AES Proposal: Rijndael

Joan Daemen, Vincent Rijmen

Joan Daemen
Proton World Int.l
Zweefvliegtuigstraat 10
B-1130 Brussel, Belgium
daemen.j@protonworld.com

Vincent Rijmen
Katholieke Universiteit Leuven, ESAT-COSIC
K. Mercierlaan 94
B-3001 Heverlee, Belgium
vincent.rijmen@esat.kuleuven.ac.be

Table of Contents

Table of Figures

List of Tables

1. Introduction

In this document we describe the cipher Rijndael. First we present the mathematical basis necessary for understanding the specifications followed by the design rationale and the description itself. Subsequently, the implementation aspects of the cipher and its inverse are treated. This is followed by the motivations of all design choices and the treatment of the resistance against known types of attacks. We give our security claims and goals, the advantages and limitations of the cipher, ways how it can be extended and how it can be used for functionality other than block encryption/decryption. We conclude with the acknowledgements, the references and the list of annexes.

Patent Statement: Rijndael or any of its implementations is not and will not be subject to patents.

1.1 Document history

This is the second version of the Rijndael documentation. The main difference with the first version is the correction of a number of errors and inconsistencies, the addition of a motivation for the number of rounds, the addition of some figures in the section on differential and linear cryptanalysis, the inclusion of Brian Gladman's performance figures and the specification of Rijndael extensions supporting block and key lengths of 160 and 224 bits.

2. Mathematical preliminaries

Several operations in Rijndael are defined at byte level, with bytes representing elements in the finite field GF(2^8). Other operations are defined in terms of 4-byte *words*. In this section we introduce the basic mathematical concepts needed in the following of the document.

2.1 The field GF(2^8)

The elements of a finite field [LiNi86] can be represented in several different ways. For any prime power there is a single finite field, hence all representations of GF(2^8) are isomorphic. Despite this equivalence, the representation has an impact on the implementation complexity. We have chosen for the classical polynomial representation.

A byte b, consisting of bits $b_7\ b_6\ b_5\ b_4\ b_3\ b_2\ b_1\ b_0$, is considered as a polynomial with coefficient in {0,1}:

$$b_7 x^7 + b_6 x^6 + b_5 x^5 + b_4 x^4 + b_3 x^3 + b_2 x^2 + b_1 x + b_0$$

Example: the byte with hexadecimal value '57' (binary `01010111`) corresponds with polynomial

$$x^6 + x^4 + x^2 + x + 1\ .$$

2.1.1 Addition

In the polynomial representation, the sum of two elements is the polynomial with coefficients that are given by the sum modulo 2 (i.e., 1 + 1 = 0) of the coefficients of the two terms.

Example: '57' + '83' = 'D4', or with the polynomial notation:

$$(x^6 + x^4 + x^2 + x + 1) + (x^7 + x + 1) = x^7 + x^6 + x^4 + x^2 .$$

In binary notation we have: "01010111" + "10000011" = "11010100". Clearly, the addition corresponds with the simple bitwise EXOR (denoted by $\oplus$) at the byte level.

All necessary conditions are fulfilled to have an Abelian group: internal, associative, neutral element ('00'), inverse element (every element is its own additive inverse) and commutative. As every element is its own additive inverse, subtraction and addition are the same.

2.1.2 Multiplication

In the polynomial representation, multiplication in GF(2^8) corresponds with multiplication of polynomials modulo an irreducible binary polynomial of degree 8. A polynomial is irreducible if it has no divisors other than 1 and itself. For Rijndael, this polynomial is called $m(x)$ and given by

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

or '11B' in hexadecimal representation.

Example: '57' • '83' = 'C1', or:

$$\begin{aligned}(x^6 + x^4 + x^2 + x + 1)\ (x^7 + x + 1) \quad &= \quad x^{13} + x^{11} + x^9 + x^8 + x^7 + \\ &\qquad x^7 + x^5 + x^3 + x^2 + x + \\ &\qquad x^6 + x^4 + x^2 + x + 1 \\ &= \quad x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1\end{aligned}$$

$$\begin{aligned}x^{13} + x^{11} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + 1 \text{ modulo } x^8 + x^4 + x^3 + x + 1 \\ = \quad x^7 + x^6 + 1\end{aligned}$$

Clearly, the result will be a binary polynomial of degree below 8. Unlike for addition, there is no simple operation at byte level.

The multiplication defined above is associative and there is a neutral element ('01'). For any binary polynomial $b(x)$ of degree below 8, the extended algorithm of Euclid can be used to compute polynomials $a(x)$, $c(x)$ such that

$$b(x)a(x) + m(x)c(x) = 1 .$$

Hence, $a(x) \bullet b(x) \bmod m(x) = 1$ or

$$b^{-1}(x) = a(x) \bmod m(x)$$

Moreover, it holds that $a(x) \bullet (b(x) + c(x)) = a(x) \bullet b(x) + a(x) \bullet c(x)$.

It follows that the set of 256 possible byte values, with the EXOR as addition and the multiplication defined as above has the structure of the finite field GF(2^8).

2.1.3 Multiplication by *x*

If we multiply $b(x)$ by the polynomial x, we have:

$$b_7 x^8 + b_6 x^7 + b_5 x^6 + b_4 x^5 + b_3 x^4 + b_2 x^3 + b_1 x^2 + b_0 x$$

$x \bullet b(x)$ is obtained by reducing the above result modulo $m(x)$. If $b_7 = 0$, this reduction is the identity operation, If $b_7 = 1$, $m(x)$ must be subtracted (i.e., EXORed). It follows that multiplication by x (hexadecimal '02') can be implemented at byte level as a left shift and a subsequent conditional bitwise EXOR with '1B'. This operation is denoted by `b = xtime(a)`. In dedicated hardware, `xtime` takes only 4 EXORs. Multiplication by higher powers of x can be implemented by repeated application of `xtime`. By adding intermediate results, multiplication by any constant can be implemented.

Example: '57' • '13' = 'FE'

'57' • '02' = `xtime(57)` = 'AE'

'57' • '04' = `xtime(AE)` = '47'

'57' • '08' = `xtime(47)` = '8E'

'57' • '10' = `xtime(8E)` = '07'

'57' • '13' = '57' • ('01' ⊕ '02' ⊕ '10') = '57' ⊕ 'AE' ⊕ '07' = 'FE'

2.2 Polynomials with coefficients in GF(2^8)

Polynomials can be defined with coefficients in GF(2^8). In this way, a 4-byte vector corresponds with a polynomial of degree below 4.

Polynomials can be added by simply adding the corresponding coefficients. As the addition in GF(2^8) is the bitwise EXOR, the addition of two vectors is a simple bitwise EXOR.

Multiplication is more complicated. Assume we have two polynomials over GF(2^8):

$$a(x) = a_3 x^3 + a_2 x^2 + a_1 x + a_0 \text{ and } b(x) = b_3 x^3 + b_2 x^2 + b_1 x + b_0.$$

Their product $c(x) = a(x)b(x)$ is given by

$$c(x) = c_6 x^6 + c_5 x^5 + c_4 x^4 + c_3 x^3 + c_2 x^2 + c_1 x + c_0 \quad \text{with}$$

$$c_0 = a_0 \bullet b_0$$

$$c_1 = a_1 \bullet b_0 \oplus a_0 \bullet b_1$$

$$c_2 = a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2$$

$$c_3 = a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3$$

$$c_4 = a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3$$

$$c_5 = a_3 \bullet b_2 \oplus a_2 \bullet b_3$$

$$c_6 = a_3 \bullet b_3$$

Clearly, $c(x)$ can no longer be represented by a 4-byte vector. By reducing $c(x)$ modulo a polynomial of degree 4, the result can be reduced to a polynomial of degree below 4. In Rijndael, this is done with the polynomial $M(x) = x^4 + 1$. As

$$x^j \bmod x^4 + 1 = x^{j \bmod 4},$$

the modular product of $a(x)$ and $b(x)$, denoted by $d(x) = a(x) \otimes b(x)$ is given by

$$d(x) = d_3 x^3 + d_2 x^2 + d_1 x + d_0 \qquad \text{with}$$

$$d_0 = a_0 \bullet b_0 \oplus a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3$$

$$d_1 = a_1 \bullet b_0 \oplus a_0 \bullet b_1 \oplus a_3 \bullet b_2 \oplus a_2 \bullet b_3$$

$$d_2 = a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2 \oplus a_3 \bullet b_3$$

$$d_3 = a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3$$

The operation consisting of multiplication by a fixed polynomial $a(x)$ can be written as matrix multiplication where the matrix is a circulant matrix. We have

$$\begin{bmatrix} d_0 \\ d_1 \\ d_2 \\ d_3 \end{bmatrix} = \begin{bmatrix} a_0 & a_3 & a_2 & a_1 \\ a_1 & a_0 & a_3 & a_2 \\ a_2 & a_1 & a_0 & a_3 \\ a_3 & a_2 & a_1 & a_0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Note: $x^4 + 1$ is not an irreducible polynomial over GF(2^8), hence multiplication by a fixed polynomial is not necessarily invertible. In the Rijndael cipher we have chosen a fixed polynomial that does have an inverse.

2.2.1 Multiplication by *x*

If we multiply $b(x)$ by the polynomial x, we have:

$$b_3 x^4 + b_2 x^3 + b_1 x^2 + b_0 x$$

$x \otimes b(x)$ is obtained by reducing the above result modulo $1 + x^4$. This gives

$$b_2 x^3 + b_1 x^2 + b_0 x + b_3$$

The multiplication by x is equivalent to multiplication by a matrix as above with all a_i ='00' except a_1 ='01'. Let $c(x) = x \otimes b(x)$. We have:

$$\begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \end{bmatrix} = \begin{bmatrix} 00 & 00 & 00 & 01 \\ 01 & 00 & 00 & 00 \\ 00 & 01 & 00 & 00 \\ 00 & 00 & 01 & 00 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

Hence, multiplication by x, or powers of x, corresponds to a cyclic shift of the bytes inside the vector.

3. Design rationale

The three criteria taken into account in the design of Rijndael are the following:

- Resistance against all known attacks;
- Speed and code compactness on a wide range of platforms;
- Design simplicity.

In most ciphers, the round transformation has the Feistel Structure. In this structure typically part of the bits of the intermediate State are simply transposed unchanged to another position. The round transformation of Rijndael does *not* have the Feistel structure. Instead, the round transformation is composed of three distinct invertible uniform transformations, called *layers*. By "uniform", we mean that every bit of the State is treated in a similar way.

The specific choices for the different layers are for a large part based on the application of the Wide Trail Strategy [Da95] (see Annex), a design method to provide resistance against linear and differential cryptanalysis (see Section 8.2). In the Wide Trail Strategy, every layer has its own function:

The linear mixing layer:	guarantees high diffusion over multiple rounds.
The non-linear layer:	parallel application of S-boxes that have optimum worst-case nonlinearity properties.
The key addition layer:	A simple EXOR of the Round Key to the intermediate State.

Before the first round, a key addition layer is applied. The motivation for this initial key addition is the following. Any layer after the last key addition in the cipher (or before the first in the context of known-plaintext attacks) can be simply peeled off without knowledge of the key and therefore does not contribute to the security of the cipher. (e.g., the initial and final permutation in the DES). Initial or terminal key addition is applied in several designs, e.g., IDEA, SAFER and Blowfish.

In order to make the cipher and its inverse more similar in structure, the linear mixing layer of the last round is different from the mixing layer in the other rounds. It can be shown that this does not improve or reduce the security of the cipher in any way. This is similar to the absence of the swap operation in the last round of the DES.

4. Specification

Rijndael is an iterated block cipher with a variable block length and a variable key length. The block length and the key length can be independently specified to 128, 192 or 256 bits.

Note: this section is intended to explain the cipher structure and not as an implementation guideline. For implementation aspects, we refer to Section 5.

4.1 The State, the Cipher Key and the number of rounds

The different transformations operate on the intermediate result, called the *State*:

Definition: the intermediate cipher result is called the *State.*

The State can be pictured as a rectangular array of bytes. This array has four rows, the number of columns is denoted by **Nb** and is equal to the block length divided by 32.

The Cipher Key is similarly pictured as a rectangular array with four rows. The number of columns of the Cipher Key is denoted by **Nk** and is equal to the key length divided by 32.

These representations are illustrated in Figure 1.

