Efficient Software Implementation of AES on 32-Bit Platforms*

Guido Bertoni[1], Luca Breveglieri[1], Pasqualina Fragneto[2], Marco Macchetti[3], and Stefano Marchesin[3]

[1] Politecnico di Milano , Milano, Italy
`{bertoni,breveglieri}@elet.polimi.it`
[2] STMicroelectronics, Agrate B.za MI, Italy
`{pasqualina.fragneto}@st.com`
[3] ALaRI Università della Svizzera Italiana, Lugano, Switzerland
`{macchetti,marchesin}@alari.ch`

Abstract. Rijndael is the winner algorithm of the AES contest; therefore it should become the most used symmetric-key cryptographic algorithm. One important application of this new standard is cryptography on smart cards. In this paper we present an optimisation of the Rijndael algorithm to speed up execution on 32-bits processors with memory constraints, such as those used in smart cards. First a theoretical analysis of the Rijndael algorithm and of the proposed optimisation is discussed, and then simulation results of the optimised algorithm on different processors are presented and compared with other reference implementations, as known from the technical literature.

1 Introduction

Rijndael, a block cipher algorithm designed by Vincent Rijmen and Joan Daemen [1], has been selected by NIST as the winner of the Advanced Encryption Standard competition [2]. Although the initial specification of the algorithm includes 128-bits, 192-bits and 256-bits as possible lengths for both the plaintext blocks and for the key material, the standard will consider only 128-bit as legal block length. In this paper we shall deal only with 128-bits blocks.

According to [3], the basic information unit for processing in the Rijndael algorithm is a byte, i.e. a sequence of eight bits treated as a single entity. The bit sequences corresponding to the input, the output and the cipher key are processed as arrays of bytes; these arrays are formed by dividing the sequences into groups of eight contiguous bits. Internally, the operations of the algorithm are performed on a two-dimensional array of bytes called "State". The State array consists of four rows of bytes, each row containing 4 bytes. The Rijndael cipher algorithm operates in rounds, a round being a fixed set of transformations to be applied to the State array. The number of these rounds is chosen depending on the key length and ranges from 10 to 14.

* Part of this work is under patenting process.

The Rijndael cipher algorithm is suited for an efficient implementation on a wide range of processors. The basic operations involved in the algorithm are very simple and the structure of the algorithm is straightforward. The Rijndael algorithm can be used as encryption standard in embedded systems. One important application field of the Rijndael algorithm is cryptography on smart cards. Currently DES is used in such systems, but industry is moving to replace it with the new AES algorithm.

The present paper considers optimised software implementations of the AES algorithm for several platforms, with particular regard to smart cards. Memory requirements are a fundamental issue when coding the Rijndael algorithm for smart cards. In fact, the algorithm can be considerably sped-up by precomputing part of the internal operations and storing the results in look-up tables. In our case this means that we want to achieve the best possible performance with a little amount of look-up tables, since in a smart card environment memory and silicon space are limited resources. We also have to consider that the smart card market is looking to 32-bits microprocessors as the new leading technology [5], although a large amount of manufactured cards still features 8-bits and 16-bits processors.

In this paper we shall describe a technique to enhance the time performances of the AES algorithm when running on 32-bits processors with strict silicon space constraints. This optimisation consists in a deep restructuring of the algorithm, which makes possible to organise the inner operations (i.e. the rounds and the byte-operations involved in each round) in a different way with respect to the standard formulation [12] of the algorithm. This restructuring can be nicely described in theoretical terms as a matrix transformation. The so-called State matrix of AES is transposed and the inner operations of each round are reorganised accordingly. Some inner operations are commuted with respect to other ones, and are then grouped in such a way as to fit well in processors having 32-bits words. This optimisation allows a better exploitation of the resources of the processor, and thus achieves also better time performances with respect to the standard formulation [12] of the algorithm.

