On the implementation of the Advanced Encryption Standard on a public-key crypto-coprocessor

Antonio Valverde Garcia, Jean-Pierre Seifert

Infineon Technologies, Security & ChipCard ICs
{antonio.valverde,jean-pierre.seifert}@infineon.com

Abstract

This paper describes how to implement the new Advanced Encryption Standard (AES) using a modular arithmetic crypto-coprocessor, typically used to speed up public-key crypto-systems. This idea provides a fast and secure AES implementation when a dedicated hardware AES module is not available. The advantages of using the modular arithmetic coprocessor when compared to a pure software implementation are:

- much higher execution performance,
- less memory usage, and
- optimized protection against side-channel attacks.

Keywords: AES, Crypto-Coprocessor, Implementation Issues, Secure Implementation.

1 Introduction

The Advanced Encryption Standard (AES) specifies a FIPS-approved (cf. [FIPS]) cryptographic algorithm that is used to safely protect electronic data. The AES algorithm is a symmetric block cipher that is able to encrypt (encipher) and decrypt (decipher) electronic data. The AES algorithm is capable of using cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data blocks of 128 bits. The new AES (also known as Rijndael, cf. [DR2]) is an algorithm designed to use only single byte operations. Therefore, it is an algorithm very suitable for 8-bit μ-processors with only a few kB RAM as commonly used in todays smart cards. However, Rijndael is also well suited for 32-bit μ-processors with more RAM and clearly for dedicated hardware implementations, cf. [Wo, WOL, SMTM]. An optimized implementation of the AES algorithm on an 8051 based μ-controller with a 128-bit key takes less than 1ms @ 15MHz and requires 48 bytes of directly addressable internal RAM to encrypt a 128 bit data block and a little bit more time to decrypt it. Even if this is enough for a large variety of applications, there are some others where the bit rate achieved with this implementation may not be enough (for instance in a contactless environment) or, there is a demand for a high physical attack resistancy. On the other hand, dedicated public-key coprocessors are fast arithmetic coprocessors that usually can handle non-modular and especially modular arithmetic on prime fields $\mathbb{F}_p$ and especially on fields of characteristic two $\mathbb{F}_2^d$, cf. [NR]. These coprocessors are designed to be very efficient for RSA and ECC algorithms, but they are clearly not intended to accelerate the computation of symmetric key algorithms like DES or AES. However, some of the operations usually implemented in a modular arithmetic coprocessor, specifically in those intended for elliptic curve cryptography, are still useful to implement the AES because some transformations of the AES are performed on a field $\mathbb{F}_2^d$. By performing these transformations within the coprocessor, we can reduce the execution time of the encryption and decryption algorithms, reduce the usage of internal RAM memory and protect the algorithm against various side-channel attacks [A, AK1, AK2, CJRR, CKN, DR1, DPV, Gu1, Gu2, KK, Koca], such like timing attacks [KQ, Koch], power attacks [AG, BS99, CCD, KJJ, Me], electromagnetic radiation attacks [SQ] or even fault attacks [ABFHS, BDL, BDHJNT, BS97, BS02, BMM, JLQ, JPY, JQBD, JQYY, KR, KWMK, Ma, Pai,

SA, YJ, YKLM1, YKLM2, ZM].

Although many implementations of Rijndael have been brought into the literature, since this algorithm has won the AES contest, none of these implementations so far used a public-key crypto-coprocessor. Therefore, we cannot compare our implementation with any other, and we recommend to look at cf. [Li] to get an overview of alternative implementations on other platforms.

In the course of this paper we first give some hints of the utility of our implementation in many smart card applications. In the next chapter we describe the minimum requirements for the needed coprocessor and give an example of its required architecture. Hereafter, we briefly describe the AES itself. The following chapter is the most important one, as it describes our proposed implementation technique used for the AES. Finally, some security considerations are discussed around the implementation presented here and some estimation figures on the performance of the implementation are also given.

2 Applications

2.1 Chipcard ICs

Chipcards are mainly used to identify and authenticate a card user to a system. The identification or authentication protocol is normally based on symmetric and asymmetric cryptography. Moreover, all the data transfers between the Chipcard and the Terminal are usually protected by a Message Authentication Code (MAC) calculated with a symmetric algorithm. Triple DES is the most currently symmetric algorithm used today in smart cards. However, the new encryption standard (AES) will progressively replace the Triple DES within the next years. Thus, a very efficient AES implementation will be required in those environments where the transaction time is required to be as short as possible, as in the case of contactless applications.

2.2 Security ICs

In the area of Security ICs, like a Trusted Platform Module or a SmartUSB μ-controller, the use of a modular arithmetic coprocessor for the AES implementation described here, will provide an encryption engine, fast enough and very secure for many applications, like bulk encryption, that with a standard software implementation could not be achieved.

2.3 Secure Storage ICs

The main product that can benefit of the AES implementation described here is the so called multimedia card also known as a secure storage IC. This card is typically composed of a large flash memory, a fast I/O interface and some security logic. When a small CPU and a modular arithmetic coprocessor is incorporated, the AES implementation described here will provide new features like data encryption and decryption which will allow to build new applications like fast and secure memory personalization. This kind of applications require a fast encryption/decryption engine, as fast as the I/O interface to avoid a penalty during the execution time of the application.