In some instances, these blocks are also considered as one-dimensional arrays of 4-byte vectors, where each vector consists of the corresponding column in the rectangular array representation. These arrays hence have lengths of 4, 6 or 8 respectively and indices in the ranges 0..3, 0..5 or 0..7. 4-byte vectors will sometimes be referred to as words.

Where it is necessary to specify the four individual bytes within a 4-byte vector or word the notation (a, b, c, d) will be used where a, b, c and d are the bytes at positions 0, 1, 2 and 3 respectively within the column, vector or word being considered.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Figure 1: Example of State (with Nb = 6) and Cipher Key (with Nk = 4) layout.

The input and output used by Rijndael at its external interface are considered to be one-dimensional arrays of 8-bit bytes numbered upwards from 0 to the 4***Nb**–1. These blocks hence have lengths of 16, 24 or 32 bytes and array indices in the ranges 0..15, 0..23 or 0..31. The Cipher Key is considered to be a one-dimensional arrays of 8-bit bytes numbered upwards from 0 to the 4***Nk**–1. These blocks hence have lengths of 16, 24 or 32 bytes and array indices in the ranges 0..15, 0..23 or 0..31.

The cipher input bytes (the “plaintext” if the mode of use is ECB encryption) are mapped onto the state bytes in the order $a_{0,0}$, $a_{1,0}$, $a_{2,0}$, $a_{3,0}$, $a_{0,1}$, $a_{1,1}$, $a_{2,1}$, $a_{3,1}$, $a_{4,1}$ ... , and the bytes of the Cipher Key are mapped onto the array in the order $k_{0,0}$, $k_{1,0}$, $k_{2,0}$, $k_{3,0}$, $k_{0,1}$, $k_{1,1}$, $k_{2,1}$, $k_{3,1}$, $k_{4,1}$ ... At the end of the cipher operation, the cipher output is extracted from the state by taking the state bytes in the same order.

Hence if the one-dimensional index of a byte within a block is n and the two dimensional index is (i ,j), we have:

$$i = n \bmod 4\,; \qquad j = \lfloor n \,/\, 4 \rfloor; \qquad n = i + 4 * j$$

Moreover, the index i is also the byte number within a 4-byte vector or word and j is the index for the vector or word within the enclosing block.

The number of rounds is denoted by **Nr** and depends on the values **Nb** and **Nk**. It is given in Table 1.

Nr	**Nb** = 4	**Nb** = 6	**Nb** = 8
Nk = 4	10	12	14
Nk = 6	12	12	14
Nk = 8	14	14	14

Table 1: Number of rounds (Nr) as a function of the block and key length.

4.2 The round transformation

The round transformation is composed of four different transformations. In pseudo C notation we have:

```
Round(State,RoundKey)
{
ByteSub(State);
ShiftRow(State);
MixColumn(State);
AddRoundKey(State,RoundKey);
}
```

The final round of the cipher is slightly different. It is defined by:

```
FinalRound(State,RoundKey)
{
ByteSub(State) ;
ShiftRow(State) ;
AddRoundKey(State,RoundKey);
}
```

In this notation, the “functions” (`Round`, `ByteSub`, `ShiftRow`, …) operate on arrays to which pointers (`State`, `RoundKey`) are provided.

It can be seen that the final round is equal to the round with the MixColumn step removed.

The component transformations are specified in the following subsections.

4.2.1 The ByteSub transformation

The ByteSub Transformation is a non-linear byte substitution, operating on each of the State bytes independently. The substitution table (or S-box) is invertible and is constructed by the composition of two transformations:

1. First, taking the multiplicative inverse in GF(2^8), with the representation defined in Section 2.1. '`00`' is mapped onto itself.
2. Then, applying an affine (over GF(2)) transformation defined by:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \\ x_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

The application of the described S-box to all bytes of the State is denoted by:

`ByteSub(State)` .

Figure 2 illustrates the effect of the ByteSub transformation on the State.

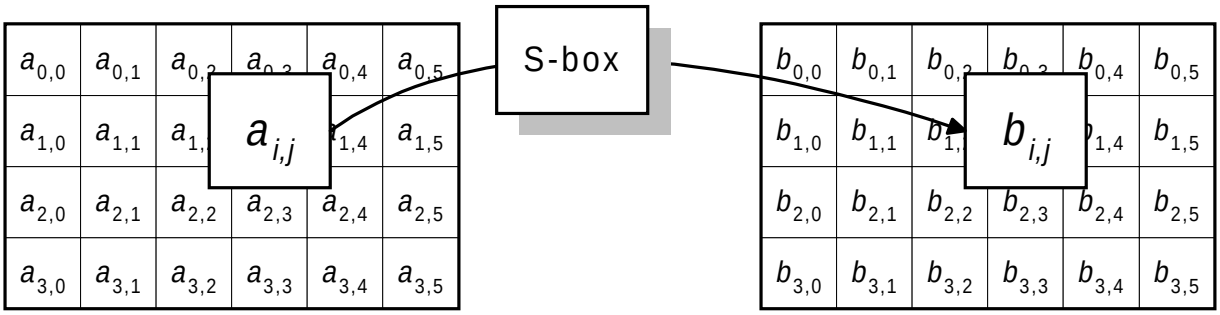

Figure 2: ByteSub acts on the individual bytes of the State.

The inverse of ByteSub is the byte substitution where the inverse table is applied. This is obtained by the inverse of the affine mapping followed by taking the multiplicative inverse in GF(2^8).

4.2.2 The ShiftRow transformation

In ShiftRow, the rows of the State are cyclically shifted over different offsets. Row 0 is not shifted, Row 1 is shifted over `C1` bytes, row 2 over `C2` bytes and row 3 over `C3` bytes.

The shift offsets `C1`, `C2` and `C3` depend on the block length **`Nb`**. The different values are specified in Table 2.

Nb	C1	C2	C3
4	1	2	3
6	1	2	3
8	1	3	4

Table 2: Shift offsets for different block lengths.

The operation of shifting the rows of the State over the specified offsets is denoted by:

`ShiftRow(State)` .

Figure 3 illustrates the effect of the ShiftRow transformation on the State.

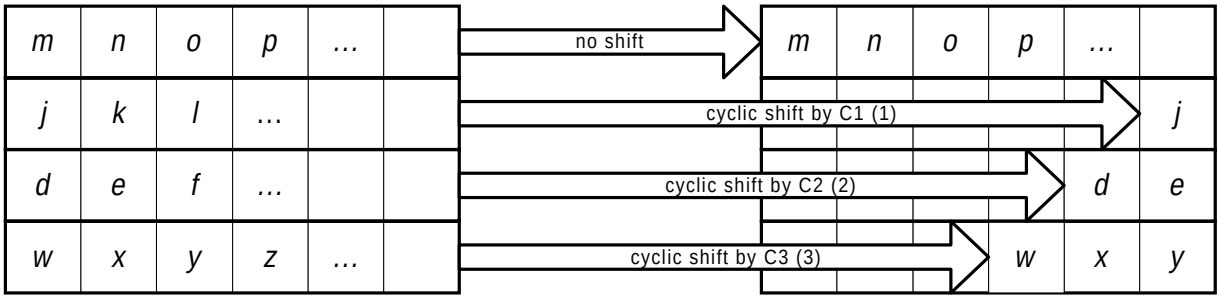

Figure 3: ShiftRow operates on the rows of the State.

The inverse of ShiftRow is a cyclic shift of the 3 bottom rows over **Nb**-C1, **Nb**-C2 and **Nb**-C3 bytes respectively so that the byte at position *j* in row *i* moves to position (*j* + **Nb**-C*i*) mod **Nb**.

4.2.3 The MixColumn transformation

In MixColumn, the columns of the State are considered as polynomials over GF(2^8) and multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x)$, given by

$$c(x) = \text{'03'}\ x^3 + \text{'01'}\ x^2 + \text{'01'}\ x + \text{'02'} .$$

This polynomial is coprime to $x^4 + 1$ and therefore invertible. As described in Section 2.2, this can be written as a matrix multiplication. Let b(x) = c(x) ⊗ a(x),

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

The application of this operation on all columns of the State is denoted by

`MixColumn(State)`.

Figure 4 illustrates the effect of the MixColumn transformation on the State.

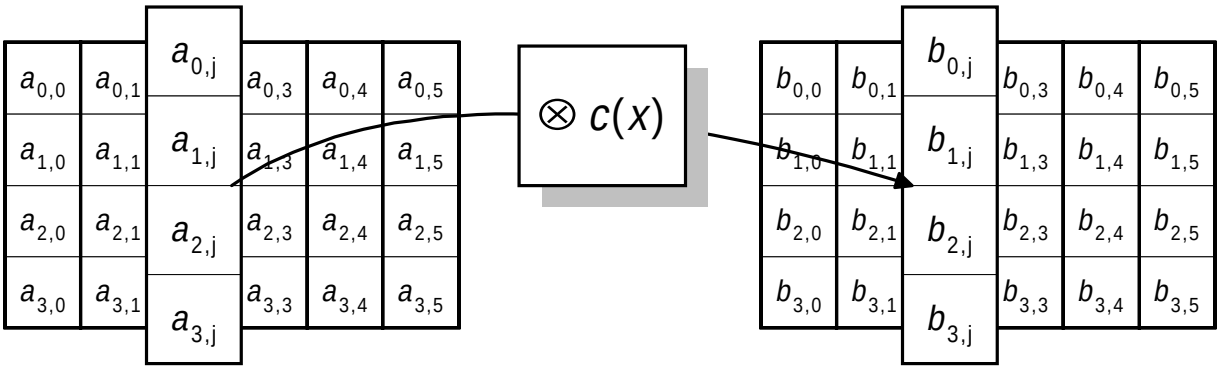

Figure 4: MixColumn operates on the columns of the State.

The inverse of MixColumn is similar to MixColumn. Every column is transformed by multiplying it with a specific multiplication polynomial $d(x)$, defined by

$$(\text{'03'}\ x^3 + \text{'01'}\ x^2 + \text{'01'}\ x + \text{'02'}) \otimes d(x) = \text{'01'}.$$

It is given by:

$$d(x) = \text{'0B'}\ x^3 + \text{'0D'}\ x^2 + \text{'09'}\ x + \text{'0E'}.$$

4.2.4 The Round Key addition

In this operation, a Round Key is applied to the State by a simple bitwise EXOR. The Round Key is derived from the Cipher Key by means of the key schedule. The Round Key length is equal to the block length **Nb**.

The transformation that consists of EXORing a Round Key to the State is denoted by:

`AddRoundKey(State,RoundKey)`.

This transformation is illustrated in Figure 5.

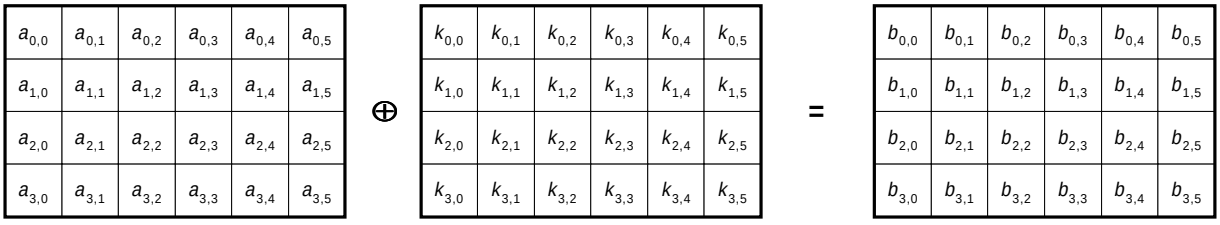

Figure 5: In the key addition the Round Key is bitwise EXORed to the State.

AddRoundKey is its own inverse.

4.3 Key schedule

The Round Keys are derived from the Cipher Key by means of the key schedule. This consists of two components: the Key Expansion and the Round Key Selection. The basic principle is the following:

- The total number of Round Key bits is equal to the block length multiplied by the number of rounds plus 1. (e.g., for a block length of 128 bits and 10 rounds, 1408 Round Key bits are needed).
- The Cipher Key is expanded into an *Expanded Key*.
- Round Keys are taken from this Expanded Key in the following way: the first Round Key consists of the first **Nb** words, the second one of the following **Nb** words, and so on.