The optimised version of the algorithm was coded in C for evaluation on various platforms, covering a wide range of possible applications. Namely, ARM (a typical processor for embedded systems), ST 22 (one of the most advanced 32-bits processors for smart cards) and also Intel Pentium (typical for general purpose systems) are considered. Simulations have been carried out for both Gladman's standard AES implementation in C code and our optimised AES implementation, on all the previously mentioned platforms. Some other implementations of AES, known from the literature, are also considered. The time performances obtained by simulation are summarized in tables, compared and discussed. In several of the examined cases, such results are highly favourable to our proposed optimised version of the AES algorithm.

This paper is organized as follows. Section 2 will provide a synthetic outline of the Rijndael algorithm. Section 3 will describe and analyse theoretically our new, optimised approach to the algorithm. In Section 4 we shall discuss our implementation in C code and we shall show simulation results and comparisons with other implementations on various platforms. Section 5 concludes the paper.

2 Description of the Rijndael Algorithm

This section provides a brief recall of the AES algorithm (Rijndael, [1][2][3]), useful for understanding the subsequent optimisations, which will be described in section 3.

The core data structure of AES is the State matrix: it is a 4 * 4 bytes matrix. As we have already said in the introduction, the Rijndael encryption algorithm operates in rounds; a round is a fixed set of transformations that are applied to the State matrix. The number of these rounds is chosen depending on the length of the key; it is necessary to perform 10, 12 or 14 rounds in the cases of 128, 192 or 256-bits keys, respectively. For each round of the AES algorithm a round key is derived from the original key; this process is called Key Scheduling.

The transformations that are applied in each round are four. According to [6], they correspond to the round key addition step, the non-linear step, the dispersion step and the diffusion step. They are described as follows.

AddRoundKey. In this transformation, a round key is added to the State matrix by a simple bitwise XOR operation, that is, a sum in the field $GF(2^8)$. Each round key is obtained from the key schedule.

SubBytes. This transformation is a non-linear byte substitution operating independently on each byte of the State matrix, using a substitution table (called S-BOX). This S-BOX, which is invertible, is constructed by composing two transformations in $GF(2^8)$, an inversion and an affine function.

ShiftRows. In this transformation, the bytes in the last three rows of the State matrix are cyclically shifted over different numbers of bytes (offsets). The first row, row 0, is not rotated. Row 1 is rotated to the left by 1 byte position; row 2 is rotated to the left by 2 byte positions; row 3 is rotated to the left by 3 byte positions.

MixColumns. This transformation operates on the State matrix in a column-by-column mode, treating each column as a four-term polynomial over $GF(2^8)$. These polynomials are multiplied modulo $(x^4 + 1)$ with a fixed polynomial a(x), given by the expression:

$$a(x) = 03 * x^3 + 01 * x^2 + 01 * x + 02$$

This polynomial is coprime to $(x^4 + 1)$, and therefore the transformation is invertible. This transformation can be written under the form of a matrix multiplication. Pose $s'_c(x) = a(x) \otimes s_c(x)$, for $0 \leq c \leq 3$, that is for all the 4 columns in the State matrix. As a result of this multiplication, the 4 bytes in a column c are replaced by the following ones (for c = 0, 1, 2 and 3):

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

where the * operator stands for a multiplication in $GF(2^8)$, with:

$$m(x) = x^8 + x^4 + x^3 + x + 1$$

as irreducible generator polynomial. Performing a complete round means simply applying these 4 transformations to the State matrix, in the following order:

$$round = \{SubBytes, ShiftRows, MixColumns, AddRoundKey\}$$

Performing the final round means simply applying to the State matrix the following transformations, in the order:

$$finalround = \{SubBytes, ShiftRows, AddRoundKey\}$$

The Rijndael encryption algorithm consists in an initial application of the AddRoundKey operation, followed by (number of rounds - 1) rounds and concluded with a final round. The Rijndael decryption algorithm operates by applying the inverse of all the transformations described above in reverse order, to return to the plaintext; for specific details, see [1].

3 A New Technique for Computing the Rijndael Algorithm

This section illustrates our optimised version of the Rijndael AES algorithm. Both the encryption and the decryption algorithms have been optimised. For reasons of brevity, attention is focused on encryption, giving only the necessary hints for understanding the way to optimise decryption as well. The task is divided in two parts: optimisation 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. In section 3.1 the algorithm optimisation is explained, while in section 3.2 the key scheduling is considered.

