s'_3 & s'_7 & s'_{11} & s'_{15} \\ \hline\end{array} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \otimes \begin{array}{|c|c|c|c|}\hline s_0 & s_4 & s_8 & s_{12} \\ \hline s_1 & s_5 & s_9 & s_{13} \\ \hline s_2 & s_6 & s_{10} & s_{14} \\ \hline s_3 & s_7 & s_{11} & s_{15} \\ \hline\end{array}$$

field GF(2^8)

bit-wise XOR

polynomial multiplications

AddRoundKey state array round key

$$\begin{array}{|c|c|c|c|}\hline s'_0 & s'_4 & s'_8 & s'_{12} \\ \hline s'_1 & s'_5 & s'_9 & s'_{13} \\ \hline s'_2 & s'_6 & s'_{10} & s'_{14} \\ \hline s'_3 & s'_7 & s'_{11} & s'_{15} \\ \hline\end{array} = \begin{array}{|c|c|c|c|}\hline s_0 & s_4 & s_8 & s_{12} \\ \hline s_1 & s_5 & s_9 & s_{13} \\ \hline s_2 & s_6 & s_{10} & s_{14} \\ \hline s_3 & s_7 & s_{11} & s_{15} \\ \hline\end{array} \oplus \begin{array}{|c|c|c|c|}\hline k_0 & k_4 & k_8 & k_{12} \\ \hline k_1 & k_5 & k_9 & k_{13} \\ \hline k_2 & k_6 & k_{10} & k_{14} \\ \hline k_3 & k_7 & k_{11} & k_{15} \\ \hline\end{array}$$

Optimisation – The Idea

- To improve the time performances of AES, a transposed state array has been used.

state

s_0	s_4	s_8	s_{12}
s_1	s_5	s_9	s_{13}
s_2	s_6	s_{10}	s_{14}
s_3	s_7	s_{11}	s_{15}

$\Rightarrow$

transposed state

s_0	s_1	s_2	s_3
s_4	s_5	s_6	s_7
s_8	s_9	s_{10}	s_{11}
s_{12}	s_{13}	s_{14}	s_{15}

- Very simple idea, but yields interesting consequences!

Optimisation - Consequences

- The following round transformations are essentially invariant with respect to transposition (and their speed is unchanged):
  - SubBytes
  - ShiftRows
  - AddRoundKey (but the round keys must be transposed)
- Instead, the MixColumns transformation must be completely restructured.
- The new MixColumns is considerably sped-up by the transposition of the state.

Old MixColumns

- It is a matricial product (in GF(2^8)):

Mix Column number *c*
($0 \le c \le 3$)

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

state column

- In C language a macro is used:

```
fwd_mcol(x)
(f2 = FFmulX(x), f2^upr(x^f2,3)^upr(x,2)^upr(x,1))
```

Old MixColumns - Cost

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = 02 \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \oplus 03 \begin{bmatrix} s_{3,c} \\ s_{0,c} \\ s_{1,c} \\ s_{2,c} \end{bmatrix} \oplus \begin{bmatrix} s_{2,c} \\ s_{3,c} \\ s_{0,c} \\ s_{1,c} \end{bmatrix} \oplus \begin{bmatrix} s_{1,c} \\ s_{2,c} \\ s_{3,c} \\ s_{0,c} \end{bmatrix}$$

- The cost per column is: a single "doubling", 4 additions (XOR) and 3 rotations (all operations work on 32 bits).
- For a complete MixColumns transformation 4 "doublings", 16 additions (XOR) and 12 rotations are required.
- "doubling" means 4 multiplications in GF(2^8) of each byte of the 32-bit word.

New MixColumns

$$\begin{bmatrix} s'_0 & s'_4 & s'_8 & s'_{12} \\ s'_1 & s'_5 & s'_9 & s'_{13} \\ s'_2 & s'_6 & s'_{10} & s'_{14} \\ s'_3 & s'_7 & s'_{11} & s'_{15} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \otimes \begin{bmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{bmatrix}$$

$$\begin{bmatrix} s'_0 & s'_4 & s'_8 & s'_{12} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \end{bmatrix} \otimes \begin{bmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{bmatrix}$$

state row

Transposition is equivalent to processing the state array by rows, instead of processing it by columns!

New MixColumns

- The New MixColumns transformation is:

$$
\begin{aligned}
y_0 &= (\{02\} \bullet x_0) + (\{03\} \bullet x_1) + x_2 + x_3 \\
y_1 &= x_0 + (\{02\} \bullet x_1) + (\{03\} \bullet x_2) + x_3 \\
y_2 &= x_0 + x_1 + (\{02\} \bullet x_2) + (\{03\} \bullet x_3) \\
y_3 &= (\{03\} \bullet x_0) + x_1 + x_2 + (\{02\} \bullet x_3)
\end{aligned}
$$

- The symbols x_i and y_i ($0 \le i \le 3$) indicate the 32-bit rows of the state array before and after New MixColumns, respectively.
- The 32-bit word x_i accommodates 4 bytes coming from 4 different columns (and similarly for y_i).
- The operation $\{02\} \bullet x_i$ or "doublings" consists of 4 multiplications in $GF(2^8)$ of each byte of the 32bits word.