4.3.1 Key expansion

The Expanded Key is a linear array of 4-byte words and is denoted by `W[Nb*(Nr+1)]`. The first **Nk** words contain the Cipher Key. All other words are defined recursively in terms of words with smaller indices. The key expansion function depends on the value of **Nk**: there is a version for **Nk** equal to or below 6, and a version for **Nk** above 6.

For **Nk** ≤ 6, we have:

```
KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])
{
     for(i = 0; i < Nk; i++)
          W[i] = (Key[4*i],Key[4*i+1],Key[4*i+2],Key[4*i+3]);

     for(i = Nk; i < Nb * (Nr + 1); i++)
     {
          temp = W[i - 1];
          if (i % Nk == 0)
               temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
          W[i] = W[i - Nk] ^ temp;
     }
}
```

In this description, `SubByte(W)` is a function that returns a 4-byte word in which each byte is the result of applying the Rijndael S-box to the byte at the corresponding position in the input word. The function `RotByte(W)` returns a word in which the bytes are a cyclic permutation of those in its input such that the input word (a,b,c,d) produces the output word (b,c,d,a).

It can be seen that the first **Nk** words are filled with the Cipher Key. Every following word `W[i]` is equal to the EXOR of the previous word `W[i-1]` and the word **Nk** positions earlier `W[i-Nk]`. For words in positions that are a multiple of **Nk**, a transformation is applied to `W[i-1]` prior to the EXOR and a round constant is EXORed. This transformation consists of a cyclic shift of the bytes in a word (`RotByte`), followed by the application of a table lookup to all four bytes of the word (`SubByte`).

For **Nk** > 6, we have:

```
KeyExpansion(byte Key[4*Nk] word W[Nb*(Nr+1)])
{
     for(i = 0; i < Nk; i++)
          W[i] = (key[4*i],key[4*i+1],key[4*i+2],key[4*i+3]);

     for(i = Nk; i < Nb * (Nr + 1); i++)
     {
          temp = W[i - 1];
          if (i % Nk == 0)
               temp = SubByte(RotByte(temp)) ^ Rcon[i / Nk];
          else if (i % Nk == 4)
               temp = SubByte(temp);
          W[i] = W[i - Nk] ^ temp;
     }
}
```

The difference with the scheme for **Nk** ≤ 6 is that for `i-4` a multiple of **Nk**, `SubByte` is applied to `W[i-1]` prior to the EXOR.

The round constants are independent of **Nk** and defined by:

`Rcon[i] = (RC[i],'00','00','00')`

with RC[I] representing an element in GF(2^8) with a value of $x^{(i-1)}$ so that:

`RC[1] =` **`1`** `(i.e. '01')`
`RC[i] =` **`x`** `(i.e. '02')` $\bullet$`(RC[i-1]) =` $x^{(i-1)}$

4.3.2 Round Key selection

Round key `i` is given by the Round Key buffer words `W[Nb*i]` to `W[Nb*(i+1)]`. This is illustrated in Figure 6.

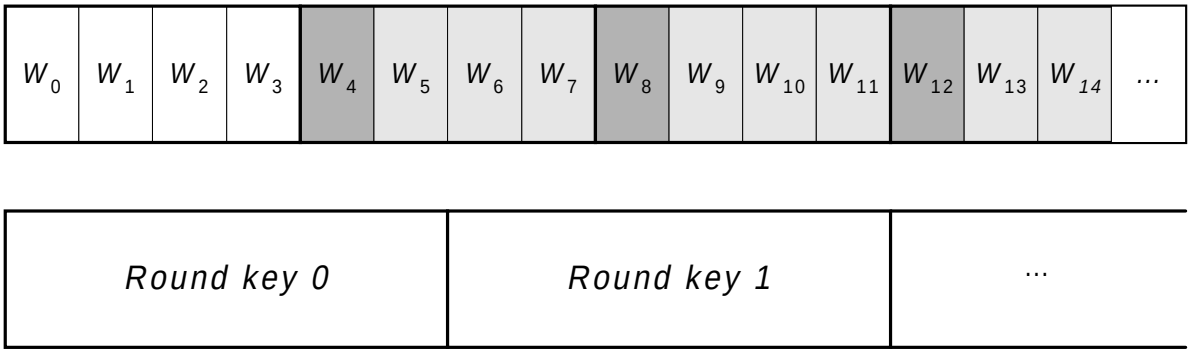

Figure 6: Key expansion and Round Key selection for Nb = 6 and Nk = 4.

Note: The key schedule can be implemented without explicit use of the array `W[Nb*(Nr+1)]`. For implementations where RAM is scarce, the Round Keys can be computed on-the-fly using a buffer of **Nk** words with almost no computational overhead.

4.4 The cipher

The cipher Rijndael consists of

- an initial Round Key addition;
- **Nr**-1 Rounds;
- a final round.

In pseudo C code, this gives:

```
Rijndael(State,CipherKey)
{
KeyExpansion(CipherKey,ExpandedKey) ;
AddRoundKey(State,ExpandedKey);
For( i=1 ; i<Nr ; i++ ) Round(State,ExpandedKey + Nb*i) ;
FinalRound(State,ExpandedKey + Nb*Nr);
}
```

The key expansion can be done on beforehand and Rijndael can be specified in terms of the Expanded Key.

```
Rijndael(State,ExpandedKey)
{
AddRoundKey(State,ExpandedKey);
For( i=1 ; i<Nr ; i++ ) Round(State,ExpandedKey + Nb*i) ;
FinalRound(State,ExpandedKey + Nb*Nr);
}
```

Note: the Expanded Key shall **always** be derived from the Cipher Key and never be specified directly. There are however no restrictions on the selection of the Cipher Key itself.

5. Implementation aspects

The Rijndael cipher is suited to be implemented efficiently on a wide range of processors and in dedicated hardware. We will concentrate on 8-bit processors, typical for current Smart Cards and on 32-bit processors, typical for PCs.

5.1 8-bit processor

On an 8-bit processor, Rijndael can be programmed by simply implementing the different component transformations. This is straightforward for RowShift and for the Round Key addition. The implementation of ByteSub requires a table of 256 bytes.

The Round Key addition, ByteSub and RowShift can be efficiently combined and executed serially per State byte. Indexing overhead is minimised by explicitly coding the operation for every State byte.

The transformation MixColumn requires matrix multiplication in the field GF(2^8). This can be implemented in an efficient way. We illustrate it for one column:

```
Tmp = a[0] ^ a[1] ^ a[2] ^ a[3] ; /* a is a byte array */
Tm = a[0] ^ a[1] ; Tm = xtime(Tm); a[0] ^= Tm ^ Tmp ;
Tm = a[1] ^ a[2] ; Tm = xtime(Tm); a[1] ^= Tm ^ Tmp ;
Tm = a[2] ^ a[3] ; Tm = xtime(Tm); a[2] ^= Tm ^ Tmp ;
Tm = a[3] ^ a[0] ; Tm = xtime(Tm); a[3] ^= Tm ^ Tmp ;
```

This description is for clarity. In practice, coding is of course done in assembly language. To prevent timing attacks, attention must be paid that `xtime` is implemented to take a fixed number of cycles, independent of the value of its argument. In practice this can be achieved by using a dedicated table-lookup.

Obviously, implementing the key expansion in a single shot operation is likely to occupy too much RAM in a Smart Card. Moreover, in most applications, such as debit cards or electronic purses, the amount of data to be enciphered, deciphered or that is subject to a MAC is typically only a few blocks per session. Hence, not much performance can be gained by expanding the key only once for multiple applications of the block cipher.

The key expansion can be implemented in a cyclic buffer of `4*max(`**`Nb,Nk`**`)` bytes. The Round Key is updated in between Rounds. All operations in this key update can be implemented efficiently on byte level. If the Cipher Key length and the blocks length are equal or differ by a factor 2, the implementation is straightforward. If this is not the case, an additional buffer pointer is required.

5.2 32-bit processor

5.2.1 The Round Transformation

The different steps of the round transformation can be combined in a single set of table lookups, allowing for very fast implementations on processors with word length 32 or above. In this section, it is explained how this can be done.

We express one column of the round output *e* in terms of bytes of the round input *a.* In this section, $a_{i,j}$ denotes the byte of *a* in row *i* and column *j*, a_j denotes the column *j* of State *a.* For the key addition and the MixColumn transformation, we have

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix} \text{ and } \begin{bmatrix} d_{0,j} \\ d_{1,j} \\ d_{2,j} \\ d_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix}.$$

For the ShiftRow and the ByteSub transformations, we have:

$$\begin{bmatrix} c_{0,j} \\ c_{1,j} \\ c_{2,j} \\ c_{3,j} \end{bmatrix} = \begin{bmatrix} b_{0,j} \\ b_{1,j-C1} \\ b_{2,j-C2} \\ b_{3,j-C3} \end{bmatrix} \text{ and } b_{i,j} = \mathrm{S}\left[a_{i,j}\right].$$

In this expression the column indices must be taken modulo **Nb**. By substitution, the above expressions can be combined into:

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} \mathrm{S}\left[a_{0,j}\right] \\ \mathrm{S}\left[a_{1,j-C1}\right] \\ \mathrm{S}\left[a_{2,j-C2}\right] \\ \mathrm{S}\left[a_{3,j-C3}\right] \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}.$$

The matrix multiplication can be expressed as a linear a combination of vectors:

$$\begin{bmatrix} e_{0,j} \\ e_{1,j} \\ e_{2,j} \\ e_{3,j} \end{bmatrix} = \mathrm{S}\left[a_{0,j}\right]\begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus \mathrm{S}\left[a_{1,j-C1}\right]\begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus \mathrm{S}\left[a_{2,j-C2}\right]\begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus \mathrm{S}\left[a_{3,j-C3}\right]\begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \oplus \begin{bmatrix} k_{0,j} \\ k_{1,j} \\ k_{2,j} \\ k_{3,j} \end{bmatrix}.$$

The multiplication factors S[$a_{i,j}$] of the four vectors are obtained by performing a table lookup on input bytes $a_{i,j}$ in the S-box table S[256].

We define tables T_0 to T_3 :

$$\mathrm{T}_0[a] = \begin{bmatrix} \mathrm{S}[a] \bullet 02 \\ \mathrm{S}[a] \\ \mathrm{S}[a] \\ \mathrm{S}[a] \bullet 03 \end{bmatrix} \quad \mathrm{T}_1[a] = \begin{bmatrix} \mathrm{S}[a] \bullet 03 \\ \mathrm{S}[a] \bullet 02 \\ \mathrm{S}[a] \\ \mathrm{S}[a] \end{bmatrix} \quad T_2[a] = \begin{bmatrix} \mathrm{S}[a] \\ \mathrm{S}[a] \bullet 03 \\ \mathrm{S}[a] \bullet 02 \\ \mathrm{S}[a] \end{bmatrix} \quad \mathrm{T}_3[a] = \begin{bmatrix} \mathrm{S}[a] \\ \mathrm{S}[a] \\ \mathrm{S}[a] \bullet 03 \\ \mathrm{S}[a] \bullet 02 \end{bmatrix}.$$

These are 4 tables with 256 4-byte word entries and make up for 4KByte of total space. Using these tables, the round transformation can be expressed as:

$$e_j = \mathrm{T}_0\left[a_{0,j}\right] \oplus \mathrm{T}_1\left[a_{1,j-C1}\right] \oplus \mathrm{T}_2\left[a_{2,j-C2}\right] \oplus \mathrm{T}_3\left[a_{3,j-C3}\right] \oplus k_j .$$

Hence, a table-lookup implementation with 4 Kbytes of tables takes only 4 table lookups and 4 EXORs per column per round.

It can be seen that $\mathrm{T}_i[a]$ = RotByte($\mathrm{T}_{i-1}[a]$). At the cost of 3 additional rotations per round per column, the table-lookup implementation can be realised with only one table, i.e., with a total table size of 1KByte. We have

$$e_j = k_j \oplus \mathrm{T}_0\left[b_{0,j}\right] \oplus \mathrm{Rotbyte}(\mathrm{T}_0\left[b_{1,j-C1}\right] \oplus \mathrm{Rotbyte}(\mathrm{T}_0\left[b_{2,j-C2}\right] \oplus \mathrm{Rotbyte}(\mathrm{T}_0\left[b_{3,j-C3}\right])))$$

The code-size (relevant in applets) can be kept small by including code to generate the tables instead of the tables themselves.

In the final round, there is no MixColumn operation. This boils down to the fact that the S table must be used instead of the T tables. The need for additional tables can be suppressed by extracting the S table from the T tables by masking while executing the final round.

