A Specification for Rijndael, the AES Algorithm

1. Notation and Conventions

1.1 Rijndael Inputs and Outputs

The input, output and cipher key for Rijndael are sequences containing 128, 160, 192, 224 or 256 bits, where input and output sequences have the same length (called the block size). Here a bit is a binary digit (0 or 1) and 'length' refers to the number of elements in a sequence (in this case bits). In general the the block size and the key length can be any of the five allowed values but for the Advanced Encryption Standard (AES) the block size is fixed at 128 bits and the key length can only be 128, 192 or 256 bits.

Sequences and sub-sequences will be enumerated from zero to one less than the number of elements in the sequence. The number associated with a sequence element is called its index and sequences will be presented with lower numbered elements to the left. Unless specified, the enumeration order of sub-sequences will match that of the sequence from which they are derived. For Rijndael input, output and key sequences, the number i associated with a bit will hence be in one of the five ranges $0 \le i < 128$, $0 \le i < 160$, $0 \le i < 192$, $0 \le i < 224$ or $0 \le i < 256$.

1.2 Bytes

Bit sequences can be represented as one-dimensional arrays of 8-bit sub-sequences called bytes where byte n consists of the sub-sequence $8n$ to $8n+7$. In such an array, denoted by a, the n'th byte will be referred to as either a_n or $a[n]$, where n is in one of the ranges $0 \le n < 16$, $0 \le n < 20$, $0 \le n < 24$, $0 \le n < 28$ or $0 \le n < 32$. The order i of a bit within a byte has a value $7 - k$, where k is the bit's index, and the bit with order i in a byte b will be denoted by b_i . Internally bytes represent finite field elements using:

$$b_7x^7 + b_6x^6 + b_5x^5 + b_4x^4 + b_3x^3 + b_2x^2 + b_1x^1 + b_0 = \sum_0^7 b_i x^i \qquad (1.2.1)$$

in a polynomial representation. Externally bytes have their normal meanings. The external integer byte value `0x01` is mapped through the interface (i.e. without translation) to the internal finite field value `{01}` (this notation is described in section 1.3 below).

1.3 Byte Literals

The values of bytes will be presented as a concatenation of their bits between braces with higher numbered bits to the left. Hence `{011000011}` identifies a specific finite field element. It is also convenient to denote byte values using hexadecimal notation, with each of two groups of four bits being denoted by a character as follows.

bit pattern	character
0000	0
0001	1
0010	2
0011	3

bit pattern	character
0100	4
0101	5
0110	6
0111	7

bit pattern	character
1000	8
1001	9
1010	a
1011	b

bit pattern	character
1100	c
1101	d
1110	e
1111	f

Hence the value `{011000011}` can also be written as `{63}`, where the character denoting the 4-bit group containing the higher numbered bits is again to the left. Some finite field operations an extra bit (b_8) to the left of an 8-bit byte. Where this bit is present it will appear immediately to the left of the left brace as, for example, in `1{1b}`.

1.4 The Rijndael State

Internally Rijndael operates on a two dimensional array of bytes called the **state** that contains 4 rows and *Nc* columns, where *Nc* is the input sequence length divided by 32. In this state array, denoted by the symbol s, each individual byte has two indexes: its row number r, in the range $0 \le r < 4$, and its column number c, in the range $0 \le c < Nc$, hence allowing it to be referred to either as $s_{r,c}$ or $s[r, c]$. For AES the range for c is $0 \le c < 4$ since Nc has a fixed value of 4.

At the start (end) of an encryption or decryption operation the bytes of the cipher input (output) are copied to (from) this state array in the order shown in Figure 1.

cipher input bytes

in_0	in_4	in_8	in_{12}	…
in_1	in_5	in_9	in_{13}	…
in_2	in_6	in_{10}	in_{14}	…
in_3	in_7	in_{11}	in_{15}	…

→

cipher state array

$s_{0,0}$	$s_{0,1}$	$s_{0,2}$	$s_{0,3}$	…
$s_{1,0}$	$s_{1,1}$	$s_{1,2}$	$s_{1,3}$	…
$s_{2,0}$	$s_{2,1}$	$s_{2,2}$	$s_{2,3}$	…
$s_{3,0}$	$s_{3,1}$	$s_{3,2}$	$s_{3,3}$	…

→

cipher output bytes

out_0	out_4	out_8	out_{12}	…
out_1	out_5	out_9	out_{13}	…
out_2	out_6	out_{10}	out_{14}	…
out_3	out_7	out_{11}	out_{15}	…

Figure 1 – Input to, and output from, the cipher state array

Hence at the start of encryption or decryption the input array *in* is copied to the state array according to the scheme:

$$s[r, c] = in[r + 4c] \qquad \text{for } 0 \le r < 4 \text{ and } 0 \le c < Nc \qquad (1.4.1)$$

and when the cipher is complete the state is copied to the output array *out* according to:

$$out[r + 4c] = s[r, c] \qquad \text{for } 0 \le r < 4 \text{ and } 0 \le c < Nc \qquad (1.4.2)$$

1.5 Arrays of 32-bit Words

The four bytes in each column of the state can be thought of as an array of four bytes indexed by the row number r or as a single 32-bit **word** (bytes within all 32-bit words will always be enumerated using the index r). The state can hence be considered as a one-dimensional array of words for which the column number c provides the array index.

The key schedule for Rijndael, described fully in Section 4, is an array of 32-bit words, denoted by the symbol k, with the lower elements initialised from the cipher key input so that byte $4i+r$ of the key is copied into byte r of key schedule word $k[i]$. The cipher iterates through a number of cycles, called **rounds**, each of which uses Nc words from this key schedule. Hence the key schedule can also be viewed as an array of **round keys**, each of which consists of an Nc word sub-array. Hence word c of round key n, which is $k[Nc * n + c]$, will also be referred to using two dimensional array notation as either $k[n,c]$ or $k_{n,c}$. Here the round key for round n as a whole, an Nc word sub-array, will sometimes be referred to by replacing the second index with '-' as in $k[n,-]$ and $k_{n,-}$.

2. Finite Field Operations

2.1 Finite Field Addition

The addition of two finite field elements is achieved by adding the coefficients for corresponding powers in their polynomial representations, this addition being performed in GF(2), that is, modulo 2, so that 1 + 1 = 0. Consequently, addition and subtraction are both equivalent to an exclusive-or operation on the bytes that represent field elements. Addition operations for finite field elements will be denoted by the symbol ⊕. For example, the following expressions are equivalent:

$$(x^6 + x^4 + x^2 + x + 1) \ + \ (x^7 + x + 1) \equiv x^7 + x^6 + x^4 + x^2 \qquad \text{(polynomial notation)}$$

`{01010111}` ⊕ `{10000011}` ≡ `{11010100}` (binary notation)

`{57}` ⊕ `{83}` ≡ `{d4}` (hexadecimal notation)

2.2 Finite Field Multiplication

Finite field multiplication is more difficult than addition and is achieved by multiplying the polynomials for the two elements concerned and collecting like powers of x in the result. Since each polynomial can have powers of x up to 7, the result can have powers of x up to 14 and will no longer fit within a single byte.