2.4 The required modular arithmetic coprocessor

The modular arithmetic coprocessor must have at least 6 registers (4 if only encryption is implemented), each of length greater or equal than 16 bytes each. On the other hand, the coprocessor shall be able to perform the following arithmetic and logical operations:

- Multiplication in $\mathbb{F}_2^d$, $d \geq 128$, of a long register by an 8-bit value,
- Addition modulo 2 ($\oplus$, i.e. exclusive OR) of two long registers,
- Right and Left shifting of a long register,
- Logical AND of two long registers (optional),
- Simultaneous rotation of 4 bytes words (optional).

The following figure gives an example how such a coprocessor could look like: Here, it is supposed that the standard CPU can directly operate on the data

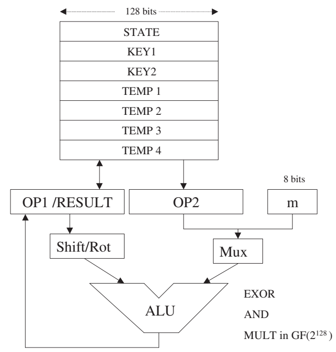

Figure 1: Example of the copprocessor's architecture.

stored on the coprocessors registers but that operations on these registers are much less efficient than on the standard CPU internal registers, because the data stored in those registers have to be transferred to the CPU through some external bus, as these data are usually organized as a so called XRAM.

3 Description of the Advanced Encryption Standard

In this section we briefly describe the Advanced Encryption Standard (AES). For a more detailed description we refer to [DR2].

AES encrypts plaintexts consisting of `lb` bytes, where $\mathtt{lb} = 16, 24$, or 32. The plaintext is organized as a $(4 \times \mathtt{Nb})$ array (a_{ij}), $0 \leq i < 4, 0 \leq j < \mathtt{Nb} - 1$, where $\mathtt{Nb} = 4, 6, 8$, depending on the value of `lb`. The n-th byte of the plaintext is stored in byte $a_{i,j}$ with $i = n \mod 4$, $j = \lfloor \frac{n}{4} \rfloor$.

AES uses a secret key, called *cipher key*, consisting of `lk` bytes, where $\mathtt{lk} = 16, 24$, or 32. Any combination of values `lb` and `lk` is allowed. The cipher key is organized in a $4 \times \mathtt{Nk}$ array (k_{ij}), $0 \leq i < 4, 0 \leq j \leq \mathtt{Nk} - 1$, where $\mathtt{Nk} = 4, 6, 8$, depending on the value of `lk`. The n-th key byte is stored in byte k_{ij} with $i = n \mod 4$, $j = \lfloor \frac{n}{4} \rfloor$.

The AES encryption process is composed of *rounds*. Except for the last round, each round consists of four transformations called `ByteSub`, `ShiftRow`, `MixColumn`, and `AddRoundKey`. In the last round the transformation `MixColumn` is omitted. The four transformations operate on intermediate results, called *states*. A state is a $4 \times \mathtt{Nb}$ array (a_{ij}) of bytes. Initially, the state is given by the plaintext to be encrypted. The number of rounds `Nr` is 10, 12, or 14, depending on $\max\{\mathtt{Nb}, \mathtt{Nk}\}$. In addition to the transformations performed in the `Nr` rounds there is an `AddRoundKey` applied to the plaintext prior to the first round. We call this the *initial* `AddRoundKey`.

Next, we are going to describe the transformations used in the AES encryption process. We begin with `AddRoundKey`.

The transformation `AddRoundKey` The input to the transformation `AddRoundKey` is a state (a_{ij}), $0 \leq i < 4, 0 \leq j < \mathtt{Nb}$, and a *round key*, which is an array of bytes $(\mathtt{rk}_{ij})$, $0 \leq i < 4, 0 \leq j < \mathtt{Nb}$. The output of `AddRoundKey` is the state $(b_{ij}), 0 \leq i < 4, 0 \leq j < \mathtt{Nb}$, where

$$b_{ij} = a_{ij} \oplus \mathtt{rk}_{ij}.$$

The round keys are obtained from the cipher key by expanding the cipher key array (k_{ij}) into an array (k_{ij}), $0 \leq i < 4, 0 \leq j \leq \mathtt{Nr} \cdot \mathtt{Nb}$, called the *expanded key*. The round key for the initial application of `AddRoundKey` is given by the first `Nb` columns of the expanded key. The round key for the application of `AddRoundKey` in the m-th round of AES is given by columns $m\mathtt{Nb}, \ldots, (m+1)\mathtt{Nb} - 1$ of the expanded key, $1 \leq m \leq \mathtt{Nr}$.

The transformation `ByteSub` Given a state (a_{ij}), $0 \leq i < 4, 0 \leq j < \mathtt{Nb}$, the transformation `ByteSub` applies an invertible function $S : \{0,1\}^8 \rightarrow \{0,1\}^8$ to each state byte a_{ij} separately. The exact nature of S is of no relevance for the implementation described later. We just mention that S is non-linear, and in fact, it is the only non-linear part of the AES encryption process. In practice, S is often realized by a substitution table or *S-box*.

The transformation `ShiftRow` The transformation `ShiftRow` cyclically shifts each row of a state (a_{ij}) separately to the left. Row 0 is not shifted. Rows $1, 2, 3$ are shifted by C_1, C_2, C_3 bytes, respectively, where the values of the C_i depend on `Nb`.

