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
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**

- **PLAINTEXT**
  - ROUND 0
  - ROUND 1
  - .........
  - ROUND 9
  - ROUND 10
- **SECRET KEY**
  - ROUND KEY 0
  - ROUND KEY 1
  - ROUND KEY 9
  - ROUND KEY 10
- **ENCRYPTED DATA**

**structure of a generic round**

- **INPUT DATA**
  - SUBBYTES
  - SHIFTROWS
  - MIXCOLUMNS
  - ADDROUNDKEY
- **OUTPUT DATA**

**KEY SCHEDULE**
Algorithm Description – Encrypt.

SubBytes

\[
\begin{array}{cccc}
  s_0 & s_1 & s_2 & s_3 \\
  s_4 & s_5 & s_6 & s_7 \\
  s_8 & s_9 & s_{10} & s_{11} \\
  s_{12} & s_{13} & s_{14} & s_{15}
\end{array}
\]

\[S-BOX\]

ShiftRows

\[
\begin{array}{cccc}
  s_0 & s_1 & s_2 & s_3 \\
  s_4 & s_5 & s_6 & s_7 \\
  s_8 & s_9 & s_{10} & s_{11} \\
  s_{12} & s_{13} & s_{14} & s_{15}
\end{array}
\]

state array

\[
\begin{array}{cccc}
  s'_0 & s'_1 & s'_2 & s'_3 \\
  s'_4 & s'_5 & s'_6 & s'_7 \\
  s'_8 & s'_9 & s'_{10} & s'_{11} \\
  s'_{12} & s'_{13} & s'_{14} & s'_{15}
\end{array}
\]

state array

one byte

rotation of

1 byte

2 bytes

3 bytes

08/09/2002 CHES 2002 Workshop – Redwood City (SF Bay), CA, USA pp. 5 / 23
## Algorithm Description – Encrypt.

### MixColumns

<table>
<thead>
<tr>
<th>$s'_0$</th>
<th>$s'_4$</th>
<th>$s'_8$</th>
<th>$s'_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s'_1$</td>
<td>$s'_5$</td>
<td>$s'_9$</td>
<td>$s'_{13}$</td>
</tr>
<tr>
<td>$s'_2$</td>
<td>$s'_6$</td>
<td>$s'_{10}$</td>
<td>$s'_{14}$</td>
</tr>
<tr>
<td>$s'_3$</td>
<td>$s'_7$</td>
<td>$s'_{11}$</td>
<td>$s'_{15}$</td>
</tr>
</tbody>
</table>

**coeff.s matrix**

\[
\begin{bmatrix}
02 & 03 & 01 & 01 \\
01 & 02 & 03 & 01 \\
01 & 01 & 02 & 03 \\
03 & 01 & 01 & 02 \\
\end{bmatrix}
\]

**state array**

<table>
<thead>
<tr>
<th>$s_0$</th>
<th>$s_4$</th>
<th>$s_8$</th>
<th>$s_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$s_5$</td>
<td>$s_9$</td>
<td>$s_{13}$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_6$</td>
<td>$s_{10}$</td>
<td>$s_{14}$</td>
</tr>
<tr>
<td>$s_3$</td>
<td>$s_7$</td>
<td>$s_{11}$</td>
<td>$s_{15}$</td>
</tr>
</tbody>
</table>

- field $GF(2^8)$
- bit-wise XOR

### AddRoundKey

<table>
<thead>
<tr>
<th>$s'_0$</th>
<th>$s'_4$</th>
<th>$s'_8$</th>
<th>$s'_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s'_1$</td>
<td>$s'_5$</td>
<td>$s'_9$</td>
<td>$s'_{13}$</td>
</tr>
<tr>
<td>$s'_2$</td>
<td>$s'_6$</td>
<td>$s'_{10}$</td>
<td>$s'_{14}$</td>
</tr>
<tr>
<td>$s'_3$</td>
<td>$s'_7$</td>
<td>$s'_{11}$</td>
<td>$s'_{15}$</td>
</tr>
</tbody>
</table>

**state array**

<table>
<thead>
<tr>
<th>$s_0$</th>
<th>$s_4$</th>
<th>$s_8$</th>
<th>$s_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$s_5$</td>
<td>$s_9$</td>
<td>$s_{13}$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$s_6$</td>
<td>$s_{10}$</td>
<td>$s_{14}$</td>
</tr>
<tr>
<td>$s_3$</td>
<td>$s_7$</td>
<td>$s_{11}$</td>
<td>$s_{15}$</td>
</tr>
</tbody>
</table>

**round key**

<table>
<thead>
<tr>
<th>$k_0$</th>
<th>$k_4$</th>
<th>$k_8$</th>
<th>$k_{12}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k_1$</td>
<td>$k_5$</td>
<td>$k_9$</td>
<td>$k_{13}$</td>
</tr>
<tr>
<td>$k_2$</td>
<td>$k_6$</td>
<td>$k_{10}$</td>
<td>$k_{14}$</td>
</tr>
<tr>
<td>$k_3$</td>
<td>$k_7$</td>
<td>$k_{11}$</td>
<td>$k_{15}$</td>
</tr>
</tbody>
</table>
Optimisation – The Idea

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

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

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 \( \text{GF}(2^8) \)):

\[
\begin{bmatrix}
  s_{0,c} \\
  s_{1,c} \\
  s_{2,c} \\
  s_{3,c}
\end{bmatrix} = \begin{bmatrix}
  02 & 03 & 01 & 01 \\
  01 & 02 & 03 & 01 \\
  01 & 01 & 02 & 03 \\
  03 & 01 & 01 & 02
\end{bmatrix} \begin{bmatrix}
  s_{0,c} \\
  s_{1,c} \\
  s_{2,c} \\
  s_{3,c}
\end{bmatrix}
\]

