

# The Block Cipher Rijndael

Joan Daemen<sup>1</sup> and Vincent Rijmen<sup>\*2</sup>

<sup>1</sup> Proton World Int'l  
Zweefvliegtuigstraat 10  
B-1130 Brussel, Belgium  
Daemen.J@protonworld.com

<sup>2</sup> Katholieke Universiteit Leuven, Dept. Electrical Engineering-ESAT  
Kardinaal Mercierlaan 94, B-3001 Heverlee, Belgium  
vincent.rijmen@esat.kuleuven.ac.be

**Abstract.** In this paper we present the block cipher Rijndael, which is one of the fifteen candidate algorithms for the Advanced Encryption Standard (AES). We show that the cipher can be implemented very efficiently on Smart Cards.

## 1 Introduction

The US National Institute of Standards and Technology (NIST) issued a call for an Advanced Encryption Standard (AES) to replace the current Data Encryption Standard (DES). The AES call asks for a 128-bit block cipher with a variable key length. (At least key lengths of 128, 192 and 256 bits are to be supported.) The cipher should be efficient on a Pentium platform, 8-bit processors and in hardware. Rijndael is one of the fifteen submissions that have been accepted as a candidate algorithm.

We describe the block cipher in Section 2 and discuss its Smart Card implementation in Section 3. We give some performance figures in Section 4 and we conclude in Section 5.

## 2 Description of Rijndael

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. Like SQUARE [2] and BKSQ [4], Rijndael has been designed following the wide trail strategy [1,7]. This design strategy provides resistance against linear and differential cryptanalysis. In the strategy, the round transformation is divided into different components, each with its own functionality. In this section we explain the cipher structure and the component transformations. For implementation aspects, we refer to Section 3.

---

\* F.W.O. postdoctoral researcher, sponsored by the Fund for Scientific Research – Flanders (Belgium).

### 2.1 The State, the Cipher Key, and the Number of Rounds

We define the *state* of the block cipher as the intermediate result of the encryption process. The state is initialised with the plaintext, in the order  $a_{0,0}, a_{1,0}, a_{2,0}, a_{3,0}, a_{0,1}, a_{1,1}, \dots$ . The round transformations are built from component transformations that operate on the state. Finally, at the end of the cipher operation, the ciphertext is read from the state by taking the state bytes in the same order.

The state can be pictured as a rectangular array of bytes. This array has four rows, the number of columns is denoted by  $N_b$  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  $N_k$  and is equal to the key length divided by 32. This is illustrated in Figure 1. Sometimes the Cipher Key is pictured as a linear array of four-byte words. The words consist of the four bytes that are in the corresponding column.

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

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

**Fig. 1.** Example of state layout (with  $N_b = 6$ ) and cipher key layout (with  $N_k = 4$ ).

The number of rounds is denoted by  $N_r$  and depends on the values  $N_b$  and  $N_k$ . It is given in Table 1.

**Table 1.** Number of Rounds ( $N_r$ ) as a function of the block and key length.

		$N_b$		
		4	6	8
$N_k$	4	10	12	14
	6	12	12	14
	8	14	14	14

### 2.2 The Round Transformation

The round transformation is composed of four different component 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);
}

```

It can be seen that in the final round the `MixColumn` step has been removed. The component transformations are specified in the following subsections.

**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  $\text{GF}(2^8)$  [6], the zero element is mapped onto itself.
2. Then, applying an affine transformation (over  $\text{GF}(2)$ ).

The application of the described S-box to all bytes of the state is denoted by `ByteSub(State)`.

**The ShiftRow Transformation.** In `ShiftRow`, the last three rows of the state are shifted cyclically over different offsets. Row 1 is shifted over  $C_1$  bytes, row 2 over  $C_2$  bytes and row 3 over  $C_3$  bytes. The shift offsets  $C_1$ ,  $C_2$  and  $C_3$  depend on the block length  $N_b$ . The different values are specified in Table 2.

**Table 2.** Shift offsets for different block lengths.

$N_b$	$C_1$	$C_2$	$C_3$
4	1	2	3
6	1	2	3
8	1	3	4

The operation of shifting the last three rows of the state over the specified offsets is denoted by `ShiftRow(State)`.

**The MixColumn Transformation.** In MixColumn, the columns of the state are considered as polynomials over  $\text{GF}(2^8)$ , and multiplied modulo  $x^4 + 1$  with a fixed polynomial  $c(x)$ , given by  $c(x) = 3x^3 + x^2 + x + 2$ . This can also be written as a matrix multiplication. Let  $b(x) = c(x) \otimes a(x)$ , then

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}.$$

The application of this operation on all four columns of the state is denoted by MixColumn(State).

**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  $N_b$ . The transformation that consists of EXORing a round key to the state is denoted by AddRoundKey(State, RoundKey).

### 2.3 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 principles are 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  $N_b$  words, the second one of the following  $N_b$  words, and so on.

**Key Expansion.** The expanded key is a linear array of four-byte words and is denoted by  $W[N_b(N_r + 1)]$ . The first  $N_k$  words contain the cipher key. All other words are defined recursively in terms of words with smaller indices. The key schedule depends on the value of  $N_k$ : there is a version for  $N_k \leq 6$ , and a version for  $N_k > 6$ . For  $N_k \leq 6$ , we have:

```

KeyExpansion(CipherKey,W) {
  for( $i = 0$ ;  $i < N_k$ ;  $i++$ )  $W[i] = \text{CipherKey}[i]$ ;
  for( $j = N_k$ ;  $j < N_b(N_r + 1)$ ;  $j += N_k$ ) {
     $W[j] = W[j - N_k] \oplus \text{SubByte}(\text{Rotl}(W[j - 1])) \oplus \text{Rcon}[j/N_k]$ ;
    for( $i = 1$ ;  $i < N_k$  &&  $i + j < N_b(N_r + 1)$ ;  $i++$ )
       $W[i + j] = W[i + j - N_k] \oplus W[i + j - 1]$ ;
  }
}

```

When the cipher key words are used, every following word  $W[i]$  is equal to the EXOR of the previous word  $W[i - 1]$  and the word  $N_k$  positions earlier  $W[i - N_k]$ . For words in positions that are a multiple of  $N_k$ , 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, denoted with  $\text{Rotl}$ , followed by  $\text{SubByte}$ , the application of a table lookup to all four bytes of the word.

The key expansion for  $N_k > 6$  is very similar, but uses an extra application of  $\text{SubByte}$ . It is described in detail in [3].

The round constants are independent of  $N - k$  and defined by:  $\text{Rcon}[i] = (RC[i], 0, 0, 0)$ , with  $RC[0] = 1$ ,  $RC[i] = 2 \cdot RC[i - 1]$  (multiplication in the field  $\text{GF}(2^8)$ ).

**Round Key Selection.** Round key  $i$  is given by the round key buffer words  $W[N_b i]$  to  $W[N_b(i + 1)]$ . The key schedule can be implemented without explicit use of the array  $W$ . For implementations where RAM is scarce, the round keys can be computed just-in-time using a buffer of  $N_k$  words.

## 2.4 The Cipher

The cipher Rijndael consists of

- an initial round key addition,
- $N_r - 1$  rounds,
- a final round.

In pseudo C code, this gives:

```
Rijndael(State,CipherKey) {
  KeyExpansion(CipherKey,ExpandedKey);
  AddRoundKey(State,ExpandedKey);
  for( $i = 1; i < N_r; i++$ ) Round(State,ExpandedKey +  $N_b i$ );
  FinalRound(State,ExpandedKey +  $N_b N_r$ );
}
```

The Key Expansion can be done on beforehand and Rijndael can be specified in terms of this expanded key.

```
Rijndael(State,ExpandedKey) {
  AddRoundKey(State,ExpandedKey);
  For( $i = 1; i < N_r; i++$ ) Round(State,ExpandedKey +  $N_b i$ );
  FinalRound(State,ExpandedKey +  $N_b N_r$ );
}
```

## 3 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 Smart Cards.

### 3.1 Implementation on Eight-Bit Processors

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  $\text{GF}(2^8)$ . The choice for the coefficients of the polynomial  $c(x)$  (cf. Section 2.2) has been influenced by implementation considerations. We define the 256-byte table  $X2$  as follows:  $X2[i] = i \cdot 2$ , which multiplication in the Galois Field. Thus,  $X2$  implements the multiplication with two. We can now implement the matrix multiplication very efficiently. Indeed, the matrix has only entries 1, 2 and 3. Multiplication with 3 can be done by multiplying with 2 and then adding the argument. We illustrate it for one column:

```

p = a[0] ⊕ a[1] ⊕ a[2] ⊕ a[3]; /* a is a byte array */
q = X2[a[0] ⊕ a[1]]; a[0] = a[0] ⊕ q ⊕ p;
q = X2[a[1] ⊕ a[2]]; a[1] = a[1] ⊕ q ⊕ p;
q = X2[a[2] ⊕ a[3]]; a[2] = a[2] ⊕ q ⊕ p;
q = X2[a[3] ⊕ a[0]]; a[3] = a[3] ⊕ q ⊕ p;

```

Instead of using the table  $X2$ , the multiplication with two can be implemented with a shift and a conditional exor. In a straightforward implementation, the execution time of this operation will depend on the input value. This may allow an attacker to mount a timing attack [5]. The timing attack can be countered by inserting additional NOP-operations to make the execution time of both branches equal to one another, but this will probably introduce weaknesses with respect to a power analysis attack. The use of a table effectively counters these types of attacks.

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 one or two 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  $4N_b$  bytes. The round key is updated in between rounds. All operations in this key update can be efficiently implemented on byte level. If the cipher key length is equal to the block length or an integer multiple of it, the implementation is straightforward. If this is not the case, an additional buffer pointer is required.

### 3.2 The Inverse Cipher

The round transformation of Rijndael is not a Feistel network. An advantage of a Feistel cipher is that the inverse cipher is almost the same as the cipher. Since the round transformation of a Feistel cipher is an involution, only the order of

the round keys has to be inverted. For Rijndael, this is not the case. In principle, the decryption has to be done by applying the inverses of the component transformations in inverse order.

However, the round transformation and the cipher structure have been designed to alleviate this problem partially. By using some algebraic properties we can derive an equivalent representation for the inverse cipher, that has the same *structure* as the cipher. This means that a round of the inverse cipher looks the same as a round of the cipher, except that `ByteSub`, `MixColumn` and `ShiftRow` have been replaced by their inverses. The round keys in this representation are different from the round keys used in the encryption mode.

The elements in the matrix corresponding to the inverse operation of `MixColumn`, have other values than 1, 2 and 3. Therefore, the multiplication cannot be done with the same efficiency. If we use only the table `X2`, the performance of the cipher drops with about 50%. The performance loss can be alleviated by using additional tables to define multiplications with other field elements.

## 4 Performance

Rijndael has been implemented in assembler on two different types of microprocessors that are representative for Smart Cards in use today. In these implementation 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. Only the forward operation of the cipher has been implemented, backwards operation is expected to be slower by a factor of 1.5 to 2, as explained in Section 3.2.

### 4.1 Intel 8051 Processor

Rijndael has been implemented on the Intel 8051 microprocessor, using 8051 Development tools of Keil Elektronik: `μVision 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).

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

(Key, block length)	Number of cycles	Code size (bytes)
(128,128)	4065	768
	3744	826
	3168	1016
(192,128)	4512	1125
(256,128)	5221	1041

## 4.2 Motorola 68HC08 Processor

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 optimization of code length has been attempted for this processor.

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

(Key, block length)	Number of cycles	Required RAM (bytes)	Code size (bytes)
(128,128)	8390	36	919
(192,128)	10780	44	1170
(256,128)	12490	52	1135

## 5 Conclusions

Rijndael is a very fast block cipher. It can be implemented very efficiently on a Smart Card with a small amount of code, using a small amount of RAM and taking a small number of cycles. Some ROM/performance trade-off is possible. It is easy to make the implementation of the cipher resistant to timing attacks. The variable block length allows the construction of a collision-resistant hash function with Rijndael as compression function.

The most important disadvantage is the fact that the inverse cipher is different from the cipher. The inverse cipher is typically 1.5 to 2 times slower on Smart Card (or takes more ROM).

## References

1. J. Daemen, "Cipher and hash function design strategies based on linear and differential cryptanalysis," *Doctoral Dissertation*, March 1995, K.U.Leuven.
2. 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>.
3. J. Daemen and V. Rijmen, "The Rijndael block cipher," presented at the First Advanced Encryption Standard Conference, Ventura (California), 1998, available from URL <http://www.nist.gov/aes>.
4. J. Daemen and V. Rijmen, "The block cipher BKSQ," *this volume*.
5. P.C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems," *Advances in Cryptology, Proceedings Crypto'96*, LNCS 1109, N. Kobitz, Ed., Springer-Verlag, 1996, pp. 104–113.
6. R. Lidl and H. Niederreiter, *Introduction to finite fields and their applications*, Cambridge University Press, 1986.
7. V. Rijmen, "Cryptanalysis and design of iterated block ciphers," *Doctoral Dissertation*, October 1997, K.U.Leuven.