The transformation `MixColumn` The transformation `MixColumn` is crucial to the kind of our special implementation. The transformation `MixColumn` operates on the columns of a state separately. To each column a fixed linear transformation is applied. To do so, bytes are interpreted as elements in the field $\mathbb{F}_{2^8}$. As is usually done, we will denote elements in this field in hexadecimal notation. Hence $01, 02$ and 03 correspond to the bytes $00000001, 00000010$, and 00000011, respectively. Now `MixColumn` applies to each row of a state the linear transformation defined by the following matrix

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}. \tag{1}$$

One complete round of the AES encryption procedure is schematically shown in figure 2.

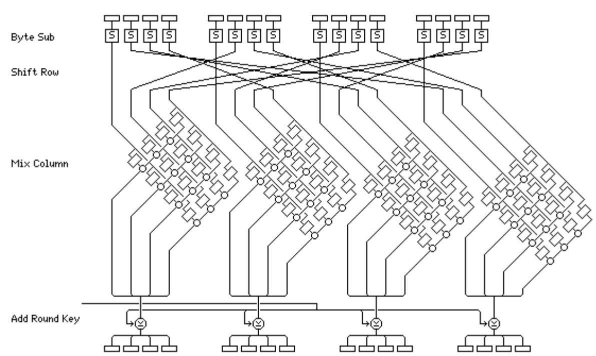

Figure 2: AES round description, cf. [Sa].

The operation `xtime` The multiplications in $\mathbb{F}_{2^8}$ necessary to compute the transformation `MixColumn` are of great importance to our implementation. Therefore we are going to describe them in more detail. First we need to say a few words about the representation of the field $\mathbb{F}_{2^8}$. In AES the field $\mathbb{F}_{2^8}$ is represented as

$$\mathbb{F}_{2^8} = \mathbb{F}_2[x]/(x^8 + x^4 + x^3 + x + 1). \tag{2}$$

That is, elements of $\mathbb{F}_{2^8}$ are polynomials over $\mathbb{F}_2$ of degree at most 7. The addition and multiplication of two polynomials is done modulo the polynomial $x^8 + x^4 + x^3 + x + 1$. Since this is an irreducible polynomial over $\mathbb{F}_2$, (2) defines a field. In this representation of $\mathbb{F}_{2^8}$ the byte $\mathbf{a} = (a_7, \ldots, a_1, a_0)$ corresponds to the polynomial $a_7x^7 + \cdots a_1x + a_0$. The multiplication of an element $\mathbf{a} = (a_7, \ldots, a_1, a_0)$ in $\mathbb{F}_{2^8}$ by $01, 02$, and 03 is realized by multiplying the polynomial $a_7x^7 + \cdots a_1x + a_0$ with the polynomials $1, x, x+1$, respectively, and reducing the result modulo $x^8 + x^4 + x^3 + x + 1$. Hence

$$\begin{aligned} 01 \cdot \mathbf{a} &= \mathbf{a} \\ 03 \cdot \mathbf{a} &= 02 \cdot \mathbf{a} + \mathbf{a}. \end{aligned}$$

We see that the only non-trivial multiplication needed to multiply a column of a state by the matrix in (1) is the multiplication by 02. Following the notation in [DR2] we denote the multiplication of byte $\mathbf{a}$ by 02 by $\mathtt{xtime}(\mathbf{a})$. The crucial observation is that $\mathtt{xtime}(\mathbf{a})$ is simply a shift of byte $\mathbf{a}$, followed in some cases by an xor of two bytes. More precisely, for $\mathbf{a} = (a_7, \ldots, a_0)$

$$\mathtt{xtime}(\mathbf{a}) = \begin{cases} (a_6, \ldots, a_0, 0), \\ \quad \text{if } a_7 = 0 \\ \\ (a_6, \ldots, a_0, 0) \oplus (0,0,0,1,1,0,1,1), \\ \quad \text{if } a_7 = 1 \end{cases}$$

This finishes our brief description of the AES encryption procedure.

In a pure software implementation of the algorithm on an 8051 based μ-controller these transformations are performed one after the other within the CPU using 48 bytes of directly addressable internal RAM, and taking roughly 12000 clock cycles to encrypt a 128 bit data block with a 128-bit key. The decryption algorithm takes about 30% more time than the cipher and requires at least the same bytes of internal RAM resources. This is due to the fact that the software implementation of the inverse `MixColumn` transformation used for decryption is less efficient than the `MixColumn` transformation used for encryption.

4 The public-key coprocessor based AES implementation

The formerly mentioned type of public-key coprocessor is actually useful to improve the performance of the following transformations of the AES cipher:

- `MixColumn`,
- inverse `MixColumn`,

- KeyExpansion and
- `AddRoundKey`.

Other transformations like the `ByteSub` and `ShiftRow` are performed inside the standard CPU and therefore remain unchanged. The reason of not using the coprocessor to accelerate these two last transformations is the following. The fastest way of performing the `ByteSub` transformation is by the use of a look-up table (the so called S-Box) containing 256 8-bit values. Because both of them, table indices and table contents are 8-bit values, the 8-bit CPU is the most suitable unit to perform this table access. Nevertheless, we advice the reader to carefully consult our section 5 on the physical security of the AES.