3.1 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. In particular, the internal transformations of a round could be implemented by using look-up tables. In our study, we have chosen to reserve a little amount of space for look-up tables: the choice is to tabularise only the S-BOX and the inverse S-BOX transformations. All the remaining operations are computed. In particular, this means that we must carry out several GF multiplications only by means of software techniques. All the primitives considered in our study behave in a peculiar way, operating on a transposed version of the State matrix. Of course, all the steps of the algorithm must be modified in order to preserve global functionality while operating on the transposed State matrix. In particular:

The **SubBytes** transformation is not modified, since it operates on single bytes, independently of their position in the State matrix.

The **ShiftRows** transformation does not shift the rows of the State matrix any longer; instead, it operates now in the same way on the columns.

The **MixColumns** transformation is deeply revised. Denote with x_i, for $0 \leq i \leq 3$, the 32-bits words (or columns) of the transposed State matrix before applying the MixColumns transformation, and denote with y_i, for $0 \leq i \leq 3$, 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:

$$\begin{aligned}
y_0 &= 02 * x_0 \oplus 03 * x_1 \oplus x_2 \oplus x_3 \\
y_1 &= x_0 \oplus 02 * x_1 \oplus 03 * x_2 \oplus x_3 \\
y_2 &= x_0 \oplus x_1 \oplus 02 * x_2 \oplus 03 * x_3 \\
y_3 &= 03 * x_0 \oplus x_1 \oplus x_2 \oplus 02 * x_3
\end{aligned}$$

The variables y_i and x_i 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. Note that here the symbol * does not denote an ordinary GF multiplication over factors (polynomials) of 32 bits. Instead, here the operator * denotes a set of 4 ordinary multiplications in the field $GF(2^8)$, performed in parallel on the 4 bytes of each 32-bits word. The generator polynomial used for representing the field $GF(2^8)$ is the standard one of AES.
A simple way to calculate all the above operations, composing the MixColumns transformation, is to use the y_i variable as an accumulator, and to use the x_i variable for storing the product of the initial values of x_i and of the 4 partial products: x_i, 2 * x_i, 4 * x_i and 8 * x_i. In the case of encryption the products to be used in the calculation are only two: 2^0 and 2^1. Therefore the MixColumns

transformation is computed in only 3 steps: a sum step, a doubling step and a final sum step. Table 1 shows the three steps.

Table 1. The three steps necessary to compute MixColumns.

First	Second	Third
$y_0 = x_1 \oplus x_2 \oplus x_3$	$x_0 = 02 * x_0$	$y_0 \oplus = x_0 \oplus x_1$
$y_1 = x_0 \oplus x_2 \oplus x_3$	$x_1 = 02 * x_1$	$y_1 \oplus = x_1 \oplus x_2$
$y_2 = x_0 \oplus x_1 \oplus x_3$	$x_2 = 02 * x_2$	$y_2 \oplus = x_2 \oplus x_3$
$y_3 = x_0 \oplus x_1 \oplus x_2$	$x_3 = 02 * x_3$	$y_3 \oplus = x_0 \oplus x_3$

In the case of decryption this double-and-add method is used in a more substantial way, and the steps necessary to compute the InvMixColumns transformation are 7: 4 sum steps and 3 doubling steps. In the case of InvMixColumns the double-and-add method can be improved by considering the particular values of the constant coefficients. Note that there are only two coefficients containing a bit 1 in the third position, namely the coefficients 0e and 0d. These coefficients are used in combination with the operands x_0 and x_2 contained both in the first row and in the third row, and are used in combination with the operands x_1 and x_3 contained both in the second row and in the fourth row. To save a doubling operation we can add these two operand pairs and store the result in x_0 and x_1, respectively. Instead of calculating the subexpression 04 * $x_0 \oplus 04 * x_2$, we can calculate the subexpression 04*$(x_0 \oplus x_2)$, since we do not need to store separately either addend. Moreover, note that every coefficient contains a bit 1 in the fourth position. The last calculation deals with this bit. Hence we can add the previously computed values x_0 and x_1, which are 04*$(x_0 \oplus x_2)$ and 04*$(x_1 \oplus x_3)$, respectively, and then double them, so that the subexpression 08*$(x_0 \oplus x_1 \oplus x_2 \oplus x_3)$ is obtained, which must be accumulated to every operand y_i.