This situation is handled by replacing the result with the remainder polynomial after division by a special eighth order **irreducible polynomial**, which, for Rijndael, is:

$$m(x) = x^8 + x^4 + x^3 + x + 1 \tag{2.2.1}$$

Since this polynomial has powers of x up to 8 it cannot be represented by a single byte and will be written as either `1{00011011}` or `1{1b}` as indicated earlier. This process is illustrated in the following example of the product `{57}` • `{83}` ≡ `{c1}` (where • is used to represent finite field multiplication):

$$\begin{array}{ll}
(x^6+x^4+x^2+x+1) \bullet (x^7+x+1) \rightarrow & \\
(x^6+x^4+x^2+x+1) \bullet x^7 = & x^{13}+x^{11}+x^9+x^8+x^7+ \\
(x^6+x^4+x^2+x+1) \bullet x = & x^7+x^5+x^3+x^2+x \\
(x^6+x^4+x^2+x+1) \bullet 1 = & x^6+x^4+x^2+x+1 \\
\cline{2-2}
 & x^{13}+x^{11}+x^9+x^8+x^6+x^5+x^4+x^3+1
\end{array}$$

This intermediate result is now divided by $m(x)$ above:

$$\begin{array}{ll}
 & x^{13}+x^{11}+x^9+x^8+x^6+x^5+x^4+x^3+1 \\
(x^8+x^4+x^3+x+1) \bullet x^5 = & x^{13}+x^9+x^8+x^6+x^5 \\
\cline{2-2}
\text{subtract to give intermediate remainder} & x^{11}+x^4+x^3+1 \\
(x^8+x^4+x^3+x+1) \bullet x^3 = & x^{11}+x^7+x^6+x^4+x^3 \\
\cline{2-2}
\text{subtract to give the final remainder} & x^7+x^6+1
\end{array}$$

Multiplication is associative, and there is a neutral element `{01}`; for any binary polynomial $b(x)$ of degree less than 8, the extended Euclidean algorithm can be used to compute polynomials $a(x)$ and $c(x)$, such that:

$$b(x) \bullet a(x) \oplus m(x) \bullet c(x) = 1 \tag{2.2.2}$$

$$a(x) \bullet b(x) \bmod m(x) = 1 \tag{2.2.3}$$

which shows that the polynomials $a(x)$ and $b(x)$ are mutual inverses. Furthermore:

$$a(x) \bullet (b(x) \oplus c(x)) = a(x) \bullet b(x) \oplus a(x) \bullet c(x) \tag{2.2.4}$$

It hence follows that the set of 256 byte values, with the XOR as addition and multiplication as defined above has the structure of the finite field GF(256).

2.3 Multiplication by Repeated Shifts

The finite field element `{00000010}` is the polynomial x, which means that multiplying another element by this value increases all it's powers of x by 1. This is equivalent to shifting its byte representation up by one bit so that the bit at position i moves to postion i+1. If the top bit is set prior to this move it will overflow to create an x^8 term, in which case the modular polynomial is added to cancel this additional bit, leaving a result that fits within a single byte.

For example, multiplying `{11001000}` by x, that is `{00000010}`, the initial result is `1{10010000}`. The 'overflow' bit is then removed by adding `1{00011011}`, the modular polynomial, using an exclusive-or operation to give a final result of `{10001011}`.

By repeating this process, a finite field element can be multiplied by all powers of x from 0 to 7. Multiplication of this element by any other field element can then be achieved by adding the results for the appropriate powers of x. For example, Table 1 carries out this calculation for the product of the field elements {57} and {83} to give {c1}.

p	{57} • x^p	⊕ $m(x)$	{57} • x^p	{83}	⊕ to result	result
0	{01010111}		{01010111}	1	{01010111}	{01010111}
1	{10101110}		{10101110}	1	{10101110}	{11111001}
2	1{01011100}	1{00011011}	{01000111}	0		
3	{10001110}		{10001110}	0		
4	1{00011100}	1{00011011}	{00000111}	0		
5	{00001110}		{00001110}	0		
6	{00011100}		{00011100}	0		
7	{00111000}		{00111000}	1	{00111000}	{11000001}

Table 1 – Finite field multiply {57} • {83}

2.4 Finite Field Multiplication Using Tables

When certain finite field elements (known as generators) are repeatedly multiplied to produce a list of their powers, g^p, they progressively generate all 255 non-zero elements in the field. When p reaches 256 the original field element recurs, indicating that g^{255} is equal to {01}. The p values for each field element can be thought of as logarithms and these provide a way of converting multiplication into addition. Hence the two elements $a = g^{\alpha}$ and $b = g^{\beta}$ have the product $a \bullet b = g^{\alpha + \beta}$. With a 'logarithm' table listing the power of the generator for each finite field element we can hence find the powers α and β corresponding to the elements a and b and add these values to find the power of g for the result. A reverse table can then be used to look up the product element.

Since the two initial power values can each be as high as 255, their sum may be greater than 255 but if this occurs, 255 can be subtracted from the value to bring it into the range of the tables because g^{255} = {01}. Although decimal exponents have been used in this explanation, all exponents in what follows are in hexadecimal notation.

L(xy)		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0		00	19	01	32	02	1a	c6	4b	c7	1b	68	33	ee	df	03
	1	64	04	e0	0e	34	8d	81	ef	4c	71	08	c8	f8	69	1c	c1
	2	7d	c2	1d	b5	f9	b9	27	6a	4d	e4	a6	72	9a	c9	09	78
	3	65	2f	8a	05	21	0f	e1	24	12	f0	82	45	35	93	da	8e
	4	96	8f	db	bd	36	d0	ce	94	13	5c	d2	f1	40	46	83	38
	5	66	dd	fd	30	bf	06	8b	62	b3	25	e2	98	22	88	91	10
	6	7e	6e	48	c3	a3	b6	1e	42	3a	6b	28	54	fa	85	3d	ba
	7	2b	79	0a	15	9b	9f	5e	ca	4e	d4	ac	e5	f3	73	a7	57
	8	af	58	a8	50	f4	ea	d6	74	4f	ae	e9	d5	e7	e6	ad	e8
	9	2c	d7	75	7a	eb	16	0b	f5	59	cb	5f	b0	9c	a9	51	a0
	a	7f	0c	f6	6f	17	c4	49	ec	d8	43	1f	2d	a4	76	7b	b7
	b	cc	bb	3e	5a	fb	60	b1	86	3b	52	a1	6c	aa	55	29	9d
	c	97	b2	87	90	61	be	dc	fc	bc	95	cf	cd	37	3f	5b	d1
	d	53	39	84	3c	41	a2	6d	47	14	2a	9e	5d	56	f2	d3	ab
	e	44	11	92	d9	23	20	2e	89	b4	7c	b8	26	77	99	e3	a5
	f	67	4a	ed	de	c5	31	fe	18	0d	63	8c	80	c0	f7	70	07

Table 2 – 'Logs' – L values such that {xy} = {03}^{L} for a given a finite field element {xy}