Most operations in the key expansion can be implemented by 32-bit word EXORs. The additional transformations are the application of the S-box and a cyclic shift over 8-bits. This can be implemented very efficiently.

5.2.2 Parallelism

It can be seen that there is considerable parallelism in the round transformation. All four component transformations of the round act in a parallel way on bytes, rows or columns of the State.

In the table-lookup implementation, all table lookups can in principle be done in parallel. The EXORs can be done in parallel for the most part also.

The key expansion is clearly of a more sequential nature: the value of `W[i-1]` is needed for the computation of `W[i]`. However, in most applications where speed is critical, the KeyExpansion has to be done only once for a large number of cipher executions. In applications where the Cipher Key changes often (in extremis once per application of the Block Cipher), the key expansion and the cipher Rounds can be done in parallel..

5.2.3 Hardware suitability

The cipher is suited to be implemented in dedicated hardware. There are several trade-offs between area and speed possible. Because the implementation in software on general-purpose processors is already very fast, the need for hardware implementations will very probably be limited to two specific cases:

- Extremely high speed chip with no area restrictions: the *T* tables can be hardwired and the EXORs can be conducted in parallel.
- Compact co-processor on a Smart Card to speed up Rijndael execution: for this platform typically the S-box and the `xtime` (or the complete MixColumn) operation can be hardwired.

5.3 The inverse cipher

In the table-lookup implementation it is essential that the only non-linear step (ByteSub) is the first transformation in a round and that the rows are shifted before MixColumn is applied. In the Inverse of a round, the order of the transformations in the round is reversed, and consequently the non-linear step will end up being the last step of the inverse round and the rows are shifted after the application of (the inverse of) MixColumn. The inverse of a round can therefore not be implemented with the table lookups described above.

This implementation aspect has been anticipated in the design. The structure of Rijndael is such that the sequence of transformations of its inverse is equal to that of the cipher itself, with the transformations replaced by their inverses and a change in the key schedule. This is shown in the following subsections.

Note: this identity in *structure* differs from the identity of *components and structure* in IDEA [LaMaMu91].

5.3.1 Inverse of a two-round Rijndael variant

The inverse of a round is given by:

```
InvRound(State,RoundKey)
{
AddRoundKey(State,RoundKey);
InvMixColumn(State);
InvShiftRow(State);
InvByteSub(State);
}
```

The inverse of the final round is given by:

```
InvFinalRound(State,RoundKey)
{
AddRoundKey(State,RoundKey);
InvShiftRow(State);
InvByteSub(State);
}
```

The inverse of a two-round variant of Rijndael consists of the inverse of the final round followed by the inverse of a round, followed by a Round Key Addition. We have:

```
AddRoundKey(State,ExpandedKey+2*Nb);
InvShiftRow(State);
InvByteSub(State);
AddRoundKey(State,ExpandedKey+Nb);
InvMixColumn(State);
InvShiftRow(State);
InvByteSub(State);
AddRoundKey(State,ExpandedKey);
```

5.3.2 Algebraic properties

In deriving the equivalent structure of the inverse cipher, we make use of two properties of the component transformations.

First, the order of ShiftRow and ByteSub is indifferent. ShiftRow simply transposes the bytes and has no effect on the byte values. ByteSub works on individual bytes, independent of their position.

Second, the sequence

```
AddRoundKey(State,RoundKey);
InvMixColumn(State);
```

can be replaced by:

```
InvMixColumn(State);
AddRoundKey(State,InvRoundKey);
```

with InvRoundKey obtained by applying InvMixColumn to the corresponding RoundKey. This is based on the fact that for a linear transformation A, we have $A(x+k) = A(x)+A(k)$.

5.3.3 The equivalent inverse cipher structure

Using the properties described above, the inverse of the two-round Rijndael variant can be transformed into:

```
AddRoundKey(State,ExpandedKey+2*Nb);

InvByteSub(State);
InvShiftRow(State);
InvMixColumn(State);
AddRoundKey(State,I_ExpandedKey+Nb);

InvByteSub(State);
InvShiftRow(State);
AddRoundKey(State,ExpandedKey);
```

It can be seen that we have again an initial Round Key addition, a round and a final round. The Round and the final round have the same structure as those of the cipher itself. This can be generalised to any number of rounds.

We define a round and the final round of the inverse cipher as follows:

```
I_Round(State,I_RoundKey)
{
InvByteSub(State);
InvShiftRow(State);
InvMixColumn(State);
AddRoundKey(State,I_RoundKey);
}

I_FinalRound(State,I_RoundKey)
{
InvByteSub(State);
InvShiftRow(State);
AddRoundKey(State,RoundKey0);
}
```

The Inverse of the Rijndael Cipher can now be expressed as follows:

```
I_Rijndael(State,CipherKey)
{
I_KeyExpansion(CipherKey,I_ExpandedKey) ;
AddRoundKey(State,I_ExpandedKey+ Nb*Nr);
For( i=Nr-1 ; i>0 ; i-- ) Round(State,I_ExpandedKey+ Nb*i) ;
FinalRound(State,I_ExpandedKey);
}
```

The key expansion for the Inverse Cipher is defined as follows:

1. Apply the Key Expansion.
2. Apply InvMixColumn to all Round Keys except the first and the last one.

In Pseudo C code, this gives:

```
I_KeyExpansion(CipherKey,I_ExpandedKey)
{
KeyExpansion(CipherKey,I_ExpandedKey);
for( i=1 ; i < Nr ; i++ )
     InvMixColumn(I_ExpandedKey + Nb*i) ;
}
```

5.3.4 Implementations of the inverse cipher

The choice of the MixColumn polynomial and the key expansion was partly based on cipher performance arguments. Since the inverse cipher is similar in structure, but uses a MixColumn transformation with another polynomial and (in some cases) a modified key schedule, a performance degradation is observed on 8-bit processors.

This asymmetry is due to the fact that the performance of the inverse cipher is considered to be less important than that of the cipher. In many applications of a block cipher, the inverse cipher operation is not used. This is the case for the calculation of MACs, but also when the cipher is used in CFB-mode or OFB-mode.

5.3.4.1 8-bit processors

As explained in Section 4.1, the operation MixColumn can be implemented quite efficiently on 8-bit processors. This is because the coefficients of MixColumn are limited to '01', '02' and '03' and because of the particular arrangement in the polynomial. Multiplication with these coefficients can be done very efficiently by means of the procedure `xtime()`. The coefficients of InvMixColumn are '09', '0E', '0B' and '0D'. In our 8-bit implementation, these multiplications take significantly more time. A considerable speed-up can be obtained by using table lookups at the cost of additional tables.

The key expansion operation that generates `W` is defined in such a way that we can also start with the last **`Nk`** words of Round Key information and roll back to the original Cipher Key. So, calculation 'on-the-fly' of the Round Keys, starting from an "Inverse Cipher Key", is still possible.

5.3.4.2 32-bit processors

The Round of the inverse cipher can be implemented with table lookups in exactly the same way as the round of the cipher and there is no performance degradation with respect to the cipher. The look-up tables for the inverse are of course different.

The key expansion for the inverse cipher is slower, because after the key expansion all but two of the Round Keys are subject to InvMixColumn (cf. Section 5.3.3).

5.3.4.3 Hardware suitability

Because the cipher and its inverse use different transformations, a circuit that implements Rijndael does not automatically support the computation of the inverse of Rijndael. Still, in a circuit implementing both Rijndael and its inverse, parts of the circuit can be used for both functions.

This is for instance the case for the non-linear layer. The S-box is constructed from two mappings:

$$S(x) = f(g(x)),$$

where $g(x)$ is the mapping:

$$x \Rightarrow x^{-1} \text{ in } GF(2^8)$$

and $f(x)$ is the affine mapping.

The mapping $g(x)$ is self-inverse and hence $S^{-1}(x) = g^{-1}(f^{-1}(x)) = g(f^{-1}(x))$. Therefore when we want both S and S^{-1}, we need to implement only g, f and f^{-1}. Since both f and f^{-1} are very simple bit-level functions, the extra hardware can be reduced significantly compared to having two full S-boxes.

Similar arguments apply to the re-use of the `xtime` transformation in the diffusion layer.

6. Performance figures

6.1 8-bit processors

Rijndael has been implemented in assembly language for two types of microprocessors that are representative for Smart Cards in use today.

In these implementations the Round Keys are computed in between the rounds of the cipher (just-in-time calculation of the Round Keys) and therefore the key schedule is repeated for every cipher execution. This means that there is no extra time required for key set-up or a key change. There is also no time required for algorithm set-up. We have only implemented the forward operation of the cipher. Implementation efforts by other people have indicated that the inverse cipher turns out to be about 30 % slower. This is due to reasons explained in the section on implementation.

6.1.1 Intel 8051

Rijndael has been implemented on the Intel 8051 microprocessor, using 8051 Development tools of Keil Elektronik: uVision IDE for Windows and dScope Debugger/Simulator for Windows.

Execution time for several code sizes is given in Table 3 (1 cycle = 12 oscillator periods).

Key/Block Length	Number of Cycles	Code length
(128,128) a)	4065 cycles	768 bytes
(128,128) b)	3744 cycles	826 bytes
(128,128) c)	3168 cycles	1016 bytes
(192,128)	4512 cycles	1125 bytes
(256,128)	5221 cycles	1041 bytes

Table 3: Execution time and code size for Rijndael in Intel 8051 assembler.

6.1.2 Motorola 68HC08

Rijndael has been implemented on the Motorola 68HC08 microprocessor using the 68HC08 development tools by P&E Microcomputer Systems, Woburn, MA USA, the IASM08 68HC08 Integrated Assembler and SIML8 68HC08 simulator. Execution time, code size and required RAM for a number of implementations are given in Table 4 (1 cycle = 1oscillator period). No optimisation of code length has been attempted for this processor.

Key/Block Length	Number of Cycles	Required RAM	Code length
(128,128) a)	8390 cycles	36 bytes	919 bytes
(192,128)	10780 cycles	44 bytes	1170 bytes
(256,128)	12490 cycles	52 bytes	1135 bytes

Table 4: Execution time and code size for Rijndael in Motorola 68HC08 Assembler.

6.2 32-bit processors

6.2.1 Optimised ANSI C

We have no access to a Pentium Pro computer. Speed estimates for this platform were originally generated by compiling the code with EGCS (release 1.0.2) and executing it on a 200 MHz Pentium, running Linux. However, since this report was first published further performance figures have become available and those published by Brian Gladman are reported below.

The AES CD figures are for ANSI C using the NIST API. The figures reported by Brian Gladman are for the Pentium Pro and Pentium II processor families using a more efficient interface. These results were obtained with the Microsoft Visual C++ (version 6) compiler that provides fast intrinsic rotate instructions. The ability to use these instructions within C code provides substantial performance gains without incurring significant portability problems since many C compilers now offer equivalent facilities. The speed figures given in the tables have been scaled to be those that would apply on the 200MHz Pentium Pro reference platform.

Algorithm set-up takes no time. Key set-up and key change take exactly the same time: the time to generate the Expanded Key from the Cipher Key. The key set-up for the inverse cipher takes more time than the key set-up for the cipher itself (cf. Section 5.3.3).

Table 5 lists the number of cycles needed for the key expansion.

# cycles	**AES CD (ANSI C)**		**Brian Gladman (Visual C++)**	
(key,block) length	Rijndael	Rijndael^{-1}	Rijndael	Rijndael^{-1}
(128,128)	2100	2900	305	1389
(192,128)	2600	3600	277	1595
(256,128)	2800	3800	374	1960

Table 5: Number of cycles for the key expansion

The cipher and its inverse take the same time. The difference in performance that is discussed in the section on implementation, is only caused by the difference in the key set-up. Table 6 gives the figures for the raw encryption, when implemented in C, without counting the overhead caused by the AES API.

(key,block) length	AES CD (ANSI C)		Brian Gladman (Visual C++)	
	speed (Mbits/Sec)	# cycles/block	speed (Mbits/Sec)	# cycles/block
(128,128)	27.0	950	70.5	363
(192,128)	22.8	1125	59.3	432
(256,128)	19.8	1295	51.2	500

Table 6: Cipher (and inverse) performance

6.2.2 Java

We gratefully accepted the generous offer from Cryptix to produce the Java implementation. Cryptix provides however no performance figures. Our estimates are based on the execution time of the KAT and MCT code on a 200 MHz Pentium, running Linux. The JDK1.1.1 Java compiler was used. The performance figures of the Java implementation are given in Table 7.