The **AddRoundKey** transformation remains quite unchanged, since it consists in a simple bitwise XOR between the State matrix and the round keys. Of course, we have to ensure that round keys are transposed before being used, which is shown in the next section.

3.2 The Transposed State Matrix Key Scheduling

As we stated before, we need to transpose the round keys before using them. A trivial solution would be simply to apply the key scheduling and then to transpose every created round key. In this way we would introduce a huge computation overhead. One alternative, which is the implemented one, is to redesign the key scheduling directly in the ”transposed manner”.

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 i^{th} word of the actual round key with $K[i]$, where $0 \leq i \leq 3$, and the i^{th} word of the next round key with $K'[i]$. $K'[0]$ is computed by an XOR between $K[0]$, a constant rcon and $K[3]$, the latter being pre rotated and transformed using the S-BOX. The other three words $K'[1]$, $K'[2]$ and $K'[3]$ are calculated as $K'[i] = K[i] \oplus K'[i-1]$.

We must now rewrite this set of transformations to cope with transposed keys. We indicate with K_T the transposed round key we are working on, and with K'_T the new transposed key. Clearly we have:

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

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

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

The symbol $>> j$ ($<< j$) indicates a right (left) shift of j bit positions, with the insertion of j bits of value 0 in the most (least) significant positions, while *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.

The key schedule for the other key sizes, i.e. 192 and 256-bits, is similar; note however that in the 192-bits case the calculations to be performed are slightly more complex. In fact, in this case it is necessary to operate on blocks of 6 32-bits words, while the round keys to be generated in the transposed form occupy 4 words of 32-bits. To reduce the overhead our solution consists in using a full array of 8 words of 32-bits for the calculations, as in the 256-bit key case. The new round key (similarly to the old one) occupies the first 4 words and the top half of the last 4 words. Then the words are suitably shifted and stored. The key schedule for 256-bits keys is very similar to that for 128-bits keys and therefore it will not be discussed here.

4 Implementation and Time Performance Figures

We have a C/C++ implementation of our proposal of the AES optimised algorithm. A simulation campaign has been carried out, in order to evaluate time performances. We report the results of the AES optimised algorithm only in the case of a key size of 128 bits, but we have tested the proposed algorithm also with 192 and 256-bits key size. The results are quite the same as those obtained with 128-bits key size, only scaled by a constant factor due to the larger number of rounds required by these versions of the AES algorithm. The time performance gain of each round, with respect to the standard AES algorithm, remains the same in all the cases.

We have chosen to make a direct comparison with an equivalent version of AES by Dr. Brian Gladman [8], since Gladman has been involved in the definition of the AES standard and his version is well referenced. For the comparison with embedded processors (ARM7, ARM9 and ST22) few refinements have been added to Gladman original implementation in C language [8].

Our code has been compiled and evaluated on some 32-bits architectures, including the ARM7TDMI and ARM9TDMI processors [7], 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.

The ARM7TDMI processor is a widely used 32-bits RISC CPU. It contains sixteen 32 bits registers. No cache is available and the internal structure consists of a pipeline of 3 stages [7]. The ARM9 processor differs from ARM7 in the internal structure. It is designed accordingly to the Harvard architecture model with two different busses for data and instructions, respectively, and the core is pipelined in 5 different stages. There exist different implementations of ARM9, depending on the amount of cache memory. In our simulation the standard core ARM9TDMI has been used, without cache memory, but the time latency for the accesses to the memory is only of one clock cycle, while the code and the data are stored in two different memories.

The ST 22 processor is a 32-bits RISC processor particularly designed for smart cards. It is a dedicated processor and is not used for applications different from smart cards. The internal details are not completely known; however the processor contains some 32 bits registers and does not have cache memory [5].