E(xy)		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	01	03	05	0f	11	33	55	ff	1a	2e	72	96	a1	f8	13	35
	1	5f	e1	38	48	d8	73	95	a4	f7	02	06	0a	1e	22	66	aa
	2	e5	34	5c	e4	37	59	eb	26	6a	be	d9	70	90	ab	e6	31
	3	53	f5	04	0c	14	3c	44	cc	4f	d1	68	b8	d3	6e	b2	cd
	4	4c	d4	67	a9	e0	3b	4d	d7	62	a6	f1	08	18	28	78	88
	5	83	9e	b9	d0	6b	bd	dc	7f	81	98	b3	ce	49	db	76	9a
	6	b5	c4	57	f9	10	30	50	f0	0b	1d	27	69	bb	d6	61	a3
	7	fe	19	2b	7d	87	92	ad	ec	2f	71	93	ae	e9	20	60	a0
	8	fb	16	3a	4e	d2	6d	b7	c2	5d	e7	32	56	fa	15	3f	41
	9	c3	5e	e2	3d	47	c9	40	c0	5b	ed	2c	74	9c	bf	da	75
	a	9f	ba	d5	64	ac	ef	2a	7e	82	9d	bc	df	7a	8e	89	80
	b	9b	b6	c1	58	e8	23	65	af	ea	25	6f	b1	c8	43	c5	54
	c	fc	1f	21	63	a5	f4	07	09	1b	2d	77	99	b0	cb	46	ca
	d	45	cf	4a	de	79	8b	86	91	a8	e3	3e	42	c6	51	f3	0e
	e	12	36	5a	ee	29	7b	8d	8c	8f	8a	85	94	a7	f2	0d	17
	f	39	4b	dd	7c	84	97	a2	fd	1c	24	6c	b4	c7	52	f6	01

Table 3 – 'Antilogs' – field elements {E} such that {E} = $\{03\}^{(xy)}$ given the power (xy)

For the Rijndael field {03} is a generator that yields Table 2 and Table 3. Using the previous example, Table 2 shows that {57} = $\{03\}^{(62)}$ and {83} = $\{03\}^{(50)}$ (where the brackets on the exponents identify them as hexadecimal numbers). This gives the product as {57} • {83} = $\{03\}^{(62)+(50)}$ and since (62) + (50) = (b2) in hexadecimal, Table 3 gives the result {c1}, as before. These tables can also be used to find the inverses of field elements since $g^{(x)}$ has the inverse $g^{(ff)-(x)}$. Hence the element {af} = $\{03\}^{(b7)}$ has the inverse $g^{(ff)-(b7)} = g^{(48)}$ = {62}. All elements except {00} have inverses.

2.5 Polynomials with Coefficients in GF(256)

Four term polynomials can be defined with coefficients that are finite field elements as:

$$a(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \tag{2.5.1}$$

where the four coefficients, each represented by a byte, will be denoted as a 32-bit word in the form [a_3 , a_2 , a_1 , a_0]. With a second polynomial:

$$b(x) = b_3x^3 + b_2x^2 + b_1x + b_0 \tag{2.5.2}$$

addition can be performed by adding the finite field coefficients of like powers of x, which corresponds to an XOR operation between the corresponding bytes in each of the words or an XOR of the complete 32-bit word values (note that the variable x here is different to that used in the definition of individual finite field elements).

Multiplication is achieved by algebraically expanding the polynomial product and collecting like powers of x to give:

$$c(x) = c_6x^6 + c_5x^5 + c_4x^4 + c_3x^3 + c_2x^2 + c_1x + c_0 \tag{2.5.3}$$

where:

$$\begin{aligned}
c_0 &= a_0 \bullet b_0 & c_4 &= a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3 \\
c_1 &= a_1 \bullet b_0 \oplus a_0 \bullet b_1 & c_5 &= a_3 \bullet b_2 \oplus a_2 \bullet b_3 \\
c_2 &= a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2 & c_5 &= a_3 \bullet b_3 \\
c_3 &= a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3
\end{aligned} \tag{2.5.4}$$

with $\bullet$ and $\oplus$ representing finite field multiplication and addition (XOR) respectively. This result requires six bytes to represent its coefficients but it can be reduced modulo a degree 4 polynomial to produce a result that is of degree less than 4.

In Rijndael the polynomial used is $x^4 + 1$ and reduction produces the following polynomial coefficients:

$$\begin{aligned}
d_3 &= a_3 \bullet b_0 \oplus a_2 \bullet b_1 \oplus a_1 \bullet b_2 \oplus a_0 \bullet b_3 \\
d_2 &= a_2 \bullet b_0 \oplus a_1 \bullet b_1 \oplus a_0 \bullet b_2 \oplus a_3 \bullet b_3 \\
d_1 &= a_1 \bullet b_0 \oplus a_0 \bullet b_1 \oplus a_3 \bullet b_2 \oplus a_2 \bullet b_3 \\
d_0 &= a_0 \bullet b_0 \oplus a_3 \bullet b_1 \oplus a_2 \bullet b_2 \oplus a_1 \bullet b_3
\end{aligned} \tag{2.5.5}$$

If one of the polynomials is fixed, this can conveniently be written in matrix form as:

$$\begin{bmatrix} d_3 \\ d_2 \\ d_1 \\ d_0 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \\ a_3 & a_0 & a_1 & a_2 \\ a_2 & a_3 & a_0 & a_1 \\ a_1 & a_2 & a_3 & a_0 \end{bmatrix} \begin{bmatrix} b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} \tag{2.5.6}$$

Because $x^4 + 1$ is not an irreducible polynomial, not all polynomial multiplications are invertible. For Rijndael, however, a polynomial that has an inverse has been chosen:

$$a(x) = \{03\}\, x^3 + \{01\}\, x^2 + \{01\}\, x + \{02\} \tag{2.5.7}$$

$$a^{-1}(x) = \{0b\}\, x^3 + \{0d\}\, x^2 + \{09\}\, x + \{0e\} \tag{2.5.8}$$

Another polynomial that Rijndael uses has $a_0 = a_2 = a_3 =$ `{00}` and $a_1 =$ `{01}`, which is the polynomial x. Inspection of (2.5.6) above will show that its effect is to form the output word by rotating the bytes in the input word so that $[b_3, b_2, b_1, b_0]$ is transformed into $[b_2, b_1, b_0, b_3]$, with bytes moving to higher index positions and the top byte wrapping round to the lowest position. Higher powers of x correspond to the other cyclic permutations of the four bytes within a 32-bit word. The `RotWord` function that is used in the key schedule corresponds to x^3.

3. The Cipher

At the start of the cipher the cipher input is copied into the internal state using the conventions described in Section 1.4. An initial round key is then added and the state is then transformed by iterating a **round function** in a number of cycles. The number of cycles *Nn* varies with the key length and block size. On completion the final state is copied into the cipher output using the same conventions.

The round function is parameterised using a round key which consists of an *Nc* word sub-array from the key schedule. The latter is considered either as a one-dimensional array of 32-bit words or an array of round keys with a structure and initialisation as decribed in section 1.5. In general the length of the cipher input, the cipher output and the cipher state, *Nc*, measured in multiples of 32 bits, is 4, 5, 6, 7 or 8 but the AES standard only allows a length of 4. The length of the cipher key, *Nk*, has the same values but only lengths of 4, 6 or 8 are allowed in the AES standard.

The cipher is described in the following pseudo code with the individual transformations and the key schedule described subsequently. Here the key schedule is treated as an array of *Nn* + 1 individual round keys, each of which is itself an array of *Nc* words.

```
Cipher(byte in[4*Nc], byte out[4*Nc], word k[Nn+1,Nc], Nc, Nn)
Begin
  byte state[4,Nc]   // The notation k[Nn+1,Nc] above indicates that
                     // the array k contains Nn + 1 individual round
  state = in         // keys that are each arrays of Nc words

  XorRoundKey(state, k[0,-], Nc)       // k[0,-] = k[0..Nc-1]

  for round = 1 step 1 to Nn – 1
    SubBytes(state, Nc)
    ShiftRows(state, Nc)
    MixColumns(state, Nc)
    XorRoundKey(state, k[round,-], Nc) // k[round*Nc..(round+1)*Nc-1]
  end for

  SubBytes(state, Nc)
  ShiftRows(state, Nc)
  XorRoundKey(state, k[Nn,-], Nc)      // k[Nn*Nc..(Nn+1)*Nc-1]

  out = state
end
```

The number of rounds for the cipher (*Nn*) varies with the block length and the key length as shown in the following table. Remember that for AES the block size, *Nc*, is fixed at 4 and the key, *Nk*, can only have the lengths 4, 6 or 8.