On the other hand, the `ShiftRow` transformation can be embedded into the `ByteSub` transformation in such a way that there is no performance loss. The next figure describes the execution parts executed in the CPU and the other ones executed within the coprocessor:

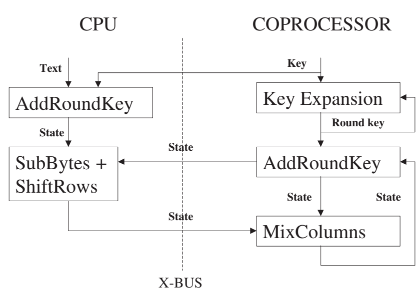

Figure 3: Execution of the AES transformations .

4.1 The `MixColumn` transformation

The multiplication of columns (`MixColumn`) is based on the `xtime` operation as defined within the AES specification. It multiplies a byte of the so called state by 2 modulo the irreducible polynomial $x^8 + x^4 + x^3 + x + 1$. This operation is usually performed on a byte by left shifting the byte (multiplication by 2) and, in case of overflow, xoring (addition modulo 2) with the hexadecimal value $0x1b$.

The MixColumn transformation requires matrix multiplication in the field $\mathbb{F}_2^8$. In an 8-bit CPU, this can be implemented in an efficient way for each column as follows:

$$\begin{aligned}
y_0 &= 02 * x_0 \oplus 03 * x_1 \oplus 01 * x_2 \oplus 01 * x_3 \\
y_1 &= 01 * x_0 \oplus 02 * x_1 \oplus 03 * x_2 \oplus 01 * x_3 \\
y_2 &= 01 * x_0 \oplus 01 * x_1 \oplus 02 * x_2 \oplus 03 * x_3 \\
y_3 &= 03 * x_0 \oplus 01 * x_1 \oplus 01 * x_2 \oplus 02 * x_3,
\end{aligned}$$

where $*$ represents the `xtime` operation. After reordering the equations we get:

$$\begin{aligned}
y_0 &= 02 * x_0 \oplus 03 * x_1 \oplus x_2 \oplus x3 \\
y_1 &= 02 * x_1 \oplus 03 * x_2 \oplus x_3 \oplus x0 \\
y_2 &= 02 * x_2 \oplus 03 * x_3 \oplus x_0 \oplus x1 \\
y_3 &= 02 * x_3 \oplus 03 * x_0 \oplus x_1 \oplus x2
\end{aligned}$$

The `xtime` operation can be performed inside the coprocessor on the 16 bytes of the state in parallel via the following formula:

$$\begin{aligned}
\texttt{xtime}(\text{state}) &= ((\text{state}\&m_2) << 1) \oplus \\
&\quad (((\text{state}\&m_1) >> 7) * m_3),
\end{aligned}$$

where $m_1 = 0x8080...80$ (16 bytes), $m_2 = 0x7f7f...7f$ (16 bytes) and $m_3 = 0x1b$. Here, $*$ denotes the multiplication operation in $\mathbb{F}_2^8$, $\oplus$ is the addition modulo 2, & the AND operation and $<<$ and $>>$ are the bit-left and bit-right shift operations respectively.

The `xtime` operation itself can be implemented inside the coprocessor with only two temporary registers, as shown below:

$$\begin{aligned}
t_1 &= \text{state}\&m_1 \\
t_1 &= t_1 >> 7 \\
t_1 &= t_1 * m_3 \\
t_2 &= \text{state}\&m_2 \\
t_2 &= t_2 << 1 \\
t_1 &= t_1 \oplus t_2
\end{aligned}$$

If the AND operation is not supported by the coprocessor, it has to be done in the standard CPU before loading the state into the coprocessor's register. Then, one has to load the result of the AND operations in both t_1 and t_2. Based on the previous definition of the `xtime` operation, the whole `MixColumn` transformation can be defined to operate on the 16 bytes of the state in parallel. The implementation is based on the previous definition of the xtime operation:

$$\begin{aligned} t_1 &= \texttt{xtime}(\text{state}) \\ t_2 &= t_1 \oplus \text{state} \\ t_2 &= \texttt{RotWord}(t_2) \\ t_1 &= t_1 \oplus t_2 \\ t_2 &= \texttt{RotWord}(\text{state}) \\ t_2 &= \texttt{RotWord}(t_2) \\ t_1 &= t_1 \oplus t_2 \\ t_2 &= \texttt{RotWord}(t_2) \\ \text{state} &= t_1 \oplus t_2 \end{aligned}$$

The total number of registers needed for the implementation of the MixColumn transformation in the coprocessor is 3, two temporal registers for the intermediate results and another for the state.

The RotWord operation as defined in the AES specification has to be performed on every 4 bytes of the state independently. If it is not supported by the coprocessor, this operation must be done by the standard CPU, accessing the internal coprocessor's registers.