PentiumIII is a typical processor for PC systems. It is a 32-bits processor with a large amount of available memory, organized in three levels, with 32 kbytes of cache memory at the first level (divided in two blocks of 16 kbytes for data and instructions, respectively), with at least 256 kbytes of cache memory at the second level, and with some megabytes of RAM as central memory [4].

In the Rijndael algorithm the encryption and decryption operations must be executed using the round keys. For each round a different round key is used. The process of deriving the round keys from the original key is called Key Scheduling. The round key can be computed either in advance (key unrolling) or the so-called ”on-the-fly” approach can be used. In the on-the-fly approach the various round keys are computed exactly when they are needed, and soon after they are discarded. In a software implementation of the Rijndael algorithm the "on-the-fly" approach is not useful in terms of speed since the key schedule must be performed for each data block to encrypt. Some particular systems, i.e. smart cards, do not allow using memory to store the unrolled key, both for reasons of security and of memory shortage. In those cases the on-the-fly approach is mandatory. We have implemented our optimisation of AES in two versions: the former one using key unrolling and the latter one using the on-the-fly approach. In order to implement decryption following the on-the-fly approach, it is advisable to store the last round key, from which the previous round keys can be reconstructed. This requires 16 bytes of memory. The alternative would be to generate all the round keys before starting decryption, and then to use them in reverse order; but this approach consumes more memory.

Since Gladman public C code for AES does not permit to use the on-the-fly approach, we have extended Gladman code implementing also the on-the-fly approach. In [12] it is possible to find some time performances of on-the-fly encryption algorithm on the ARM processor for smart cards applications (the so-called cAESar implementation, written in assembler language). The core processor specified in the paper illustrating the cAESar implementation is not clearly described but should be very similar to ARM7TDMI. As above mentioned, we assume to tabularise in a look-up table only the S-BOX transformation; all the other round transformations are computed as soon as they are needed, and the result is not stored for future use. The amount of used look-up tables is thus of 512 bytes for the S-BOX and the Inv-S-BOX, plus 10 bytes for the round constants rcon.

We have used the ARM simulator (ARMulator), the ST 22 development tools (courtesy by STM) and Microsoft Visual Studio 6.0. Table 2 shows the time performances of our proposal of AES and of Gladman's in the hypothesis of adopting key unrolling. Table 3 shows the time performances of our proposal of AES, Gladman's and cAESar (only on ARM) in the hypothesis of adopting the on-the-fly approach. In Table 2 Key Scheduling is listed separately from Encryption and Decryption, since it is computed completely in advance. The total time can be obtained by adding the key scheduling time to either the encryption or the decryption time. In Table 3 the key scheduling time is already included both in the encryption and the decryption time (since key scheduling is in this case interleaved with encryption or decryption).

As explained before, the application of the Rijndael algorithm consists of 3 parts, namely key scheduling, encryption and decryption. Our proposal provided us with a speed gain in the MixColumns (during encryption) and InvMix-Columns (during decryption) transformations, requiring only few changes to the key scheduling. In general these two operations work as follows. A single Mix-Columns is a composition of sums and doublings in the field $GF(2^8)$, plus some rotations of the elements of the column. A sum in $GF(2^8)$ is a bitwise XOR of bytes and a doubling is a composition of a masking, a shift and a conditional bitwise XOR of bytes. Since a column is composed by 4 elements of the field

$GF(2^8)$, 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. The InvMix-Columns works in a similar way. Gladman implementation of MixColumns and InvMixColumns are applied to each one of the 4 columns of the State matrix.

Examining the Gladman implementation of MixColums, which uses the standard representation of the State matrix, it is possible to count the number of required operations. On a 32 bits platform a single MixColums requires 4 bit-wise XORs plus one doubling of the four $GF(2^8)$ elements and 3 rotations. The MixColumns must be applied to the 4 columns giving a total of 16 XORs, 4 doublings and 12 rotations. Moreover, the original Gladman implementation requires an additional intermediate variable, which could be eliminated; therefore we shall not consider it.