We cannot provide estimates for the key set-up or algorithm set-up time.

Key/Block length	Speed	# cycles for Rijndael
(128,128)	1100 Kbit/s	23.0 Kcycles
(192,128)	930 Kbit/s	27.6 Kcycles
(256,128)	790 Kbit/s	32.3 Kcycles

Table 7: Performance figures for the cipher execution (Java)

7. Motivation for design choices

In the following subsections, we will motivate the choice of the specific transformations and constants. We believe that the cipher structure does not offer enough degrees of freedom to hide a trap door.

7.1 The reduction polynomial $m(x)$

The polynomial $m(x)$ ('11B') for the multiplication in GF(2^8) is the first one of the list of irreducible polynomials of degree 8, given in [LiNi86, p. 378].

7.2 The ByteSub S-box

The design criteria for the S-box are inspired by differential and linear cryptanalysis on the one hand and attacks using algebraic manipulations, such as interpolation attacks, on the other:

1. Invertibility;
2. Minimisation of the largest non-trivial correlation between linear combinations of input bits and linear combination of output bits;
3. Minimisation of the largest non-trivial value in the EXOR table;
4. Complexity of its algebraic expression in GF(2^8);
5. Simplicity of description.

In [Ny94] several methods are given to construct S-boxes that satisfy the first three criteria. For invertible S-boxes operating on bytes, the maximum input/output correlation can be made as low as 2^{-3} and the maximum value in the EXOR table can be as low as 4 (corresponding to a difference propagation probability of 2^{-6}).

We have decided to take from the candidate constructions in [Ny94] the S-box defined by the mapping $x \Rightarrow x^{-1}$ in GF(2^8).

By definition, the selected mapping has a very simple algebraic expression. This enables algebraic manipulations that can be used to mount attacks such as interpolation attacks [JaKn97]. Therefore, the mapping is modified by composing it with an additional invertible affine transformation. This affine transformation does not affect the properties with respect tot the first three criteria, but if properly chosen, allows the S-box to satisfy the fourth criterion.

We have chosen an affine mapping that has a very simple description per se, but a complicated algebraic expression if combined with the 'inverse' mapping. It can be seen as modular polynomial multiplication followed by an addition:

$$b(x) = (x^7 + x^6 + x^2 + x) + a(x)(x^7 + x^6 + x^5 + x^4 + 1) \bmod x^8 + 1$$

The modulus has been chosen as the simplest modulus possible. The multiplication polynomial has been chosen from the set of polynomials coprime to the modulus as the one with the simplest description. The constant has been chosen in such a way that that the S-box has no fixed points (S-box(a) = a) and no 'opposite fixed points' (S-box(a) = $\bar{a}$).

Note: other S-boxes can be found that satisfy the criteria above. In the case of suspicion of a trapdoor being built into the cipher, the current S-box might be replaced by another one. The cipher structure and number of rounds as defined even allow the use of an S-box that does not optimise the differential and linear cryptanalysis properties (criteria 2 and 3). Even an S-box that is "average" in this respect is likely to provide enough resistance against differential and linear cryptanalysis.

7.3 The MixColumn transformation

MixColumn has been chosen from the space of 4-byte to 4-byte linear transformations according to the following criteria:

1. Invertibility;
2. Linearity in GF(2);
3. Relevant diffusion power;
4. Speed on 8-bit processors;
5. Symmetry;
6. Simplicity of description.

Criteria 2, 5 and 6 have lead us to the choice to polynomial multiplication modulo x^4+1. Criteria 1, 3 and 4 impose conditions on the coefficients. Criterion 4 imposes that the coefficients have small values, in order of preference '00', '01', '02', '03'…The value '00' implies no processing at all, for '01' no multiplication needs to be executed, '02' can be implemented using `xtime` and '03' can be implemented using `xtime` and an additional EXOR.

The criterion 3 induces a more complicated conditions on the coefficients.

7.3.1 Branch number

In our design strategy, the following property of the linear transformation of MixColumn is essential. Let *F* be a linear transformation acting on byte vectors and let the *byte weight* of a vector be the number of nonzero bytes (not to be confused with the usual significance of Hamming weight, the number of nonzero bits). The byte weight of a vector is denoted by W(*a*). The *Branch Number* of a linear transformation is a measure of its diffusion power:

Definition: The *branch number* of a linear transformation *F* is

$$\min_{a \neq 0}(\mathrm{W}(a)+\mathrm{W}(F(a))).$$

A non-zero byte is called an *active* byte. For MixColumn it can be seen that if a state is applied with a single active byte, the output can have at most 4 active bytes, as MixColumn acts on the columns independently. Hence, the upper bound for the branch number is 5. The coefficients have been chosen in such a way that the upper bound is reached. If the branch number is 5, a difference in 1 input (or output) byte propagates to all 4 output (or input) bytes, a 2-byte input (or output) difference to at least 3 output (or input) bytes. Moreover, a linear relation between input and output bits involves bits from at least 5 different bytes from input and output.

7.4 The ShiftRow offsets

The choice from all possible combinations has been made based on the following criteria:

1. The four offsets are different and `C0` = 0;
2. Resistance against attacks using truncated differentials [Kn95];
3. Resistance against the Square attack [DaKnRi97];
4. Simplicity.

For certain combinations, attacks using truncated differentials can tackle more rounds (typically only one) than for other combinations. For certain combinations the Square attack can tackle more rounds than others. From the combinations that are best with respect to criteria 2 and 3, the simplest ones have been chosen.

7.5 The key expansion

The key expansion specifies the derivation of the Round Keys in terms of the Cipher Key. Its function is to provide resistance against the following types of attack:

- Attacks in which part of the Cipher Key is known to the cryptanalyst;
- Attacks where the Cipher Key is known or can be chosen, e.g., if the cipher is used as the compression function of a hash function[Kn95a];
- Related-key attacks [Bi93], [KeScWa96]. A necessary condition for resistance against related-key attacks is that there should not be two different Cipher Keys that have a large set of Round Keys in common.

The key expansion also plays an important role in the elimination of symmetry:

- Symmetry in the round transformation: the round transformation treats all bytes of a state in very much the same way. This symmetry can be removed by having round constants in the key schedule;
- Symmetry between the rounds: the round transformation is the same for all rounds. This equality can be removed by having round-dependent round constants in the key schedule.

The key expansion has been chosen according to the following criteria:

- It shall use an invertible transformation, i.e., knowledge of any **Nk** consecutive words of the Expanded Key shall allow to regenerate the whole table;
- Speed on a wide range of processors;
- Usage of round constants to eliminate symmetries;
- Diffusion of Cipher Key differences into the Round Keys;
- Knowledge of a part of the Cipher Key or Round Key bits shall not allow to calculate many other Round Key bits.
- Enough non-linearity to prohibit the full determination of Round Key differences from Cipher Key differences only;
- Simplicity of description.

In order to be efficient on 8-bit processors, a light-weight, byte oriented expansion scheme has been adopted. The application of SubByte ensures the non-linearity of the scheme, without adding much space requirements on an 8-bit processor.

7.6 Number of rounds

We have determined the number of rounds by looking at the maximum number of rounds for which shortcut attacks have been found and added a considerable security margin. (A shortcut attack is an attack more efficient than exhaustive key search.)

For Rijndael with a block length and key length of 128 bits, no shortcut attacks have been found for reduced versions with more than 6 rounds. We added 4 rounds as a security margin. This is a conservative approach, because:

- Two rounds of Rijndael provide "full diffusion" in the following sense: every state bit depends on all state bits two rounds ago, or, a change in one state bit is likely to affect half of the state bits after two rounds. Adding 4 rounds can be seen as adding a "full diffusion" step at the beginning and at the end of the cipher. The high diffusion of a Rijndael round is thanks to its uniform structure that operates on all state bits. For so-called Feistel ciphers, a round only operates on half of the state bits and full diffusion can at best be obtained after 3 rounds and in practice it typically takes 4 rounds or more.
- Generally, linear cryptanalysis, differential cryptanalysis and truncated differential attacks exploit a propagation trail through n rounds in order to attack n+1 or n+2 rounds. This is also the case for the Square attack that uses a 4-round propagation structure to attack 6 rounds. In this respect, adding 4 rounds actually doubles the number of rounds through which a propagation trail has to be found.

For Rijndael versions with a longer Key, the number of rounds is raised by one for every additional 32 bits in the Cipher Key, for the following reasons:

- One of the main objectives is the absence of shortcut attacks, i.e., attacks that are more efficient than exhaustive key search. As with the key length the workload of exhaustive key search grows, shortcut attacks can afford to be less efficient for longer keys.
- Known-key (partially) and related-key attacks exploit the knowledge of cipher key bits or ability to apply different cipher keys. If the cipher key grows, the range of possibilities available to the cryptanalyst increases.

As no threatening known-key or related-key attacks have been found for Rijndael, even for 6 rounds, this is a conservative margin.

For Rijndael versions with a higher block length, the number of rounds is raised by one for every additional 32 bits in the block length, for the following reasons:

- For a block length above 128 bits, it takes 3 rounds to realise full diffusion, i.e., the diffusion power of a round, relative to the block length, diminishes with the block length.
- The larger block length causes the range of possible patterns that can be applied at the input/output of a sequence of rounds to increase. This added flexibility may allow to extend attacks by one or more rounds.

We have found that extensions of attacks by a single round are even hard to realise for the maximum block length of 256 bits. Therefore, this is a conservative margin.

8. Strength against known attacks

8.1 Symmetry properties and weak keys of the DES type

Despite the large amount of symmetry, care has been taken to eliminate symmetry in the behaviour of the cipher. This is obtained by the round constants that are different for each round. The fact that the cipher and its inverse use different components practically eliminates the possibility for weak and semi-weak keys, as existing for DES. The non-linearity of the key expansion practically eliminates the possibility of equivalent keys.

8.2 Differential and linear cryptanalysis

Differential cryptanalysis was first described by Eli Biham and Adi Shamir [BiSh91]. Linear cryptanalysis was first described by Mitsuru Matsui [Ma94].

Chapter 5 of [Da95] gives a detailed treatment of difference propagation and correlation. To better describe the anatomy of the basic mechanisms of linear cryptanalysis (LC) and of differential cryptanalysis (DC), new formalisms and terminology were introduced. With the aid of these it was, among other things, shown how input-output correlations over multiple rounds are composed. We will use the formalisms of [Da95] in the description of DC and LC. To provide the necessary background, Chapter 5 of [Da95] has been included in Annex.

8.2.1 Differential cryptanalysis

DC attacks are possible if there are predictable difference propagations over all but a few (typically 2 or 3) rounds that have a prop ratio (the relative amount of all input pairs that for the given input difference give rise to the output difference) significantly larger than 2^{1-n} if n is the block length. A difference propagation is *composed* of differential trails, where its prop ratio is the sum of the prop ratios of all differential trails that have the specified initial and final difference patterns. To be resistant against DC, it is therefore a necessary condition that there are no differential trails with a predicted prop ratio higher than 2^{1-n}.

For Rijndael, we prove that there are no 4-round differential trails with a predicted prop ratio above 2^{-150} (and no 8-round trails with a predicted prop ratio above 2^{-300}). For all block lengths of Rijndael, this is sufficient. For the significance of these predicted prop ratios, we refer to Chapter 5 of [Da95]. The proof is given in Section 8.2.3.

In [LaMaMu91] it has been proposed to perform differential cryptanalysis with another notion of difference. This is especially applicable to ciphers where the key addition is not a simple EXOR operation. Although in Rijndael the keys are applied using EXORs, it was investigated whether attacks could be mounted using another notion of difference. We have found no attack strategies better than using EXOR as the difference.

8.2.2 Linear cryptanalysis

LC attacks are possible if there are predictable input-output correlations over all but a few (typically 2 or 3) rounds significantly larger than $2^{n/2}$. An input-output correlation is *composed* of linear trails, where its correlation is the sum of the correlation coefficients of all linear trails that have the specified initial and final selection patterns. The correlation coefficients of the linear trails are signed and their sign depends on the value of the Round Keys. To be resistant against LC, it is a necessary condition that there are no linear trails with a correlation coefficient higher than $2^{n/2}$.