Nn	*The larger of Nc or Nk*				
	4	5	6	7	8
	10	11	12	13	14

Table 4 – The number of rounds as a function of block and key size

3.1 The SubBytes Transformation

The `SubBytes` transformation is a non-linear byte substitution that acts on every byte of the state in isolation to produce a new byte value using an S-box substitution table. The action of this transformation is illustrated in Figure 2 for a block size of 6.

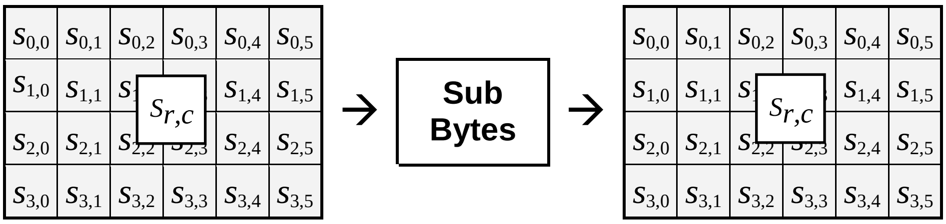

Figure 2 – `SubBytes` acts on every byte in the state in isolation

This substitution, which is invertible, is constructed by composing **two** transformations:

1. First the multiplicative inverse in the finite field described earlier (with element `{00}` mapped to itself).
2. Second the affine transformation over GF(2) defined by:

$$b_i' = b_i \oplus b_{(i+4)\bmod 8} \oplus b_{(i+5)\bmod 8} \oplus b_{(i+6)\bmod 8} \oplus b_{(i+7)\bmod 8} \oplus c_i \qquad (3.1.1)$$

for $0 \le i < 8$ where b_i is bit i of the byte and c_i is bit i of a byte c with the value `{63}` or `{01100011}`. Here and elsewhere a prime on a variable on the left of an equation indicates that its value is to be updated with the value on the right.

In matrix form the latter component of the S-box transformation can be expressed as:

$$\begin{bmatrix} b_7' \\ b_6' \\ b_5' \\ b_4' \\ b_3' \\ b_2' \\ b_1' \\ b_0' \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix} \qquad (3.1.2)$$

The final result of this two stage transformation is given in the following table.

hex		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
	5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
	6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
	7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
	8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
	9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
	a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
	b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
	c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
	d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
	e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
	f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Table 5 – The Substitution Table – Sbox[xy] (in hexadecimal)

The pseudo code for this transformation is as follows.

```
SubBytes(byte state[4,Nc], Nc)
begin
   for r = 0 step 1 to 3
      for c = 0 step 1 to Nc - 1
         state[r,c] = Sbox[state[r,c]]
      end for
   end for
end
```

3.2 The ShiftRows Transformation

The `ShiftRows` transformation operates individually on each of the last three rows of the state by cyclically shifting the bytes in the row such that:

$$s'_{r,c} = s_{r,[c+h(r,Nc)] \bmod Nc} \text{ for } 0 \le c < Nc \text{ and } 0 < r < 4 \qquad (3.2.1)$$

where the shift amount $h(r, Nc)$ depends on row number r and block length as follows:

***h*(*r*, *Nc*)**		*row (r)*		
		1	2	3
Nc	4, 5, 6	1	2	3
	7	1	2	4
	8	1	3	4

Note that the AES block size, *Nc*, is fixed at 4

Table 6 – Shift offsets for different rows and block lengths

This has the effect of moving bytes to lower positions in the row except that the lowest bytes wrap around into the top of the row (note that a prime on a variable indicates an updated value). The action of this transformation is illustrated in Figure 3 for a cipher block size of 6.

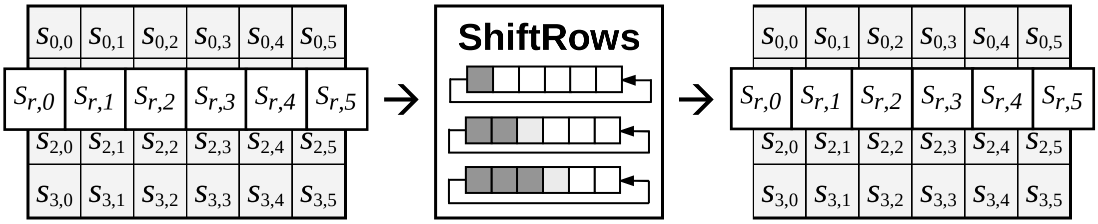

Figure 3 – `ShiftRows` acts independently on rows in the state

The pseudo code for this transformation is as follows.

```
ShiftRows(byte state[4,Nc], Nc)
begin
   byte t[Nc]
   for r = 1 step 1 to 3
      for c = 0 step 1 to Nc - 1
         t[c] = state[r, (c + h(r,Nc)) mod Nc]
      end for
      for c = 0 step 1 to Nc – 1
         state[r,c] = t[c]
      end for
   end for
end
```

3.3 The MixColumns Transformation

The `MixColumns` transformation acts independently on every column of the state and treats each column as a four-term polynomial as described in Section 2.6.

In matrix form the transformation used given in equation (3.3.1), where all the values are finite field elements as discussed in Section 2.

$$\begin{bmatrix} s'_{3,c} \\ s'_{2,c} \\ s'_{1,c} \\ s'_{0,c} \end{bmatrix} = \begin{bmatrix} 02 & 01 & 01 & 03 \\ 03 & 02 & 01 & 01 \\ 01 & 03 & 02 & 01 \\ 01 & 01 & 03 & 02 \end{bmatrix} \begin{bmatrix} s_{3,c} \\ s_{2,c} \\ s_{1,c} \\ s_{0,c} \end{bmatrix} \text{ for } 0 \le c < Nc \qquad (3.3.1)$$

The action of this transformation is illustrated in Figure 4 for a cipher block size of 6.