Our optimisation of AES allows reducing the number of elementary operations. Using the transposed State matrix the rotations can be completely avoided (see sub-section 3.1) both in MixColumns and in InvMixColumns, thus yielding a speed gain. Moreover, in the InvMixColums the transposed State matrix allows to achieve a much higher speed gain. In fact, the transposed InvMixColums requires only 7 doublings and 27 sums, while in the standard (Gladman) implementation it is necessary to compute 12 doublings, 32 sums, 12 rotations and 4 intermediate variables are required. This means that using in decryption a transposed State matrix it is possible to obtain a reduction of 5 doublings, 5 sums and 12 rotations, and to eliminate completely the intermediate variables. This yields a further speed gain.

As a general comment, our proposal implementation of AES works much better than Gladman's in decryption for all platforms, both using key unrolling and key on-the-fly. In encryption the performances are instead more or less comparable. Table 2 and Table 3 need some more explanations, in order to relate the differences of time performances with the features of the adopted processor. Here they follow.

Table 2. Clock cycles required for AES on different platforms (using key unrolling).

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

Table 3. Clock cycles required for AES on different platforms (using key on-the-fly).

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

ARM. The report and the discussion of the time performances start by considering the ARM processor. On this system our optimised version of AES is slightly slower then Gladman's as for encryption. This can be theoretically justified as follows: our encryption algorithm should be advantageous with respect to Gladman's, because it saves some rotation steps. But if the processor has some dedicated machine instruction able to combine bitwise XOR and rotation (like ARM7TDMI), then the advantage of having fewer rotations tends to disappear. However, when the algorithm is written in C, our implementation frequently executes additions (XOR) involving 3 operands of 32 bits each one. This fact can lead to a non-optimal use of the pipeline of the processor, and hence to degrade the performances of our optimised version of AES with respect to Gladman's. However, in decryption our proposal is considerably more efficient than Gladman's. The core of the ARM9 system is similar to the one of ARM7, but is more powerful, thus giving better results in comparison to ARM7. However, the performance ratios between our optimised version and Gladman's version for ARM9 remain approximately the same as those for ARM7.

ST 22, by ST Microelectronics, is an advanced smart card CPU. This processor is a 32-bits RISC CPU, having some 32 bits registers but without cache memory. Simulations have been carried out by using the ST 22 C compiler (courtesy by ST Microelectronics). It must be noted that ST 22 is not able to combine addition and shift (or rotate) in a single machine instruction. This means that ST 22 lacks instructions fitting particularly to the Gladman AES implementation. This fact impacts negatively on encryption when performing the simulation of the Gladman version of AES, while encryption in our optimised version does not suffer any penalty. On the other side, Gladman key scheduling is not affected by this feature of the processor, as it does not use shifts and rotations, while our version of key scheduling is affected negatively. The time performances are still comparable as for key scheduling and encryption, while they are much in favour to our optimised version as for decryption. For reasons of privacy, the performance figures are normalised to 1, instead of reporting the absolute values.

PentiumIII. As a last comparison we report the time performances on the PentiumIII processor. Note that in such a system an implementation with a complete replacement of the calculations with look-up tables is faster then our implementation. Therefore we report 3 versions: "Gladman with tables" (all transformations are tabularised in look-up tables), Gladman and our proposal, both of which use look-up tables only for the S-BOX and Inv-S-BOX transformations. We remember that the Gladman implementation using tables for all the transformations requires a very large amount of memory, of about 20 kbytes. This usage of memory is not affordable in systems as smart cards.

It is possible to see that our optimised version of AES works well in most cases, with few exceptions. In general the largest performance gains are obtained for decryption. However, even in those cases where our AES version behaves worse than Gladman's, the difference is limited and is by far less relevant than the considerable performance gain obtained in the case of decryption. Moreover, our proposal includes some initial and final code for transposing the State matrix (that is, the data block). The initial transposition code requires about 20 cycles for all platforms, and similarly the final transposition code. The transposition code is required for making our AES optimisation equivalent to the standard one. However, should the data block to encrypt be supplied directly in transposed form, the transposition code could be stripped off.