4.2 The inverse MixColumn transformation

The inverse MixColumn transformation requires also a matrix multiplication in the field $\mathbb{F}_2^8$. In an 8-bit CPU, this can be implemented in an efficient way for each column as follows:

$$\begin{aligned} y_0 &= 0e * x_0 \oplus 0b * x_1 \oplus 0d * x_2 \oplus 09 * x_3 \\ y_1 &= 09 * x_0 \oplus 0e * x_1 \oplus 0b * x_2 \oplus 0d * x_3 \\ y_2 &= 0d * x_0 \oplus 09 * x_1 \oplus 0e * x_2 \oplus 0b * x_3 \\ y_3 &= 0b * x_0 \oplus 0d * x_1 \oplus 09 * x_2 \oplus 0e * x_3. \end{aligned}$$

After reordering the equations we get:

$$\begin{aligned} y_0 &= 0e * x_0 \oplus 0b * x_1 \oplus 0d * x_2 \oplus 09 * x_3 \\ y_1 &= 0e * x_1 \oplus 0b * x_2 \oplus 0d * x_3 \oplus 09 * x_0 \\ y_2 &= 0e * x_2 \oplus 0b * x_3 \oplus 0d * x_0 \oplus 09 * x_1 \\ y_3 &= 0e * x_3 \oplus 0b * x_0 \oplus 0d * x_1 \oplus 09 * x_2. \end{aligned}$$

As for the MixColumn, the inverse transformation (needed for decryption) can also be defined to operate on the 16 bytes of the state in parallel. The implementation is based on the previous definition of the xtime operation:

$$\begin{aligned} t_1 &= \texttt{xtime}(\text{state}) \\ t_2 &= \texttt{xtime}(t_1) \\ t_3 &= \texttt{xtime}(t_2) \\ t_4 &= t_1 \oplus t_2 \oplus t_3 \\ t_2 &= \text{state} \oplus t_2 \oplus t_3 \\ t_1 &= \text{state} \oplus t_1 \oplus t_3 \\ t_3 &= \text{state} \oplus t_3 \\ t_1 &= \texttt{RotWord}(t_1) \\ t_2 &= \texttt{RotWord}(\texttt{RotWord}(t_2)) \\ t_3 &= \texttt{RotWord}(\texttt{RotWord}(\texttt{RotWord}(t_3))) \\ \text{state} &= t_1 \oplus t_2 \oplus t_3 \oplus t_4 \end{aligned}$$

The total number of registers needed for the implementation of the inverse transformation in the coprocessor is 5, where 4 temporal registers are used for intermediate results and one other register for the state itself.

Another way to implement the inverse MixColumn transformation is by definition of the following two new operations:

$$\begin{aligned} \texttt{xtime}_4(\text{state}) &= ((\text{state}\&m_5) << 2) \oplus \\ &\quad (((\text{state}\&m_4) >> 6) * m_3) \\ \texttt{xtime}_8(\text{state}) &= ((\text{state}\&m_7) << 3) \oplus \\ &\quad (((\text{state}\&m_6) >> 5) * m_3), \end{aligned}$$

where $m_4 = 0xc0c0...c0$ (16 bytes), $m_5 = 0x3f3f...3f$ (16 bytes), $m_6 = 0xe0e0...e0$ (16 bytes), $m_7 = 0x1f1f...1f$ (16 bytes) and $m_3 = 0x1b$. Therefore, the implementation of the inverse MixColumn transformation can be redefined as follows:

$$\begin{aligned} t_1 &= \texttt{xtime}(\text{state}) \\ t_2 &= \texttt{xtime}_4(\text{state}) \\ t_3 &= \texttt{xtime}_8(\text{state}) \\ t_4 &= t_1 \oplus t_2 \oplus t_3 \\ t_2 &= \text{state} \oplus t_2 \oplus t_3 \\ t_1 &= \text{state} \oplus t_1 \oplus t_3 \\ t_3 &= \text{state} \oplus t_3 \\ t_1 &= \texttt{RotWord}(t_1) \\ t_2 &= \texttt{RotWord}(\texttt{RotWord}(t_2)) \\ t_3 &= \texttt{RotWord}(\texttt{RotWord}(\texttt{RotWord}(t_3))) \\ \text{state} &= t_1 \oplus t_2 \oplus t_3 \oplus t_4 \end{aligned}$$

The advantage of this second implementation is that the operations xtime, $\texttt{xtime}_4$ and $\texttt{xtime}_8$ can be calculated in parallel from the state, avoiding the sequence of the first implementation. M oreover, in the case that the AND operation is not available within the coprocessor, this second solution allows to precompute all the AND values within the standard CPU before loading the state into the coprocessor.

4.3 The Key Expansion

The 16, 24 or 32 bytes of the key (depending on the key length) are loaded into the Key register[1] of the coprocessor (Key1 and Key2 registers for 256-bit keys). Then, the next round key bytes are calculated with the following sequence of operations.

For a 128-bit key, perform the following sequence, and for each intermediate round do:

$$
\begin{aligned}
t_1 &= \texttt{Rcon} \oplus \texttt{ByteSub}(\texttt{RotWord}(\text{Key})) \\
\text{Key} &= \text{Key} \oplus t_1 \\
t_1 &= \text{Key} \\
t_1 &= t_1 >> 32 \\
\text{Key} &= \text{Key} \oplus t_1 \\
t_1 &= t_1 >> 32 \\
\text{Key} &= \text{Key} \oplus t_1 \\
t_1 &= t_1 >> 32 \\
\text{Key} &= \text{Key} \oplus t_1 .
\end{aligned}
$$

The `RotWord`, `ByteSub` operations are performed by the standard CPU on the 4 rightmost bytes of the Key register, then storing the result into the 4 leftmost bytes of t_1 and clearing the other bytes. `Rcon` is the 4-byte constant defined within the AES specification.