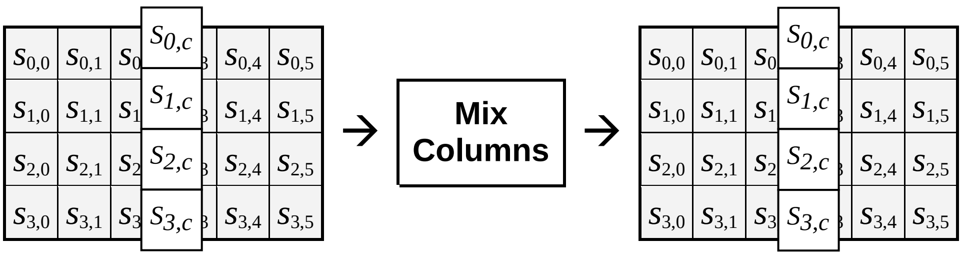

Figure 4 – `MixColumns` acts independently on each column in the state

The pseudo code for this transformation is as follows, where the function `FFmul(x, y)` returns the product of two finite field elements *x* and *y*.

```
MixColumns(byte state[4,Nc], Nc)
begin
   byte t[4]
   for c = 0 step 1 to Nc – 1
      for r = 0 step 1 to 3
         t[r] = state[r,c]
      end for
      for r = 0 step 1 to 3
         state[r,c] = FFmul(0x02, t[r]) xor
                      FFmul(0x03, t[(r + 1) mod 4]) xor
                      t[(r + 2) mod 4] xor t[(r + 3) mod 4]
      end for
   end for
end
```

3.4 The XorRoundKey Transformation

In the `XorRoundKey` transformation *Nc* words from the key schedule (the round key described later) are each added (XOR'd) into the columns of the state so that:

$$[b'_{3c}, b'_{2c}, b'_{1c}, b'_{0c}] = [b_{3c}, b_{2c}, b_{1c}, b_{0c}] \oplus [k_{round,c}] \text{ for } 0 \le c < Nc \tag{3.4.1}$$

where the round key words $k_{round,c}$ (shortened to k_c^r in the diagram below) will be described later. The round number, *round*, is in the range $0 \le round \le Nn$, with the value of 0 being used to denote the initial round key that is applied before the round function.

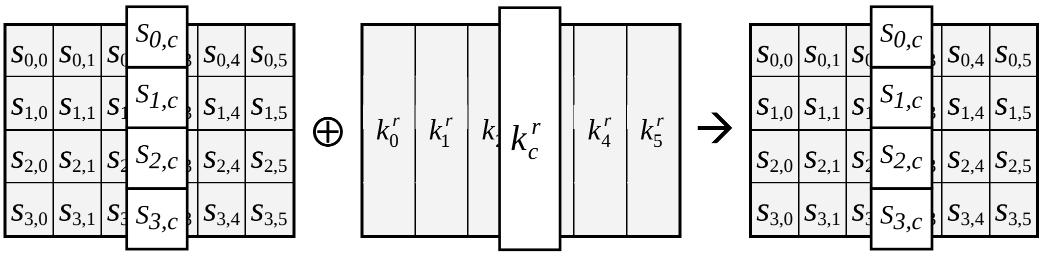

Figure 5 – Words from the key schedule are XOR'd into columns in the state

The action of this transformation is illustrated in Figure 5 for a cipher block size of 6. The byte address within each word of the key schedule is that described in Section 1.

The pseudo code for this transformation is as follows, where `xbyte(r, w)` extracts byte *r* from word *w*.

```
XorRoundKey(byte state[4,Nc], word k[round,-], Nc)
Begin
for c = 0 step 1 to Nc – 1
      for r = 0 step 1 to 3
         state[r,c] = state[r,c] xor xbyte(r, k[round,c])
      end for
   end for
end
```

4. The Key Schedule

The round keys are derived from the cipher key by means of a key schedule with each round requiring *Nc* words of key data which, with an additional initial set, makes a total of $Nc(Nn + 1)$ words, where *Nn* is the number of cipher rounds. This key schedule is considered either as a one dimensional array *k* of $Nc(Nn + 1)$ 32-bit words with an index *i* in the range $0 \le i < Nc(Nn + 1)$ or as a two dimensional array $k[n,c]$ of $Nn + 1$ round keys, each or which individually consists of a sub-array of *Nc* words.

The expansion of the input key into the key schedule proceeds according to the following pseudo code. The function `SubWord(x)` gives an output word for which the S-box substitution has been individually applied to each of the four bytes of its input `x`. The function `RotWord(x)` converts an input word $[b_3, b_2, b_1, b_0]$ to an output $[b_0, b_3, b_2, b_1]$. The word array `Rcon[i]` contains the values `[0, 0, 0,` x^{i-1}`]` with x^{i-1} being the powers of x in the field GF(256) discussed in section 2.3 (note that the index i starts at 1).

```
KeyExpansion(byte key[4*Nk], word k[Nn+1,Nc], Nc, Nk, Nn)
begin
   i = 0
   while (i < Nk)
      k[i] = word [ key[4*i+3], key[4*i+2], key[4*i+1], key[4*i] ]
      i = i + 1
   end while

   i = Nk
   while (i < Nc * (Nn + 1))
      word temp = k[i - 1]

      if (i mod Nk = 0)
         temp = SubWord(RotWord(temp)) xor Rcon[i / Nk]
      else if ((Nk > 6) and (i mod Nk = 4))
         temp = SubWord(temp)
      end if

      k[i] = k[i - Nk] xor temp
      i = i + 1
   end while
end
```

Note that this key schedule, which is illustrated in Figure 6 for *Nk* = 4 and *Nc* = 6, can be generated 'on-the fly' if necessary using a buffer of *max*(*Nc*, *Nk*) words. It can also be split into separate, somewhat simpler, key schedules for $Nk \le 6$ and $Nk > 6$ respectively.

k_0	k_1	k_2	k_3	k_4	k_5	k_4	k_7	k_8	k_9	k_{10}	k_{11}	k_{12}	k_{13}	k_{14}	k_{15}	k_{16}	k_{17}	…

round Key 0 – *k*[0,-]	round key 1 – *k*[1,-]	round key 2 – *k*[2,-]	…

Figure 6 – The key schedule and round key selection for *Nk* = 4 and *Nc* = 6

5. The Inverse Cipher

The inversion of the cipher code presented in section 3 is straightforward and provides the following pseudo code for the inverse cipher.

```
InvCipher(byte in[4*Nc], byte out[4*Nc], word k[Nn+1,Nc], Nc, Nn)
begin
   byte state[4,Nc]

   state = in

   XorRoundKey(state, k[Nn,-], Nc)      // k[Nn*Nc..(Nn+1)*Nc-1]

   for round = Nn - 1 step -1 to 1
      InvShiftRows(state, Nc)
      InvSubBytes(state, Nc)
      XorRoundKey(state, k[round,-], Nc) // k[round*Nc..(round+1)*Nc-1]
      InvMixColumns(state, Nc)
   end for

   InvShiftRows(state, Nc)
   InvSubBytes(state, Nc)
   XorRoundKey(state, k[0,-], Nc)       // k[0..Nc-1]

   out = state
end
```

5.1 The Inverse ShiftRows Transformation

The `InvShiftRows` transformation operates individually on each of the last three rows of the state cyclically shifting the bytes in the row such that:

$$s'_{r,[c+h(r,Nc)] \bmod Nc} = s_{r,c} \text{ for } 0 \le c < Nc \text{ and } 0 < r < 4 \tag{5.1.1}$$

where the cyclic shift values $h(r, Nc)$ are given in Table 6. The pseudo code for this transformation is as follows.

```
InvShiftRows(byte state[4,Nc], Nc)
begin
   byte t[Nc]
   for r = 1 step 1 to 3
      for c = 0 step 1 to Nc - 1
         t[(c + h(r,Nc)) mod Nc] = state[r,c]
      end for
      for c = 0 step 1 to Nc – 1
         state[r,c] = t[c]
      end for
   end for
end
```

5.2 The Inverse SubBytes Transformation

The inverse S-box table needed for the inverse `InvSubBytes` transformation is given in Section 3.1. The pseudo code for this transformation is as follows:

```
InvSubBytes(byte state[4,Nc], Nc)
begin
   for r = 0 step 1 to 3
      for c = 0 step 1 to Nc - 1
         state[r,c] = InvSbox[state[r,c]]
      end for
   end for
end
```

Table 7 gives the full inverse S-box, the inverse of the affine tranformation (3.1.1) being:

$$b'_i = b_{(i+2) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus d_i \text{ , where byte } d = \{05\} \tag{5.2.1}$$

hex		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
	0	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
	1	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
	2	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
	3	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
	4	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
	5	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
	6	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
x	7	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
	8	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
	9	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
	a	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
	b	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
	c	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
	d	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
	e	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
	f	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Table 7 – The Inverse Substitution Table – InvSbox[xy] (in hexadecimal)

5.3 The Inverse XorRoundKey Transformation

The `XorRoundKey` transformation is its own inverse.