For Rijndael, we prove that there are no 4-round linear trails with a correlation above 2^{-75} (and no 8-round trails with a correlation above 2^{-150}). For all block lengths of Rijndael, this is sufficient. The proof is given in Section 8.2.4.

8.2.3 Weight of differential and linear trails

In [Da95], it is shown that:

- The *prop ratio* of a *differential trail* can be approximated by the product of the prop ratios of its active S-boxes.
- The *correlation* of a *linear trail* can be approximated by the product of input-output correlations of its active S-boxes.

The wide trail strategy can be summarised as follows:

- Choose an S-box where the maximum prop ratio and the maximum input-output correlation are as small as possible. For the Rijndael S-box this is respectively 2^{-6} and 2^{-3} .
- Construct the diffusion layer in such a way that there are no multiple-round trails with few active S-boxes.

We prove that the minimum number of active S-boxes in any 4-round differential or linear trail is 25. This gives a maximum prop ratio of 2^{-150} for any 4-round differential trail and a maximum of 2^{-75} for the correlation for any 4-round linear trail. This holds for all block lengths of Rijndael and is independent of the value of the Round Keys.

Note: the nonlinearity of an S-box chosen randomly from the set of possible invertible 8-bit S-boxes is expected to be less optimum. Typical values are 2^{-5} to 2^{-4} for the maximum prop ratio and 2^{-2} for the maximum input-output correlation.

8.2.4 Propagation of patterns

For DC, the active S-boxes in a round are determined by the nonzero bytes in the difference of the states at the input of a round. Let the pattern that specifies the positions of the active S-boxes be denoted by the term (*difference*) *activity pattern* and let the (*difference*) *byte weight* be the number of active bytes in a pattern.

For LC, the active S-boxes in a round are determined by the nonzero bytes in *the selection vectors* (see Annex) at the input of a round. Let the pattern that specifies the positions of the active S-boxes be denoted by the term (*correlation*) *activity pattern* and let the (*correlation*) *byte weight* $W(a)$ be the number of active bytes in a pattern a.

Moreover, let a column of an activity pattern with at least one active byte be denoted by *active column*. Let the *column weight*, denoted by $W_C(a)$, be the number of active columns in a pattern. The byte weight of a column j of a, denoted by $W(a)|_j$, is the number of active bytes in it.

The weight of a trail is the sum of the weights of its activity patterns at the input of each round.

Difference and correlation activity patterns can be seen as propagating through the transformations of the different rounds of the block cipher to form linear and differential trails. This is illustrated with an example in Figure 7.

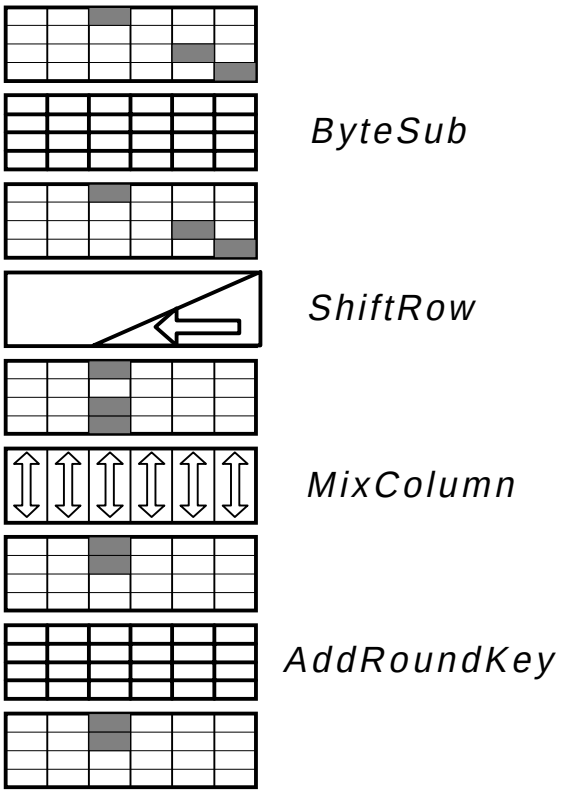

Figure 7: Propagation of activity pattern (in grey) through a single round

The different transformations of Rijndael have the following effect on these patterns and weights:

- ByteSub and AddRoundKey: activity patterns, byte and column weight are invariant.
- ShiftRow: byte weight is invariant as there is no inter-byte interaction.
- MixColumn: column weight is invariant as there is no inter-column interaction.

ByteSub and AddRoundKey do not play a role in the propagation of activity patterns and therefore in this discussion the effect of a round is reduced to that of ShiftRow followed by MixColumn. In the following, ByteSub and AddRoundKey will be ignored. MixColumn has a branch number equal to 5, implying:

- For any active column of a pattern at its input (or, equivalently, at its output), the sum of the byte weights at input *and* output for this column is lower bounded by 5.

ShiftRow has the following properties:

- The column weight of a pattern at its output is lower bounded by the maximum of the byte weights of the columns of the pattern at its input.
- The column weight of a pattern at its input is lower bounded by the maximum of the byte weights of the columns of the pattern at its output.

This is thanks to the property that MixColumn permutes the bytes of a column to all different columns.

In our description, the activity pattern at the input of a round i is denoted by a_{i-1} and the activity pattern after applying ShiftRow of round i is denoted by b_{i-1}. The initial round is numbered 1 and the initial difference pattern is denoted by a_0. Clearly, a_i and b_i are separated by ShiftRow and have the same byte weight, b_{j-1} and a_j are separated by MixColumn and have the same column weight. The weight of an m-round trail is given by the sum of the weights of a_0 to a_{m-1}. The propagation properties are illustrated in Figure 8. In this figure, active bytes are indicated in dark grey, active columns in light grey.

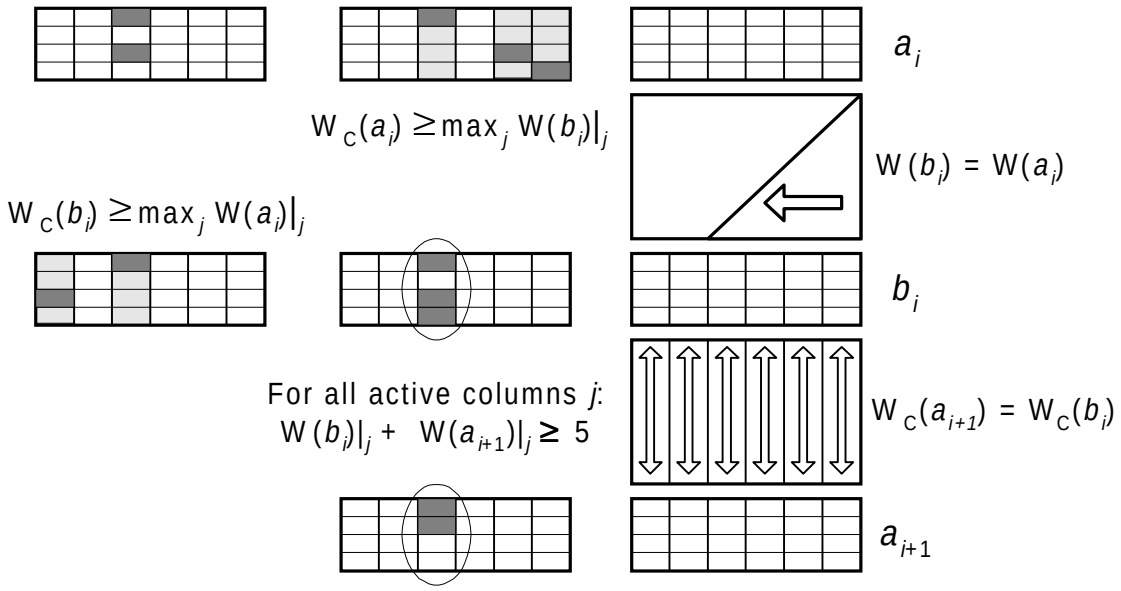

Figure 8: Propagation of patterns in a single round.

Theorem 1: The weight of a two-round trail with Q active columns at the input of the second round is lower bounded by $5Q$.

Proof: The fact that MixColumn has a Branch Number equal to 5 implies that sum of the byte weights of each column in b_0 and a_1 is lower bounded by 5. If the column weight of a_1 is Q, this gives a lower bounded of $5Q$ for the sum of the byte weights of b_0 and a_1. As a_0 and b_0 have the same byte weight, the lower bounded is also valid for the sum of the weights a_0 and a_1, proving the theorem.

QED

Theorem 1 is illustrated in Figure 9.

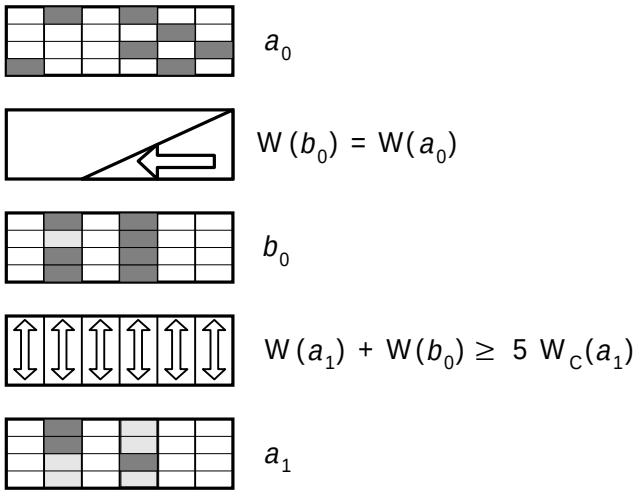

Figure 9: Illustration of Theorem 1 with $Q = 2$.

From this it follows that any two-round trail has at least 5 active S-boxes.

Lemma 1: in a two-round trail, the sum of the number of active columns at its input and the number of active columns at its output is at least 5. In other words, the sum of the columns weights of a_0 and a_2 is at least 5.

Proof: ShiftRow moves all bytes in a column of a_i to different columns in b_i and vice versa. It follows that the column weight of a_i is lower bounded the byte weights of the individual columns of b_i. Likewise the column weight of b_i is lower bounded by the byte weights of the individual columns of a_i.

In a trail, at least one column of a_1 (or equivalently b_0) is active. Let this column be denoted by "column g". Because MixColumn has a branch number of 5, the sum of the byte weights of column g in b_0 and column g in a_1 is lower bounded by 5. The column weight of a_0 is lower bounded by the byte weight of column g of b_0. The column weight of b_1 is lower bounded by the byte weight of column g of a_1. It follows that the sum of the column weights of a_0 and b_1 is lower bounded by 5. As the column weight of a_2 is equal to that of b_1, the lemma is proven.

QED

Lemma 1 is illustrated in Figure 10.

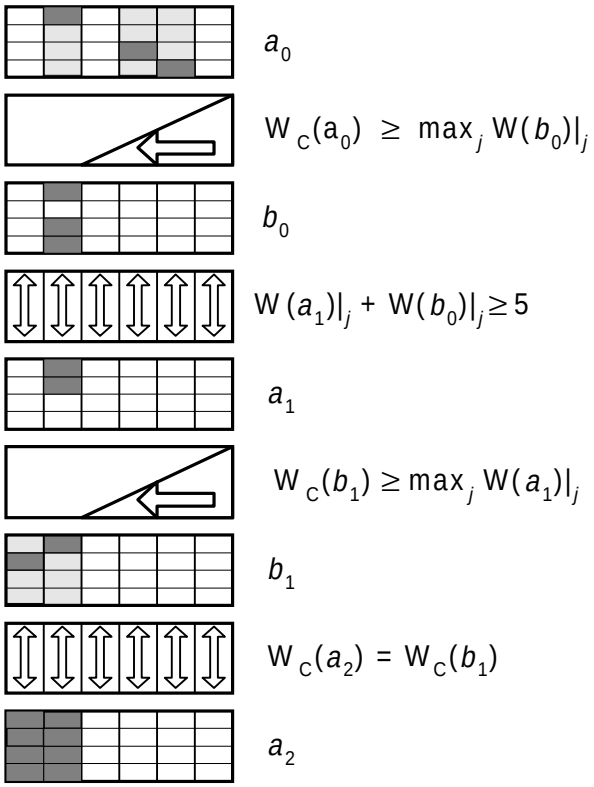

Figure 10: Illustration of Lemma 1 with one active column in a_1.

Theorem 2: Any trail over four rounds has at least 25 active bytes.