For a 256-bit key perform the following sequence, and for each intermediate "even" round do:

$$
\begin{aligned}
t_1 &= \texttt{Rcon} \oplus \texttt{ByteSub}(\texttt{RotWord}(\text{Key2})) \\
\text{Key}_1 &= \text{Key}_1 \oplus t_1 \\
t_1 &= \text{Key}_1 \\
t_1 &= t_1 >> 32 \\
\text{Key}_1 &= \text{Key}_1 \oplus t_1 \\
t_1 &= t_1 >> 32 \\
\text{Key}_1 &= \text{Key}_1 \oplus t_1 \\
t_1 &= t_1 >> 32 \\
\text{Key}_1 &= \text{Key}_1 \oplus t_1
\end{aligned}
$$

while every intermediate "odd" round (except round 1) is done as:

$$
\begin{aligned}
t_1 &= \texttt{ByteSub}(\text{Key1}) \\
\text{Key}_2 &= \text{Key}_2 \oplus t_1 \\
t_1 &= \text{Key}_2 \\
t_1 &= t_1 >> 32 \\
\text{Key}_2 &= \text{Key}_2 \oplus t_1 \\
t_1 &= t_1 >> 32 \\
\text{Key}_2 &= \text{Key}_2 \oplus t_1 \\
t_1 &= t_1 >> 32 \\
\text{Key}_2 &= \text{Key}_2 \oplus t_1
\end{aligned}
$$

For 196-bit keys, the sequence gets more complicated as in that case, new round key bytes are generated within a window of 6 bytes, but round key bytes should be delivered at a rate of 4 bytes. Basically, the process to generate the new round key bytes is similar to that for 128 bit keys, but yet longer registers (24 bytes long) and/or an additional temporary register might be needed.

Totally, the number of registers needed for the implementation of the Key Expansion transformation within the coprocessor is 2 (or at maximum 3 for keys longer than 16 bytes).

4.4 The `AddRoundKey` transformation

This transformation is performed by simply adding the state and the key modulo 2 inside the coprocessor:

$$\text{state} = \text{Key} \oplus \text{state}.$$

No temporal register is therefore needed. The Key register used will be Key1 or Key2 in the case of 256-bit keys, depending on the round number (Key1 for "even" rounds and Key2 for "odd" rounds).

[1] Mapping the encryption or decryption key to the Key register is straightforward, bytes $a_0, a_1, ..., a_{15}$ of the key are mapped one to one to bytes $k_0, k_1, ..., k_{15}$ of the Key register respectively.

5 Security Considerations

Although there is a large variety of possible physical attacks on the AES, cf. [AG, BS99, BS02, CJRR, DR1, KQ, KWMK, Me, YT], the `xtime` operation is clearly the most critical one in the AES algorithm, at least with respect to physical security or so called side-channel attacks. Namely, this operation involves a multiplication that is subject to timing and fault attacks (see [KQ, BS02]). We also stress that the recently developed fault based susceptibility due to [BS02] cannot be avoided by the simple dedicated fault-tolerant AES hardware as proposed by [KWMK].

However, thanks to the implementation described here, the aforesaid timing attack on the `xtime` operation doesnt work. This is due to the fact that the timing behaviour of modern crypto coprocessors is independent of its operands, which indeed avoids a timing attack vulnerability of our implementation.

Moreover, by performing the `xtime` operation on 16 bytes in parallel we make fault attacks very difficult to achieve, because we can use a fault in the calculation to flip a bit, but the flipped bit can be any one of the 128 bits of the state or temporary variable. Another critical part of the implementation described here might be the transfer of data through the so called X-BUS, the bus that connects the CPU and the coprocessor. This transfer of data is more significant when the AND and Rotate operations are not supported by the coprocessor and therefore have to be performed within the standard CPU. The bus contents could then be tampered via an electronic microscope, a focused ion beam, or could be revealed through measuring the power consumption or even by an electromagnetic field analysis.

Fortunately, this X-BUS is by some μ-controller ICs vendors protected by hardware and/or software mechanisms. Among the hardware countermeasures there are active shields or random bus scrambling techniques available on some existing high security m-controller ICs. Last generation of those high securitμ-controller ICs are designed using a special dual rail security logic, cf. [MAK, MACMT]. This logic not only ensures that both, a "0" and a "1" have the same Hamming weight, but also that changes between a logical "0" and a logical "1" are not distinguishable by an adversary.

As software measures some masking and encryption techniques could be applied to the data before being transferred, both in the CPU and in the coprocessor. However, these measures may have a significant impact on the overall performance of the algorithm, which makes the aforesaid hardware countermeasures the practically preferred choice.

6 Performance Estimation

An implementation of the AES encryption algorithm with a key length of 128 bits on Infineons SLE66P (8051 based) security controller family, cf. [Inf2], combined together with Infineons recently developed modular arithmetic coprocessor Spiridon, cf. [Inf1] (which has no AND or `RotWord` operation), is approximately two times faster than an optimized 8051 based implementation, and requires only 16 bytes of internal RAM memory. Most importantly, this implementation greatly benefits from the high physical attack security offered by the Spiridon coprocessor, which will be described in another publication.

However, we expect an implementation using an optimal modular arithmetic coprocessor with all the operations described at the beginning of the present paper by at least a factor of four faster than the implementation on Infineons Spiridon.

7 Acknowledgments

We would like to thank W. Fischer for valuable discussions on the material presented within this paper.

References

[A] R. Anderson, *Security Engineering*, John Wiley & Sons, New York, 2001.