5.4 The Inverse MixColumns Transformation

The `InvMixColumns` transformation acts independently on every column of the state and treats each column as a four-term polynomial as described in Section 2.6. In matrix form the transformation used given in equation (5.4.1), where all the values are finite field elements as discussed in Section 2.

$$\begin{bmatrix} s'_{3c} \\ s'_{2c} \\ s'_{1c} \\ s'_{0c} \end{bmatrix} = \begin{bmatrix} 0e & 09 & 0d & 0b \\ 0b & 0e & 09 & 0d \\ 0d & 0b & 0e & 09 \\ 09 & 0d & 0b & 0e \end{bmatrix} \begin{bmatrix} s_{3c} \\ s_{1c} \\ s_{1c} \\ s_{0c} \end{bmatrix} \text{ for } 0 \le c < Nc \qquad (5.4.1)$$

The pseudo code for this transformation is as follows, where the function `FFmul(x, y)` returns the product of two finite field elements *x* and *y*.

```
InvMixColumns(byte block[4,Nc], Nc)
begin
   byte t[4]
   for c = 0 step 1 to Nc – 1
      for r = 0 step 1 to 3
         t[r] = block[r,c]
      end for
      for r = 0 step 1 to 3
         block[r,c] =
               FFmul(0x0e, t[r]) xor
               FFmul(0x0b, t[(r + 1) mod 4]) xor
               FFmul(0x0d, t[(r + 2) mod 4]) xor
               FFmul(0x09, t[(r + 3) mod 4])
      end for
   end for
end
```

5.5 The Equivalent Inverse Cipher

The inverse cipher uses the same key schedule as the forward cipher (in reverse) but its form is different. However a series of transformations can be applied to transform the inverse cipher to match the form of the forward cipher. This is possible because the order of some operations in the inverse cipher can be changed without changing the final result.

For example the order of the `SubBytes` and `ShiftRows` transformations does not matter because `SubBytes` changes the value of bytes without changing their positions whereas `ShiftRows` does the exact opposite. Moreover, the order of the `XorRoundKey` and `InvMixColumns` operations can be inverted to put the forward and inverse ciphers in the same form provided that an adjustment is made to the key schedule. The order of round key addition and column mixing can be changed because the column mixing operation is linear with respect to the column input so that:

```
InvMixColumns(state xor k)=InvMixColumns(state) xor InvMixColumns(k)
```

where `k` represents a round key in the form of a state array. Hence, provided that an inverse column mixing operation is performed on appropriate words (columns) of the decryption key schedule, the order of these transformations can be reversed during decryption. Note, however, that this operation is not be performed on the first and last round keys (the first and last *Nc* words of the key schedule) since these do not operate in association with the column-mixing step.

The importance of this transformation is that the structure of the forward cipher allows the round function to be expressed in an efficient form for implementation. By transforming the inverse cipher into the same sequence of operations as the cipher itself, it can be implemented in the same way, thereby achieving this efficiency.

In this modified form the inverse cipher is as follows (with the modified decryption key schedule in the word array `dk[Nn+1,Nc]`).

```
EqInvCipher(byte in[4*Nc], byte out[4*Nc], word dk[Nn+1,Nc], Nc, Nn)
begin
  byte state[4,Nc]

  state = in

  XorRoundKey(state, dk[Nn,-], Nc)      // dk[Nn*Nc..(Nn+1)*Nc-1]

  for round = Nn - 1 step -1 to 1
    InvSubBytes(state, Nc)
    InvShiftRows(state, Nc)
    InvMixColumns(state, Nc)
    XorRoundKey(state, dk[round,-], Nc) // dk[round*Nc..(round+1)*Nc-1]
  end for

  InvSubBytes(state, Nc)
  InvShiftRows(state, Nc)
  XorRoundKey(state, dk[0,-], Nc)       // dk[0..Nc-1]

  out = state
end
```

where the following pseudo code is added to the end of the key expansion step (this can be made more efficient if encryption and decryption are not required simultaneously).

```
for round = 0 step 1 to Nn
  dk[i,-] = k[i,-]              // copy Nc words at a time
end for

for round = 1 step 1 to Nn – 1
  InvMixColumns(dk[round,-])    // note implicit change of type
end for
```

Note that, since `InvMixColumns` operates on a two-dimensional array of bytes while the round keys are held in an array of words, the call to `InvMixColumns` in this pseudo code sequence involves a change of type. This requires care with byte order conventions.

6. Implementation Issues

6.1 Implicit Assumptions

While hardware implementations of Rijndael can treat the input, output and cipher key inputs as bit sequences, software implementations will almost always to treat these entities as arrays of 8-bit bytes. Equally, while a hardware implementation will have to include a description of how Rijndael inputs and outputs are interfaced, a software implementation will often operate in an environment where Rijndael's two key enumerations – the enumeration of bits within 8-bit bytes and the enumeration of bytes within arrays – are already defined.

Where the environment in which Rijndael is implemented provides both for 8-bit bytes as addressable entities and for the enumeration of bits within bytes, it is reasonable to assume that Rijndael inputs and outputs will comply with these conventions.

In consequence Rijndael implementations in software should either indicate that this assumption is correct or alternatively undertake one of the following:

(a) convert inputs and outputs to (or from) these standard formats to those being used internally;

(b) document the interface to ensure that users of the implementation know that the inputs and outputs are in non-standard formats.

6.2 Bit Enumerations

In processing bytes to undertake finite field multiplication it is useful to define a function to multiply by *x*, an operation that involves shifting the value of a byte by one and then performing a conditional XOR operation. If by convention bit 0 is the 'lowest' bit in a byte (i.e. it represents a numeric value of 1) then multiplying by *x* will correspond to a left shift. This is the most likely situation but it is not unknown for bit 0 to be designated as the 'highest' bit in a byte , the bit that represents a numeric value of 128 in decimal, in which case multiplication by *x* will correspond to a right shift. When this applies, all byte values will also have their bits reversed so that `{01100011}` or `{63}`, which in former convention would be associated with a numeric value of `0x63` in hexadecimal, will instead be associated with a numeric value of `0xc6`. For this reason the terms 'left' and 'right' when referring to shifts have been avoided in this specification by using the terms 'up' and 'down' to refer to operations in which bytes at an index position move to higher or lower index positions respectively.

6.3 Bytes Within Words

A number of Rijndael operations involve the manipulation of the four 8-bit bytes within a 32-bit word, one such operation being the cyclic shift (rotation) of these four bytes into new positions. Whether the operation of moving bytes to higher array index positions correponds to a cyclic left or a cyclic right shift for a 32-bit word will depend on how the bytes are organised within words.

On some ('little-endian') processors bytes are numbered upwards from the 'low' end of 32-bits words and this means that a cyclic shift of bytes to higher array index positions will correspond to a cyclic left shift. But on other ('big-endian') processors bytes are numbered upwards starting at the 'high' end of a word so that a cyclic shift to higher index positions corresponds to a cyclic right shift.