5 Conclusions

An optimised version of the AES standard has been presented, coded in C and evaluated by simulation on various platforms: ARM for embedded systems, ST 22 for smart card applications and also Intel Pentium for general purpose systems. We have rewritten the basic transformations of the Rijndael cipher algorithm, using a transposed version of the so-called State matrix. We have shown that this relevant structural modification leads to a considerable improvement of time performances in decryption. As for encryption, the time performances of our version of AES and Gladman's are instead more or less the same. These results hold when a limited part of the AES computation is carried out using look-up tables. Namely we have supposed that only the S-BOX operation is tabularised and stored in a look-up memory.

In the other cases, that is when a considerable part of the algorithm is executed by using look-up tables, our proposal of AES and Gladman's become approximately equivalent. It must be noted, anyway, that using a large amount of memory for the look-up tables is too expensive for small systems, like for instance smart cards, and is considered unsafe, as monitoring the accesses to the look-up tables may in some cases allow to infer the key. Therefore, at least for these applications the choice of not resorting to look-up tables should be considered reasonable. Next research directions include for instance the hardware evaluation, in terms of silicon area and time latency, of our optimisation of AES, with respect to the standard implementation, and possibly the design of a suited instruction set targeted to a fast computation of AES.

References

1. J. Daemen, V. Rijmen , "AES Proposal: Rijndael", http://csrc.nist.gov/encryption/aes/, 1999
2. NIST, "Announcing the ADVANCED ENCRYPTION STANDARD (AES)," Federal Information Processing Standards Publication, n. 197, November 26, 2001.
3. B. Gladman, "A Specification for Rijndael, the AES Algorithm" http://fp.gladman.plus.com/, 2001.
4. Intel Ltd. website, www.intel.com
5. STMicroelectronics website, www.st.com
6. J. Daemen, V. Rijmen, "Efficient Block Ciphers for Smart-Cards", *Workshop on Smartcard Technology (Smartcard '99)*, pp. 29–36, USENIX Eds., 1999
7. ARM Ltd. website, www.arm.com
8. B. Gladman, available at http://fp.gladman.plus.com/cryptography_technology/rijndael/
9. D. Whiting, B. Schneier, S. Bellovin, "AES Key Agility Issues in High-Speed IPsec Implementations," *Counterpane Internet Security*, http://www.counterpane.com/aes-agility.html, 2000.
10. G. Hachez, F. Koeune, J. J. Quisquater, "cAESar Results: Implementation of Four AES Candidates on Two Smart-Cards", http://csrc.nist.gov/encryption/aes/, 1999
11. J. Daemen, V. Rijmen, "*The Block Cipher Rijndael*," in LNCS 1820, *Smart-Card Research and Applications*, pp. 288–296, J. Quisquater and B. Schneier, Eds., Springer-Verlag, 2000.
12. J. Daemen, V. Rijmen, "*Rijndael, the Advanced Encryption Standard*," Dr. Dobb's Journal, Vol. 26, No. 3, March 2001, pp. 137–139
13. M. Akkar, C. Giraud, "An Implementation of DES and AES, Secure against some Attacks," *Proceedings of CHES '01*, pp. 315–325, 2001.
14. M. McLoone, J. McCanny, "High Performance single-Chip FPGA Rijndael Algorithm Implementations," *Proceedings of CHES '01*, pp. 68–80, 2001.
15. V. Fischer, M. Drutarovsky, "Two Methods of Rijndael Implementation in Reconfigurable Hardware," *Proceedings of CHES '01*, pp. 81–96, 2001.
16. H. Kuo, I. Verbauwhede, "Architectural Optimization for a 1.82Gbits/sec VLSI Implementation of the AES Rijndael Algorithm," *Proceedings of CHES '01*, pp. 53–67, 2001.
17. A. Rudra, P.K. Dubey, C.S. Jutla, V. Kumar, J.R. Rao, P. Rohatgi, "Efficient Rijndael Encryption Implementation with Composite Field Arithmetic," *Proceedings of CHES '01*, pp. 175–188, 2001.
18. A. Dandalis, V.K. Prasanna, J.P.D. Rolim, "An adaptive cryptographic Engine for IPSec Architecutures" *Field-Programmable Custom Computing Machines, 2000 IEEE Symposium on*, pp. 132–141, 2000.