[ABFHS] C. Aumüller, B. Bier, W. Fischer, P. Hofreiter, J.-P. Seifert, "Fault attacks on RSA: Concrete results and practical countermeasures", *Proc. of CHES '02*, Springer LNCS, pp. 261-276, 2002.

[AG] M. L. Akkar, C. Giraud, "An implementation of DES and AES, secure against some attacks", *Proc. of CHES '01*, Springer LNCS vol. 2162, pp. 315-324, 2001.

[AK1] R. Anderson, M. Kuhn, "Tamper Resistance – a cautionary note", *Proc. of 2nd USENIX Workshop on Electronic Commerce*, pp. 1-11, 1996.

[AK2] R. Anderson, M. Kuhn, "Low cost attacks attacks on tamper resistant devices", *Proc. of 1997 Security Protocols Workshop*, Springer LNCS vol. 1361, pp. 125-136, 1997.

[BDL] D. Boneh, R. A. DeMillo, R. Lipton, "On the Importance of Eliminating Errors in Cryptographic Computations" *Journal of Cryptology* **14**(2):101-120, 2001.

[BDHJNT] F. Bao, R. H. Deng, Y. Han, A. Jeng, A. D. Narasimbalu, T. Ngair, "Breaking public key cryptosystems on tamper resistant dives in the presence of transient faults", *Proc. of 1997 Security Protocols Workshop*, Springer LNCS vol. 1361, pp. 115-124, 1997.

[BS97] E. Biham, A. Shamir, "Differential fault analysis of secret key cryptosystems", *Proc. of CRYPTO '97*, Springer LNCS vol. 1294, pp. 513-525, 1997.

[BS99] E. Biham, A. Shamir, "Power analysis of the key scheduling of the AES candidates", *Proc. of the second AES conference*, pp. 115-121, 1999.

[BS02] J. Blömer, J.-P. Seifert, "Fault based cryptanalysis of the AES", e-Print Archive of the IACR, 2002, `http://www.iacr.org/`.

[BMM] I. Biehl, B. Meyer, V. Müller, "Differential fault attacks on elliptic curve cryptosystems", *Proc. of CRYPTO '00*, Springer LNCS vol. 1880, pp. 131-146, 2000.

[CCD] C. Clavier, J.-S. Coron, N. Dabbous, "Differential Power Analysis in the presence of Hardware Countermeasures", *Proc. of CHES '00*, Springer LNCS vol. 1965, pp. 252-263, 2000.

[CJRR] S. Chari, C. Jutla, J. R. Rao, P. J. Rohatgi, "A cautionary note regarding evaluation of AES candidates on smartcards", *Proc. of the second AES conference*, pp. 135-150, 1999.

[CKN] J.-S. Coron, P. Kocher D. Naccache, "Statistics and Secret Leakage", *Proc. of Financial Cryptography*, Springer LNCS, 2000.

[DR1] J. Daemen, V. Rijmen, "Resistance against implementation attacks: a comparative study", *Proc. of the second AES conference*, pp. 122-132, 1999.

[DR2] J. Daemen, V. Rijmen, *The Design of Rijndael*, Springer-Verlag, Berlin, 2002.

[DPV] J. Daemen, M. Peeters, G. Van Assche, "Bitslice ciphers and implementation attacks", *Proc. of Fast Software Encryption 2000*, Springer LNCS vol. 1978, pp. 134-149, 2001.

[FIPS] Federal Information Processing Standard, "Advanced Encryption Standard (AES)", National Institute of Standards and Technology (NIST) 2001, `http://csrc.nist.gov/publications/drafts/dfips-AES.pdf`.

[Gu1] P. Gutmann, "Secure deletion of data from magnetic and solid-state memory", *Proc. of 6th USENIX Security Symposium*, pp. 77-89, 1997.

[Gu2] P. Gutmann, "Data Remanence in Semiconductor Devices", *Proc. of 7th USENIX Security Symposium*, 1998.

[Inf1] Infineon Technologies AG, "Security & Chip Card ICs, Crypto2000, Modular Arithmetic Coprocessor, Preliminary Confidential Architecture Specification", v1.1, January 2001.

[Inf2] Infineon Technologies AG, "Security & Chip Card ICs, SLE 66Cxxx, Security Controller Family, Preliminary Confidential Data Book", September 2001.

[JLQ] M. Joye, A. K. Lenstra, J.-J. Quisquater, "Chinese remaindering based cryptosystem in the presence of faults", *Journal of Cryptology* **12**(4):241-245, 1999.

[JPY] M. Joye, P. Pailler, S.-M. Yen, "Secure Evaluation of Modular Functions", *Proc. of 2001 International Workshop on Cryptology and Network Security*, pp. 227-229, 2001.

[JQBD] M. Joye, J.-J. Quisquater, F. Bao, R. H. Deng, "RSA-type signatures in the presence of transient faults", *Cryptography and Coding*, Springer LNCS vol. 1335, pp. 155-160, 1997.

[JQYY] M. Joye, J.-J. Quisquater, S. M. Yen, M. Yung, "Observability analysis — detecting when improved cryptosystems fail", *Proc. of CT-RSA Conference 2002*, Springer LNCS vol. 2271, pp. 17-29, 2002.

[KR] B. Kaliski, M. J. B. Robshaw, "Comments on some new attacks on cryptographic devices", *RSA Laboratories Bulletin* **5**, July 1997.