In consequence care is needed in implementing Rijndael to ensure that the right directions of shifts and rotates are employed for the processor or processors for which an implementation is being designed.

In general these issues can be tackled either by the conversion of input and output values before use or by ensuring that the conventions employed for implementation are those of the architecture on which the cipher will operate.

7. Implementation Techniques

In the pseudo code in this section the following symbols will be used:

`&`	bits in result are the AND of the corresponding bits in the two operands
`|`	bits in result are the OR of the corresponding bits in the two operands
`^`	bits in result are the XOR of the corresponding bits in the two operands
`>>`	right shift of left operand by amount given by right operand
`<<`	left shift of left operand by amount given by right operand
`<>`	not equal
`0x...`	hexadecimal value

7.1 Finite Field Multiplication

The basic technique for finite field multiplication is explained in Section 2.4 and is implemented as follows:

```
byte FFmul(const byte a, const byte b)
begin
  byte aa = a, bb = b, r = 0, t
  while (aa <> 0)
    if ((aa & 1) <> 0)
      r = r ^ bb
    endif
    t = bb & 0x80
    bb = bb << 1
    if (t <> 0)
      bb = bb ^ 0x1b      // top bit of field polynomial (0x11b) is not
    endif                 // needed here since bb is an 8 bit value
    aa = aa >> 1
  endwhile
  return r
end
```

But this approach can be quite slow compared with table lookup using the techniques described in Section 2.5. With a 256-byte arrays from tables 2 and 3 we obtain:

```
byte FFlog[256]     // array from table 2
byte FFpow[256]     // array from table 3

byte FFmul(const byte a, const byte b)
begin
  if ((a <> 0) and (b <> 0))
    word t = FFlog[a] + FFlog[b]
    if(t >= 255)
      t = t – 255
    endif
    return FFpow[t]
  else
    return 0
  endif
end
```

This can be speeded up by doubling the length of the `FFpow[]` array and setting the values for elements 255 to 509 to the same values as elements 0 to 254 respectively so that `FFmul()` can be coded as:

```
byte FFmul(const byte a, const byte b)
begin
  if ((a <> 0) and (b <> 0))
    return FFpow[FFlog[a] + FFlog[b]]
  else
    return 0
  endif
end
```

In practice many compilers will allow these functions to be specified as inline code and this makes finite field multiplication very efficient.

7.2 Column Mixing

Provided that the state array is arranged appropriately in memory, each of the columns will be a single 32-bit word. If the bytes in such a word are $c[0]$ to $c[3]$ then the mixing operation is:

$$\begin{aligned} c[0]' &= \{02\} \bullet c[0] \oplus \{03\} \bullet c[1] \oplus c[2] \oplus c[3] \\ c[1]' &= \{02\} \bullet c[1] \oplus \{03\} \bullet c[2] \oplus c[3] \oplus c[0] \\ c[2]' &= \{02\} \bullet c[2] \oplus \{03\} \bullet c[3] \oplus c[0] \oplus c[1] \\ c[3]' &= \{02\} \bullet c[3] \oplus \{03\} \bullet c[0] \oplus c[1] \oplus c[2] \end{aligned} \tag{7.2.1}$$

where the bytes are updated with the values on the left at the end of this sequence. But since $\{03\} \bullet c[0] = \{02\} \bullet c[0] \oplus c[0]$, this can also be written as:

$$
\begin{aligned}
c[0]' &= \mathrm{c}[0] \oplus t \oplus \{02\} \bullet (c[0] \oplus c[1]) \\
c[1]' &= \mathrm{c}[1] \oplus t \oplus \{02\} \bullet (c[1] \oplus c[2]) \\
c[2]' &= \mathrm{c}[2] \oplus t \oplus \{02\} \bullet (c[2] \oplus c[3]) \\
c[3]' &= \mathrm{c}[3] \oplus t \oplus \{02\} \bullet (c[3] \oplus c[0])
\end{aligned} \tag{7.2.2}
$$

where $t = c[0] \oplus c[1] \oplus c[2] \oplus c[3]$. When the need for temporary storage is taken into account, this code sequence becomes:

```
t = c[0] ^ c[1] ^ c[2] ^ c[3]
u = c[0] ^ t ^ FFmul(0x02, c[0] ^ c[1])
c[1] = c[1] ^ t ^ FFmul(0x02, c[1] ^ c[2])
c[2] = c[2] ^ t ^ FFmul(0x02, c[2] ^ c[3])
c[3] = c[3] ^ t ^ FFmul(0x02, c[3] ^ c[0])
c[0] = u
```

Morover, multiplication by the element {02} is just a shift followed by a conditional exclusive-or operation.

Although this formulation is quite efficient on 8-bit processors, the operations can be speeded up considerably on processors with 32 bit words provided that there are operations that can cyclicly rotate the bytes within such words. The functions required are as follows:

`rot1(w)` moves the bytes in positions 0, 1 and 2 in the word `w` to positions 1, 2 and 3 respectively and moves the byte in position 3 to position 0.

`rot2(w)` moves the bytes in positions 0, 1, 2 and 3 in `w` to positions 2, 3, 0 and 1 respectively (or exchanges byte 0 with byte 2 and byte 1 with byte 3).

`rot3(w)` moves the bytes in positions 1, 2 and 3 in `w` to positions 0, 1 and 2 respectively and moves the byte in position 0 to position 3.

Using these operations on each word `w` of the state allows the above code sequence on individual bytes to be rewritten as one operation on each word (column) as a whole:

```
w = rot3(w) ^ rot2(w) ^ rot1(w) ^ FFmulX(w ^ rot3(w))
```

where the function `FFmulX(w)` performs a finite field multiplication of each of the four bytes in the word `w` by {02}. This itself can be coded to operate in parallel on the four bytes in the word as follows:

```
word FFmulX(const word w)
begin
  word t = w & 0x80808080
  return ((w ^ t) << 1) ^ ((t >> 3) | (t >> 4) | (t >> 6) | (t >> 7))
end
```

Here the word `t` extracts the highest bits from each byte within `w`, while the term `w^t` extracts the lower 7 bits. The four individual bytes within the latter can then be multiplied by {02} in parallel using a single 32-bit left shift without creating overflows from one byte to the next. The `((t >> 3) | (t >> 4) | (t >> 6) | (t >> 7))` construction leaves zero bytes within `t` unchanged but changes the bytes whose top bits are set to `0x1b`. There are several alternative ways of performing this step including, for example `((u - (u >> 7)) & 0x1b1b1b1b)` or `((u >> 7) * 0x0000001b)`, the most efficient depending on the characteristics of the processor instruction set available for its implementation. Finally, when this value is XOR'ed into the result the effect is that required – namely, the modular polynomial is added to all bytes in which the top bits were originally set.

Similar techniques can be used to speed up the inverse column mixing operation.

7.3 Implementation Using Tables

Rijndael can be implemented very efficiently on processors with 32-bit words by transforming it in the following way.