Proof: By applying Theorem 1 on the first two rounds (1 and 2) and on the last two rounds (3 and 4), it follows that the byte weight of the trail is lower bounded by the sum of the column weight of a_1 and a_3 multiplied by 5. By applying Lemma 1, the sum of the column weight of a_1 and a_3 is lower bounded by 5. From this it follows that the byte weight of the four-round trail is lower bounded by 25.

QED

Theorem 2 is illustrated in Figure 11.

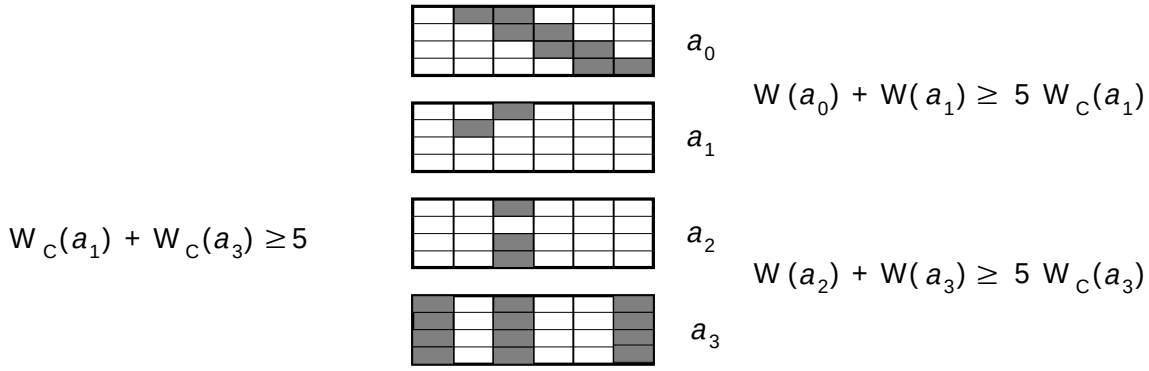

Figure 11: Illustration of Theorem 2.

8.3 Truncated differentials

The concept of truncated differentials was first published by Lars Knudsen [Kn95]. The corresponding class of attacks exploit the fact that in some ciphers differential trails tend to *cluster* [Da95] (see Annex). Clustering takes place if for certain sets of input difference patterns and output difference patterns, the number of differential trails is exceedingly large. The expected probability that a differential trail stays within the boundaries of the cluster can be computed independently of the prop ratios of the individual differential trails. Ciphers in which all transformation operate on the state in well aligned blocks are prone to be susceptible to this type of attack. Since this is the case for Rijndael, all transformations operating on bytes rather than individual bits, we investigated its resistance against “truncated differentials”. For 6 rounds or more, no attacks faster than exhaustive key search have been found.

8.4 The Square attack

The “Square” attack is a dedicated attack on Square that exploits the byte-oriented structure of Square cipher and was published in the paper presenting the Square cipher itself [DaKnRi97]. This attack is also valid for Rijndael, as Rijndael inherits many properties from Square. We describe this attack in this section.

The attack is a chosen plaintext attack and is independent of the specific choices of ByteSub, the multiplication polynomial of MixColumn and the key schedule. It is faster than an exhaustive key search for Rijndael versions of up to 6 rounds. After describing the basic attack on 4 rounds, we will show how it can be extended to 5 and 6 rounds. For 7 rounds or more, no attacks faster than exhaustive key search have been found.

8.4.1 Preliminaries

Let a Λ-set be a set of 256 states that are all different in some of the state bytes (the *active*) and all equal in the other state bytes (the *passive*) We have

$$\forall x, y \in \Lambda: \begin{cases} x_{i,j} \neq y_{i,j} \text{ if } (i, j) \text{ active} \\ x_{i,j} = y_{i,j} \text{ else} \end{cases} .$$

Applying the transformations ByteSub or AddRoundKey on (the elements of) a Λ-set results in a (generally different) Λ-set with the positions of the active bytes unchanged. Applying ShiftRow results in a Λ-set in which the active bytes are transposed by ShiftRow. Applying MixColumn to a Λ-set does not necessarily result in a Λ-set. However, since every output byte of MixColumn is a linear combination (with invertible coefficients) of the four input bytes in the same column, an input column with a single active byte gives rise to an output column with all four bytes active.

8.4.2 The basic attack

Consider a Λ-set in which only one byte is active. We will now trace the evolution of the positions of the active bytes through 3 rounds. MixColumn of the 1st round converts the active byte to a complete column of active bytes. The four active bytes of this column are spread over four distinct columns by ShiftRow of the 2nd round. MixColumn of the 2nd round subsequently converts this to 4 columns of only active bytes. This stays a Λ-set until the input of MixColumn of the 3rd round.

Since the bytes of this (in fact, any) Λ-set, denoted by *a*, range over all possible values and are therefore balanced over the Λ-set, we have

$$\begin{aligned}\bigoplus_{b=\texttt{MixColumn}(a),a\in\Lambda} b_{i,j} &= \bigoplus_{a\in\Lambda}\left(2a_{i,j} \oplus 3a_{i+1,j} \oplus a_{i+2,j} \oplus a_{i+3,j}\right)\\ &= 2\bigoplus_{a\in\Lambda} a_{i,j} \oplus 3\bigoplus_{a\in\Lambda} a_{i+1,j} \oplus \bigoplus_{a\in\Lambda} a_{i+2,j} \oplus \bigoplus_{a\in\Lambda} a_{i+3,j}\\ &= 0\oplus 0\oplus 0\oplus 0 = 0\end{aligned}$$

Hence, all bytes at the input of the 4th round are balanced. This balance is in general destroyed by the subsequent application of ByteSub.

We assume the 4th round is a final round, i.e., it does not include a MixColumn operation. Every output byte of the 4th round depends on only one input byte of the 4th round. Let a be the output of the 4th round, *b* its output and *k* the Round Key of the 4th round. We have:

$$a_{i,j} = Sbox\left(b_{i',j'}\right) \oplus k_{i,j}\ .$$

By assuming a value for $k_{i,j}$, the value of $b_{i',j'}$ for all elements of the Λ-set can be calculated from the ciphertexts. If the values of this byte are not balanced over Λ, the assumed value for the key byte was wrong. This is expected to eliminate all but approximately 1 key value. This can be repeated for the other bytes of *k*.

8.4.3 Extension by an additional round at the end

If an additional round is added, we have to calculate the above value of $b_{i',j'}$ from the output of the 5th round instead of the 4th round. This can be done by additionally assuming a value for a set of 4 bytes of the 5th Round Key. As in the case of the 4-round attack, wrong key assumptions are eliminated by verifying that $b_{i',j'}$ is not balanced.

In this 5-round attack 2^{40} key values must be checked, and this must be repeated 4 times. Since by checking a single Λ-set leaves only 1/256 of the wrong key assumptions as possible candidates, the Cipher Key can be found with overwhelming probability with only 5 Λ-sets.

8.4.4 Extension by an additional round at the beginning

The basic idea is to choose a set of plaintexts that results in a Λ-set at the output of the 1st round with a single active S-box. This requires the assumption of values of four bytes of the Round Key that is applied before the first round.

If the intermediate state after MixColumn of the 1st round has only a single active byte, this is also the case for the input of the 2nd round. This imposes the following conditions on a column of four input bytes of MixColumn of the second round: one particular linear combination of these bytes must range over all 256 possible values (active) while 3 other particular linear combinations must be constant for all 256 states. This imposes identical conditions on 4 bytes, in different positions at the input of ShiftRow of the first round. If the corresponding bytes of the first Round Key are known, these conditions can be converted to conditions on four plaintext bytes.

Now we consider a set of 2^{32} plaintexts, such that one column of bytes at the input of MixColumn of the first round range over all possible values and all other bytes are constant.

Now, an assumption is made for the value of the 4 bytes of the relevant bytes of the first Round Key. From the set of 2^{32} available plaintexts, a set of 256 plaintexts can be selected that result in a Λ-set at the input of round 2. Now the 4-round attack can be performed. For the given key assumption, the attack can be repeated for a several plaintext sets. If the byte values of the last Round Key are not consistent, the initial assumption must have been wrong. A correct assumption for the 32 bytes of the first Round Key will result in the swift and consistent recuperation of the last Round Key.

8.4.5 Working factor and memory requirements for the attacks

Combining both extensions results in a 6 round attack. Although infeasible with current technology, this attack is faster than exhaustive key search, and therefore relevant. The working factor and memory requirements are summarised in Figure 12. For the different block lengths of Rijndael no extensions to 7 rounds faster than exhaustive key search have been found.

Attack	# Plaintexts	# Cipher executions	Memory
Basic (4 rounds)	2^9	2^9	small
Extension at end	2^{11}	2^{40}	small
Extension at beginning	2^{32}	2^{40}	2^{32}
Both Extensions	2^{32}	2^{72}	2^{32}

Figure 12: Complexity of the Square attack applied to Rijndael.

8.5 Interpolation attacks

In [JaKn97] Jakobsen and Knudsen introduced a new attack on block ciphers. In this attack, the attacker constructs polynomials using cipher input/output pairs. This attack is feasible if the components in the cipher have a compact algebraic expression and can be combined to give expressions with manageable complexity. The basis of the attack is that if the constructed polynomials (or rational expressions) have a small degree, only few cipher input/output pairs are necessary to solve for the (key-dependent) coefficients of the polynomial. The complicated expression of the S-box in GF(2^8), in combination with the effect of the diffusion layer prohibits these types of attack for more than a few rounds. The expression for the S-box is given by:

$$63 + 8f\ x^{127} + b5\ x^{191} + 01\ x^{223} + f4\ x^{239} + 25\ x^{247} + f9\ x^{251} + 09\ x^{253} + 05\ x^{254}$$

8.6 Weak keys as in IDEA

The weak keys discussed in this subsection are keys that result in a block cipher mapping with detectable weaknesses. The best known case of weak keys are those of IDEA [Da95]. Typically, this weakness occurs for ciphers in which the non-linear operations depends on the actual key value. This is not the case for Rijndael, where keys are applied using the EXOR and all non-linearity is in the fixed S-box. In Rijndael, there is no restriction on key selection.

8.7 Related-key attacks

In [Bi96], Eli Biham introduced a related-key attack. Later it was demonstrated by John Kelsey, Bruce Schneier and David Wagner that several ciphers have related-key weaknesses In [KeScWa96].

In related-key attacks, the cryptanalyst can do cipher operations using different (unknown or partly unknown) keys with a chosen relation. The key schedule of Rijndael, with its high diffusion and non-linearity, makes it very improbable that this type of attack can be successful for Rijndael.

9. Expected strength

Rijndael is expected, for all key and block lengths defined, to behave as good as can be expected from a block cipher with the given block and key lengths. What we mean by this is explained in Section 10.

This implies among other things, the following. The most efficient key-recovery attack for Rijndael is exhaustive key search. Obtaining information from given plaintext-ciphertext pairs about other plaintext-ciphertext pairs cannot be done more efficiently than by determining the key by exhaustive key search. The expected effort of exhaustive key search depends on the length of the Cipher Key and is:

- for a 16-byte key, 2^{127} applications of Rijndael;
- for a 24-byte key, 2^{191} applications of Rijndael;
- for a 32-byte key, 2^{255} applications of Rijndael.

The rationale for this is that a considerable safety margin is taken with respect to all known attacks. We do however realise that it is impossible to make non-speculative statements on things unknown.

10. Security goals

In this section, we present the goals we have set for the security of Rijndael. A cryptanalytic attack will be considered successful by the designers if it demonstrates that a security goal described herein does not hold.

10.1 Definitions of security concepts

In order to formulate our goals, some security-related concepts need to be defined.

10.1.1 The set of possible ciphers for a given block length and key length

A block cipher of block length v has $V = 2^v$ possible inputs. If the key length is u it defines a set of $U = 2^u$ permutations over $\{0,1\}^v$. The number of possible permutations over $\{0,1\}^v$ is $V!$. Hence the number of all possible block ciphers of dimensions u and v is

$$((2^v)!)^{(2^u)} \text{ or equivalently } (V!)^U .$$

For practical values of the dimensions (e.g., v and u above 40), the subset of block ciphers with exploitable weaknesses form a negligible minority in this set.