[KK] O. Kömmerling, M. Kuhn, "Design Principles for Tamper-Resistant Smartcard Processors", *Proc. of the USENIX Workshop on Smartcard Technologies*, pp. 9-20, 1999.

[KQ] F. Koeune, J.-J. Quisquater, "A timing attack against Rijndael", *Université catholique de Louvain*, TR CG-1999/1, 6 pages , 1999.

[Koca] O. Kocar, "Hardwaresicherheit von Mikrochips in Chipkarten", *Datenschutz und Datensicherheit* **20**(7):421-424, 1996.

[Koch] P. Kocher, "Timing attacks on implementations of Diffie-Hellmann, RSA, DSS and other systems", *Proc. of CYRPTO '97*, Springer LNCS vol. 1109, pp. 104-113, 1997.

[KJJ] P. Kocher, J. Jaffe, J. Jun, "Differential Power Analysis", *Proc. of CYRPTO '99*, Springer LNCS vol. 1666, pp. 388-397, 1999.

[KWMK] R. Karri, K. Wu, P. Mishra, Y. Kim, "Concurrent error detection of fault-based side-channel cryptanalysis of 128-bit symmetric block ciphers", *Proc. of IEEE Design Automation Conference*, pp. 579-585, 2001.

[Li] H. Lipmaa, "AES candidates, a survey of implementations", `http://www.tcs.hut.fi/~helger /aes/rijndael.html`.

[Ma] D. P. Maher, "Fault induction attacks, tamper resistance, and hostile reverse engineering in perspective", *Proc. of Financial Cryptography*, Springer LNCS vol. 1318, pp. 109-121, 1997.

[Me] T. Messerges, "Securing the AES finalists against power analysis attacks", *Proc. of Fast Software Encryption 2000*, Springer LNCS vol. 1978, pp. 150-164, 2001.

[MAK] S. W. Moore, R. J. Anderson, M. G. Kuhn, "Improving Smartcard Security using Self-Timed Circuit Technology", *Fourth AciD-WG Workshop*, Grenoble, ISBN 2-913329-44-6, 2000.

[MACMT] S. W. Moore, R. J. Anderson, P. Cunningham, R. Mullins, G. Taylor, "Improving Smartcard Security using Self-Timed Circuit Technology", *Proc. of Asynch 2002*, IEEE Computer Society Press, 2002.

[NR] D. Naccache, D. M'Raihi, "Cryptographic smart cards", *IEEE Micro*, pp. 14-24, 1996.

[Pai] P. Pailler, "Evaluating differential fault analysis of unknown cryptosystems", *Gemplus Corporate Product R&D Division*, TR AP05-1998, 8 pages, 1999.

[Pe] I. Petersen, "Chinks in digital armor — Exploiting faults to break smartcard cryptosystems", *Science News* **151**(5):78-79, 1997.

[Sa] J. Savard, "The Advanced Encryption Standard (Rijndael)", `http://home.ecn.ab.ca/~jsavard /crypto/co040801.html`.

[SQ] D. Samyde, J.-J. Quisquater, "ElectroMagnetic Analysis (EMA): Measures and Countermeasures for Smart Cards", *Proc. of Int. Conf. on Research in Smart Cards, E-Smart 2001*, Springer LNCS vol. 2140, pp. 200-210, 2001.

[SMTM] A. Satoh, S. Morioka, K. Takano, S. Munetoh, "A compact Rijndael hardware architecture with S-Box optimization", *Proc. of ASIACRYPT '01*, Springer LNCS, pp. 241-256, 2001.

[SA] S. Skorobogatov, R. Anderson, "Optical Fault Induction Attacks", *Proc. of CHES '02*, Springer LNCS, pp. 2-12, 2002.

[Wo] J. Wolkerstorfer, "An ASIC implementation of the AES MixColumn-operation", Graz University of Technology, Institute for Applied Information Processing and Communications, Manuscript, 4 pages, 2001.

[WOL] J. Wolkerstorfer, E. Oswald, M. Lamberger, "An ASIC implementation of the AES S-Boxes", *Proc. of CT-RSA Conference 2002*, Springer LNCS vol. 2271, 2002.

[YJ] S.-M. Yen, M. Joye, "Checking before output may not be enough against fault-based cryptanalysis", *IEEE Trans. on Computers* **49**:967-970, 2000.

[YKLM1] S.-M. Yen, S.-J. Kim, S.-G. Lim, S.-J. Moon, "RSA Speedup with Residue Number System immune from Hardware fault cryptanalysis", *Proc. of the ICISC 2001*, Springer LNCS, 2001.

[YKLM2] S.-M. Yen, S.-J. Kim, S.-G. Lim, S.-J. Moon, "A countermeasure against one physical cryptanalysis may benefit another attack", *Proc. of the ICISC 2001*, Springer LNCS, 2001.

[YT] S.-M. Yen, S. Y. Tseng, "Differential power cryptanalysis of a Rijndael implementation", LCIS Technical Report TR-2K1-9, Dept. of Computer Science and Information Engineering, National Central University, Taiwan, 2001.

[ZM] Y. Zheng, T. Matsumoto, "Breaking real-world implementations of cryptosystems by manipulating their random number generation", *Proc. of the 1997 Symposium on Cryptography and Information Security*, Springer LNCS, 1997.