New MixColumns

- The transformation can be executed in three steps.
- It can be conceived as a sort of "double and add" algorithm.

$$y_0 = x_1 + x_2 + x_3$$
$$y_1 = x_0 + x_2 + x_3$$
$$y_2 = x_0 + x_1 + x_3$$
$$y_3 = x_0 + x_1 + x_2$$

$$x_0 = \{02\} \bullet x_0$$
$$x_1 = \{02\} \bullet x_1$$
$$x_2 = \{02\} \bullet x_2$$
$$x_3 = \{02\} \bullet x_3$$

$$y_0 \mathrel{+}= x_0 + x_1$$
$$y_1 \mathrel{+}= x_1 + x_2$$
$$y_2 \mathrel{+}= x_2 + x_3$$
$$y_3 \mathrel{+}= x_3 + x_0$$

Remainder:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

MixColumns – Cost Comparison

- The standard implementation of MixColumns requires:
  - 4 “doublings”,
  - 16 XOR’s and 12 rotations,
  - and one intermediate variable
- The “transposed” version of MixColumns requires:
  - 4 “doublings”,
  - 16 XOR’s and NO rotation,
  - and NO intermediate variable.
- Software time performances should improve!

Decryption

- Decryption uses the InvMixColumns transformation – inverse of MixColumns.
- Also InvMixColumns can be sped-up by the transposition of the state array.
- Transposition yields a higher speed-up for InvMixColumns than for MixColumns.
- This is due to the complex structure of the coefficient matrix of InvMixColumns.
- Mixcolumns' coeff.s: 01, 02 and 03 (hex).
- InvMixColumns' coeff.s: 09, 0b, 0d and 0e (hex).

Old InvMixColumns

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix}$$

- The entries of the coefficient matrix of InvMixColumns contain a larger number of 1's than those of MixColumns.
- Transposition exposes more parallelism and hence yields a significant speed-up.

New InvMixColumns

$$\begin{bmatrix} s'_0 & s'_4 & s'_8 & s'_{12} \end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \end{bmatrix} \otimes \begin{bmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{bmatrix}$$

Reminder:

$0e_{hex} = 1\ 1\ 1\ 0_{\ b}$

$0b_{hex} = 1\ 0\ 1\ 1_{\ b}$

$0d_{hex} = 1\ 1\ 0\ 1_{\ b}$

$09_{hex} = 1\ 0\ 0\ 1_{\ b}$

$$y_0 = x_1 + x_2 + x_3$$

$$x_0 = \{02\} \bullet x_0$$
$$x_1 = \{02\} \bullet x_1$$
$$x_2 = \{02\} \bullet x_2$$
$$x_3 = \{02\} \bullet x_3$$

$$y_0 \mathrel{+}= x_0 + x_1$$

$$x_0 = \{02\} \bullet (x_0 + x_2)$$
$$x_1 = \{02\} \bullet (x_1 + x_3)$$

$$y_0 \mathrel{+}= x_0$$

$$x_0 = \{02\} \bullet (x_0 + x_1)$$

$$y_0 \mathrel{+}= x_0$$

InvMixColumns – Cost Comparison

- The standard algorithm requires:
  - 12 “doublings”,
  - 32 XOR’s and 12 rotations,
  - and 4 intermediate variables.
- The “transposed” algorithm requires only:
  - 7 “doublings”,
  - 27 XOR’s and NO rotation,
  - and NO intermediate variable.
- Software time performances should improve!
- But time performances should improve in hardware as well!

Time Performances

- The time performances of the proposed algorithm have been tested on some 32-bit CPU's:
  - ARM 7 TDMI and ARM 9 TDMI, typical microcontrollers
  - ST 22, a CPU designed for smart card (by STM)
  - and PENTIUM III, a general purpose CPU
- The time performances are computed in CPU cycles, and are compared with those of Gladman's C code.
- Where Gladman is better, it is due to the time overhead required to transpose input and output data, to remain compliant with the standard.

Results (ARM)

CPU	Version	Key Schedule	Encryption	Decryption
ARM 7 TDMI	Transposed	634	1675	2074
	Gladman	449	1641	2763
ARM 9 TDMI	Transposed	499	1384	1764
	Gladman	333	1374	2439

Simulations have been executed by means of the ARM Development Suite ADS 1.1.

Results (ST 22 and P III)

CPU	Version	Key Schedule	Encryption	Decryption
ST 22	Transposed	0.22	0.51	0.60
	Gladman	0.13	0.61	1
P III	Transposed	370	1119	1395
	Gladman	396	1404	2152
	Gladman (look-up tab.)	202 / 306 (enc.) / (dec.)	362	381

ST 22 figures are normalized with respect to Gladman decryption.

Comparisons with Gladman

CPU	Key Schedule	Encryption	Decryption
ARM 7	41.20 %	2.07 %	-24.94 %
ARM 9	49.85 %	0.73 %	-27.68 %
ST 22	69.23 %	-16.39 %	-40.00 %
P III	-6.57 %	-20.30 %	-35.18 %

The comparison is performed setting to 100 % the time performances of Gladman's implementation for the corresponding function.

In red the cases where the transposed version has higher performances.

Conclusions and Further Developments

Conclusions:

- Study and optimization of AES.
- Some interesting time performance improvements in software.
- Part of this work is under patenting process.

Further Developments:

- Hardware implementations.

???? Any Question ????