10.1.2 K-Security

Definition: *A block cipher is K-secure if all possible attack strategies for it have the same expected work factor and storage requirements as for the majority of possible block ciphers with the same dimensions. This must be the case for all possible modes of access for the adversary (known/chosen/adaptively chosen plaintext/ciphertext, known/chosen/adaptively chosen key relations...*) *and for any a priori key distribution.*

K-security is a very strong notion of security. It can easily be seen that if one of the following weaknesses apply to a cipher, it cannot be called K-secure:

- Existence of key-recovering attacks faster than exhaustive search;
- Certain symmetry properties in the mapping (e.g., complementation property);
- Occurrence of non-negligible classes of weak keys (as in IDEA);
- related-key attacks.

K-security is essentially a relative measure. It is quite possible to build a K-secure block cipher with a 5-bit block and key length. The lack of security offered by such a scheme is due to its small dimensions, not to the fact that the scheme fails to meet the requirements imposed by these dimensions. Clearly, the longer the key, the higher the security requirements.

10.1.3 Hermetic block ciphers

It is possible to imagine ciphers that have certain weaknesses and still are K-secure. An example of such a weakness would be a block cipher with a block length larger than the key length and a single weak key, for which the cipher mapping is linear. The detection of the usage of the key would take at least a few encryptions, while checking whether the key is used would only take a single encryption.

If this cipher would be used for encipherment, this single weak key would pose no problem. However, used as a component in a larger scheme, for instance as the compression function of a hash function, this property could introduce a way to efficiently generate collisions.

For these reasons we introduce yet another security concept, denoted by the term *hermetic*.

Definition: *A block cipher is hermetic if it does not have weaknesses that are not present for the majority of block ciphers with the same block and key length.*

Informally, a block cipher is hermetic if its internal structure cannot be exploited in any application.

10.2 Goal

For all key and block lengths defined, the security goals are that the Rijndael cipher is :

- K-secure;
- Hermetic.

If Rijndael lives up to its goals, the strength against any known or unknown attacks is as good as can be expected from a block cipher with the given dimensions.

11. Advantages and limitations

11.1 Advantages

Implementation aspects:

- Rijndael can be implemented to run at speeds unusually fast for a block cipher on a Pentium (Pro). There is a trade-off between table size/performance.
- Rijndael can be implemented on a Smart Card in a small amount of code, using a small amount of RAM and taking a small number of cycles. There is some ROM/performance trade-off.
- The round transformation is parallel by design, an important advantage in future processors and dedicated hardware.
- As the cipher does not make use of arithmetic operations, it has no bias towards big- or little endian processor architectures.

Simplicity of Design:

- The cipher is fully "self-supporting". It does not make use of another cryptographic component, S-boxes "lent" from well-reputed ciphers, bits obtained from Rand tables, digits of π or any other such jokes.
- The cipher does not base its security or part of it on obscure and not well understood interactions between arithmetic operations.
- The tight cipher design does not leave enough room to hide a trapdoor.

Variable block length:

- The block lengths of 192 and 256 bits allow the construction of a collision-resistant iterated hash function using Rijndael as the compression function. The block length of 128 bits is not considered sufficient for this purpose nowadays.

Extensions:

- The design allows the specification of variants with the block length and key length both ranging from 128 to 256 bits in steps of 32 bits.
- Although the number of rounds of Rijndael is fixed in the specification, it can be modified as a parameter in case of security problems.

11.2 Limitations

The limitations of the cipher have to do with its inverse:

- The inverse cipher is less suited to be implemented on a smart card than the cipher itself: it takes more code and cycles. (Still, compared with other ciphers, even the inverse is very fast)
- In software, the cipher and its inverse make use of different code and/or tables.
- In hardware, the inverse cipher can only partially re-use the circuitry that implements the cipher.

12. Extensions

12.1 Other block and Cipher Key lengths

The key schedule supports any key length that is a multiple of 4 bytes. The only parameter that needs to be defined for other key lengths than 128, 192 or 256 is the number of rounds in the cipher.

The cipher structure lends itself for any block length that is a multiple of 4 bytes, with a minimum of 16 bytes. The key addition and the ByteSub and MixColumn transformations are independent from the block length. The only transformation that depends on the block length is ShiftRow. For every block length, a specific array `C1`, `C2`, `C3` must be defined.

We define an extension of Rijndael that also supports block and key lengths between 128 and 256 bits with increments of 32 bits. The number of rounds is given by:

$$\mathbf{Nr} = \max(\mathbf{Nk}, \mathbf{Nb}) + 6.$$

This interpolates the rule for the number of rounds to the alternative block and key lengths.

The additional values of `C1`, `C2` and `C3` are specified in Table 8.

Nb	C1	C2	C3
5	1	2	3
7	1	2	4

Table 8: Shift offsets in Shiftrow for the alternative block lengths

The choice of these shift offsets is based on the criteria discussed in Section 7.4.

12.2 Another primitive based on the same round transformation

The Rijndael Round transformation has been designed to provide high multiple-round diffusion and guaranteed distributed nonlinearity. These are exactly the requirements for the state updating transformation in a *stream/hash module* such as Panama [DaCl98]. By fitting the round transformation (for **Nb**=8) in a Panama-like scheme, a stream/hash module can be built that can hash and do stream encryption about 4 times as fast as Rijndael and perform as a very powerful pseudorandom number generator satisfying all requirements cited in [KeScWaHa98].

13. Other functionality

In this section we mention some functions that can be performed with the Rijndael block cipher, other than encryption.

13.1 MAC

Rijndael can be used as a MAC algorithm by using it as the Block cipher in a CBC-MAC algorithm. [ISO9797]

13.2 Hash function

Rijndael can be used as an iterated hash function by using it as the round function. Here is one possible implementation. It is advised to use a block and key length both equal to 256 bits. The chaining variable goes into the “input” and the message block goes into the “Cipher Key”. The new value of the chaining variable is given by the old value EXORed with the cipher output.

13.3 Synchronous stream cipher

Rijndael can be used as a synchronous stream cipher by applying the OFB mode or the Filtered Counter Mode. In the latter mode, the key stream sequence is created by encrypting some type of counter using a secret key [Da95].

13.4 Pseudorandom number generator

In [KeScWaHa98] a set of guidelines are given for designing a Pseudorandom Number Generator (PRNG). There are many ways in which Rijndael could be used to form a PRNG that satisfies these guidelines. We give an example in which Rijndael with a block length of 256 and a cipher key length of 256 is used.

There are three operations:

Reset:

- The Cipher Key and “state” are reset to 0.

Seeding (and *reseeding*):

- “seed bits” are collected taking care that their total has some minimum entropy. They are padded with zeroes until the resulting string has a length that is a multiple of 256 bits.
- A new Cipher Key is computed by encrypting with Rijndael a block of seed bits using the current Cipher Key. This is applied recursively until the seed blocks are exhausted.
- The state is updated by applying Rijndael using the new Cipher Key.

Pseudorandom Number generation:

- The state is updated by applying Rijndael using the Cipher Key. The first 128 bits of the state are output as a “pseudorandom number”. This step may be repeated many times.

13.5 Self-synchronising stream cipher

Rijndael can be used as a self-synchronising stream cipher by applying the CFB mode of operation.

14. Suitability for ATM, HDTV, B-ISDN, voice and satellite

It was requested to give comments on the suitability of Rijndael to be used for ATM, HDTV, B-ISDN, Voice and Satellite. As a matter of fact, the only thing that is relevant here, is the processor on which the cipher is implemented. As Rijndael can be implemented efficiently in software on a wide range of processors, makes use of a limited set of instructions and has sufficient parallelism to fully exploit modern pipelined multi-ALU processors, it is well suited for all mentioned applications.

For applications that require rates higher than 1 Gigabits/second, Rijndael can be implemented in dedicated hardware.

15. Acknowledgements

In the first place we would like to thank Antoon Bosselaers, Craig Clapp, Paulo Barreto and Brian Gladman for their efficient ANSI-C implementations and the Cryptix team, including Paulo Barreto, for their Java implementation.

We also thank Lars Knudsen, Bart Preneel, Johan Borst and Bart Van Rompay for their cryptanalysis of preliminary versions of the cipher.

We thank Brian Gladman and Gilles Van Assche and for proof-reading this version of the documentation and providing many suggestions for improvement. Moreover, we thank all people that have brought errors and inconsistencies in the first version of this document to our attention.

We would also like to thank all other people that did efforts to efficiently implement Rijndael and all people that have expressed their enthusiasm for the Rijndael design.

Finally we would like to thank the people of the NIST AES team for making it all possible.

16. References

[Bi93] E. Biham, "New types of cryptanalytic attacks using related keys," Advances in Cryptology, Proceedings Eurocrypt'93, LNCS 765, T. Helleseth, Ed., Springer-Verlag, 1993, pp. 398-409.

[BiSh91] E. Biham and A. Shamir, "Differential cryptanalysis of DES-like cryptosystems," Journal of Cryptology, Vol. 4, No. 1, 1991, pp. 3-72.

[Da95] J. Daemen, "Cipher and hash function design strategies based on linear and differential cryptanalysis," Doctoral Dissertation, March 1995, K.U.Leuven.

[DaKnRi97] J. Daemen, L.R. Knudsen and V. Rijmen, "The block cipher Square," Fast Software Encryption, LNCS 1267, E. Biham, Ed., Springer-Verlag, 1997, pp. 149-165. Also available as http://www.esat.kuleuven.ac.be/rijmen/square/fse.ps.gz.

[DaKnRi96] J. Daemen, L.R. Knudsen and V. Rijmen, " Linear frameworks for block ciphers," to appear in Design, Codes and Cryptography.

[DaCl98] J. Daemen and C. Clapp, “Fast hashing and stream Encryption with PANAMA,” Fast Software Encryption, LNCS 1372, S. Vaudenay, Ed., Springer-Verlag, 1998, pp. 60-74.

[ISO9797] ISO/IEC 9797, "Information technology - security techniques - data integrity mechanism using a cryptographic check function employing a block cipher algorithm", International Organization for Standardization, Geneva, 1994 (second edition).

[JaKn97] T. Jakobsen and L.R. Knudsen, "The interpolation attack on block ciphers," Fast Software Encryption, LNCS 1267, E. Biham, Ed., Springer-Verlag, 1997, pp. 28-40.

[KeScWa96] J. Kelsey, B. Schneier and D. Wagner, "Key-schedule cryptanalysis of IDEA, GDES, GOST, SAFER, and Triple-DES," Advances in Cryptology, Proceedings Crypto '96, LNCS 1109, N. Koblitz, Ed., Springer-Verlag, 1996, pp. 237-252.

[KeScWaHa98] J. Kelsey, B. Schneier, D. Wagner and Chris Hall, "Cryptanalytic attacks on pseudorandom number generators," Fast Software Encryption, LNCS 1372, S. Vaudenay, Ed., Springer-Verlag, 1998, pp. 168-188.

[Kn95] L.R. Knudsen, "Truncated and higher order differentials," Fast Software Encryption, LNCS 1008, B. Preneel, Ed., Springer-Verlag, 1995, pp. 196-211.

[Kn95a] L.R. Knudsen, "A key-schedule weakness in SAFER-K64," Advances in Cryptology, Proceedings Crypto'95, LNCS 963, D. Coppersmith, Ed., Springer-Verlag, 1995, pp. 274-286.

[LaMaMu91] X. Lai, J.L. Massey and S. Murphy, "Markov ciphers and differential cryptanalysis," Advances in Cryptology, Proceedings Eurocrypt'91, LNCS 547, D.W. Davies, Ed., Springer-Verlag, 1991, pp. 17-38.

[LiNi86] R. Lidl and H. Niederreiter, Introduction to finite fields and their applications, Cambridge University Press, 1986.

[Ma94] M. Matsui, "Linear cryptanalysis method for DES cipher," Advances in Cryptology, Proceedings Eurocrypt'93, LNCS 765, T. Helleseth, Ed., Springer-Verlag, 1994, pp. 386-397.

[Ny94] K. Nyberg, "Differentially uniform mappings for cryptography," Advances in Cryptology, Proceedings Eurocrypt'93, LNCS 765, T. Helleseth, Ed., Springer-Verlag, 1994, pp. 55-64.

[Ri97] V. Rijmen, "Cryptanalysis and design of iterated block ciphers," Doctoral Dissertation, October 1997, K.U.Leuven.

17. List of Annexes

In Annex, we have included Chapter 5 of [Da95]: “Correlation and Propagation” as this lays the fundaments for the Wide Trail Strategy.

Note: In the Annex, the EXOR is denoted by + instead of ⊕.