Mix Column number \( c \)  
\( 0 \leq c \leq 3 \)

- In C language a macro is used:

\[
fwd_mcol(x)\]

\( (f2 = \text{FFmulX}(x), f2^{\text{upr}}(x^{f2}, 3)^{\text{upr}}(x, 2)^{\text{upr}}(x, 1)) \)
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

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

Transposition is equivalent to processing the state array by \textbf{rows}, instead of processing it by \textbf{columns}!
The New MixColumns transformation is:

\[
\begin{align*}
y_0 &= (\{02\} \cdot x_0) + (\{03\} \cdot x_1) + x_2 + x_3 \\
y_1 &= x_0 + (\{02\} \cdot x_1) + (\{03\} \cdot x_2) + x_3 \\
y_2 &= x_0 + x_1 + (\{02\} \cdot x_2) + (\{03\} \cdot x_3) \\
y_3 &= (\{03\} \cdot x_0) + x_1 + x_2 + (\{02\} \cdot x_3)
\end{align*}
\]

- The symbols \(x_i\) and \(y_i\) (\(0 \leq i \leq 3\)) 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\} \cdot x_i or “doublings” consists of 4 multiplications in GF\( (2^8) \) of each byte of the 32bits word.
The transformation can be executed in three steps.

It can be conceived as a sort of “double and add” algorithm.

\[
\begin{align*}
  y_0 &= x_1 + x_2 + x_3 \\
  y_1 &= x_0 + x_2 + x_3 \\
  y_2 &= x_0 + x_1 + x_3 \\
  y_3 &= x_0 + x_1 + x_2
\end{align*}
\]

\[
\begin{align*}
  x_0 &= \{02\} \cdot x_0 \\
  x_1 &= \{02\} \cdot x_1 \\
  x_2 &= \{02\} \cdot x_2 \\
  x_3 &= \{02\} \cdot x_3
\end{align*}
\]

\[
\begin{align*}
  y_0 &= x_0 + x_1 \\
  y_1 &= x_1 + x_2 \\
  y_2 &= x_2 + x_3 \\
  y_3 &= x_3 + x_0
\end{align*}
\]

```
Remainder:
[02 03 01 01]
[01 02 03 01]
[01 01 02 03]
[03 01 01 02]
```
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

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

\[
\begin{bmatrix}
    s'_0 & s'_4 & s'_8 & s'_{12}
\end{bmatrix} = \begin{bmatrix} 0e & 0b & 0d & 09 \end{bmatrix} \otimes
\]

Reminder:

- \(0e_{\text{hex}} = 1 1 1 0\) \(_b\)
  - \(y_0 = x_1 + x_2 + x_3\)
  - \(x_0 = \{02\} \cdot (x_0 + x_2)\)
  - \(x_1 = \{02\} \cdot (x_1 + x_3)\)
  - \(y_0 += x_0\)
- \(0b_{\text{hex}} = 1 0 1 1\) \(_b\)
  - \(x_0 = \{02\} \cdot x_0\)
  - \(x_1 = \{02\} \cdot x_1\)
  - \(x_2 = \{02\} \cdot x_2\)
  - \(x_3 = \{02\} \cdot x_3\)
  - \(y_0 += x_0 + x_1\)
- \(0d_{\text{hex}} = 1 1 0 1\) \(_b\)
  - \(x_0 = \{02\} \cdot x_0\)
  - \(x_1 = \{02\} \cdot x_1\)
  - \(y_0 += x_0 + x_1\)
- \(09_{\text{hex}} = 1 0 0 1\) \(_b\)
  - \(x_0 = \{02\} \cdot (x_0 + x_1)\)
  - \(y_0 += x_0\)
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!
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)

<table>
<thead>
<tr>
<th>CPU</th>
<th>Version</th>
<th>Key Schedule</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM 7</td>
<td>Transposed</td>
<td>634</td>
<td>1675</td>
<td>2074</td>
</tr>
<tr>
<td>TDMI</td>
<td>Gladman</td>
<td>449</td>
<td>1641</td>
<td>2763</td>
</tr>
<tr>
<td>ARM 9</td>
<td>Transposed</td>
<td>499</td>
<td>1384</td>
<td>1764</td>
</tr>
<tr>
<td>TDMI</td>
<td>Gladman</td>
<td>333</td>
<td>1374</td>
<td>2439</td>
</tr>
</tbody>
</table>

Simulations have been executed by means of the ARM Development Suite ADS 1.1.
# Results (ST 22 and P III)

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

ST 22 figures are normalized with respect to Gladman decryption.

08/09/2002   CHES 2002 Workshop – Redwood City (SF Bay), CA, USA   pp. 21 / 23
# Comparisons with Gladman

<table>
<thead>
<tr>
<th>CPU</th>
<th>Key Schedule</th>
<th>Encryption</th>
<th>Decryption</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARM 7</td>
<td>41.20 %</td>
<td>2.07 %</td>
<td>-24.94 %</td>
</tr>
<tr>
<td>ARM 9</td>
<td>49.85 %</td>
<td>0.73 %</td>
<td>-27.68 %</td>
</tr>
<tr>
<td>ST 22</td>
<td>69.23 %</td>
<td>-16.39 %</td>
<td>-40.00 %</td>
</tr>
<tr>
<td>P III</td>
<td>-6.57 %</td>
<td>-20.30 %</td>
<td>-35.18 %</td>
</tr>
</tbody>
</table>

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:

- 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