Considering a single column (word) of the state and applying the `SubBytes`, `ShiftRows`, `MixColumns` and `XorRoundKey` transformations in turn gives:

after `SubBytes`:
$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} S[s_{0,c}] \\ S[s_{1,c}] \\ S[s_{2,c}] \\ S[s_{3,c}] \end{bmatrix} \tag{7.3.1}$$

after `ShiftRows`:
$$\begin{bmatrix} s''_{0,c} \\ s''_{1,c} \\ s''_{2,c} \\ s''_{3,c} \end{bmatrix} = \begin{bmatrix} S[s_{0,c}] \\ S[s_{1,[c+h(1,Nc)]\bmod Nc}] \\ S[s_{2,[c+h(2,Nc)]\bmod Nc}] \\ S[s_{3,[c+h(3,Nc)]\bmod Nc}] \end{bmatrix} = \begin{bmatrix} S[s_{0,c(0)}] \\ S[s_{1,c(1)}] \\ S[s_{2,c(2)}] \\ S[s_{3,c(3)}] \end{bmatrix} \tag{7.3.2}$$

after `MixColumns`:
$$\begin{bmatrix} s'''_{0,c} \\ s'''_{1,c} \\ s'''_{2,c} \\ s'''_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S[s_{0,c(0)}] \\ S[s_{1,c(1)}] \\ S[s_{2,c(2)}] \\ S[s_{3,c(3)}] \end{bmatrix} \tag{7.3.3}$$

after `XorRoundKey`:
$$\begin{bmatrix} s''''_{0,c} \\ s''''_{1,c} \\ s''''_{2,c} \\ s''''_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} S[s_{0,c(0)}] \\ S[s_{1,c(1)}] \\ S[s_{2,c(2)}] \\ S[s_{3,c(3)}] \end{bmatrix} \oplus \begin{bmatrix} k_{0,c} \\ k_{1,c} \\ k_{2,c} \\ k_{3,c} \end{bmatrix} \tag{7.3.4}$$

where the shorthand notation $c(r) = [c + h(r, Nc)] \bmod Nc$, with $c(0) = c$, has been used in the column index c.

Treating this as one complex transformation (i.e. with a single prime), it can be written in column vector form as:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = S[s_{0,c(0)}] \bullet \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix} \oplus S[s_{1,c(1)}] \bullet \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix} \oplus S[s_{2,c(2)}] \bullet \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix} \oplus S[s_{3,c(3)}] \bullet \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix} \oplus \begin{bmatrix} k_{0,c} \\ k_{1,c} \\ k_{2,c} \\ k_{3,c} \end{bmatrix} \tag{7.3.5}$$

And if four tables each of 256 32-bit words are defined (for $0 \le x < 256$) as follows:

$$T_0[x] = \begin{bmatrix} 02 \bullet S[x] \\ S[x] \\ S[x] \\ 03 \bullet S[x] \end{bmatrix} \quad T_1[x] = \begin{bmatrix} 03 \bullet S[x] \\ 02 \bullet S[x] \\ S[x] \\ S[x] \end{bmatrix} \quad T_2[x] = \begin{bmatrix} S[x] \\ 03 \bullet S[x] \\ 02 \bullet S[x] \\ S[x] \end{bmatrix} \quad T_3[x] = \begin{bmatrix} S[x] \\ S[x] \\ 03 \bullet S[x] \\ 02 \bullet S[x] \end{bmatrix} \tag{7.3.6}$$

equation (6.3.5) can then be expressed in the form:

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = T_0[s_{0,c(0)}] \oplus T_1[s_{1,c(1)}] \oplus T_2[s_{2,c(2)}] \oplus T_3[s_{3,c(3)}] \oplus k_{round,c} \tag{7.3.7}$$

where $c(r) = [c + h(r, Nc)] \bmod Nc$, $c(0) = c$ and $k_{round,c}$ is word c of round key *round*.

This shows that each column in the output state can be computed using four XOR instructions involving a word from the key schedule and four words from tables that are indexed using four bytes from the input state.

Equation (6.3.7) applies to all but the last round because the latter is different in that the `MixColumns` step is not present. This means that different tables are required for the last round as follows:

$$U_0[x]=\begin{bmatrix}S[x]\\0\\0\\0\end{bmatrix}\ U_1[x]=\begin{bmatrix}0\\S[x]\\0\\0\end{bmatrix}\ U_2[x]=\begin{bmatrix}0\\0\\S[x]\\0\end{bmatrix}\ U_3[x]=\begin{bmatrix}0\\0\\0\\S[x]\end{bmatrix} \qquad (7.3.8)$$

These tables can be implemented directly or can be computed either from the S-Box table or by masking the appropriate tables for normal rounds.

The tables for the main rounds amount to 4 kbytes of table space and this is doubled if the last round tables are also implemented. However, it is worth noting that these tables are closely related since $T_i(x)$ = `rot1(`$T_{i-1}(x)$`)`, and this means that the table space can be reduced by a factor of four at the expense of three additional rotations in the calculation of each column of the state.

This implementation technique can also be used for the equivalent inverse cipher since it has the same form as the forward cipher. This requires another set of tables since the inverse S-Boxes have to be used in the above transformations. The byte indexing for the table values is also different for the inverse cipher – $c(r)=[c-h(r,Nc)+Nc] \bmod Nc$.

8. Acknowledgements

This specification was originally written as an input to the AES FIPS development process but has been developed further since then as a result of comments received on the original version. I would like to acknowledge and thank Joan Daemen and Vincent Rijmen for many significant inputs that they made during its development. I would also like to thank both Jim Foti and Elaine Barker of NIST for their many helpful comments and suggestions, many of which are embodied both here and in the FIPS. My thanks also go to Paulo Barreto for his cooperation in publishing the original development test vectors and to Lawrence Bassham of NIST for independently checking their correctness. In respect of version 3.4 of this document, my thanks also go to Bryan Olson for a mapping idea used in section 1.2 and to Doug Gwynn and David Hopwood for prompting me to add a byte oriented external interface definition.

9. References

J. Daemen and V. Rijmen, *AES Proposal: Rijndael*, AES Algorithm Submission, September 3, 1999, available from the US National Institute of Standards and Technology (NIST) AES web site at http://csrc.nist.gov/encryption/aes/

10. Errors

This specification has been produced from the base document referenced in section 9 above. It has no formal status but the author would be grateful if any errors found in it could be reported to him at brg@gladman.uk.net.

Software implementations of Rijndael by the author (in C/C++) are available at:

http://fp.gladman.plus.com/cryptography_technology/rijndael/

11. An Example of Cipher Operation

The following diagram shows the hexadecimal values in the state array as the cipher progresses for a cipher input length (*Nc*) of 4 and a cipher key length (*Nk*) of 4. The notation for the folllowing inputs is given at the start of Section 12.

```
Input = 3243f6a8885a308d313198a2e0370734 (pi * 2^124)
Key   = 2b7e151628aed2a6abf7158809cf4f3c ( e * 2^124)
```

round number	start of round	after subbytes	after shiftrows	after mixcolumns		round key value	
input	32 88 31 e0					2b 28 ab 09	
	43 5a 31 37				⊕	7e ae f7 cf	=
	f6 30 98 07					15 d2 15 4f	
	a8 8d a2 34					16 a6 88 3c	
1	19 a0 9a e9	d4 e0 b8 1e	d4 e0 b8 1e	04 e0 48 28		a0 88 23 2a	
	3d f4 c6 f8	27 bf b4 41	bf b4 41 27	66 cb f8 06	⊕	fa 54 a3 6c	=
	e3 e2 8d 48	11 98 5d 52	5d 52 11 98	81 19 d3 26		fe 2c 39 76	
	be 2b 2a 08	ae f1 e5 30	30 ae f1 e5	e5 9a 7a 4c		17 b1 39 05	
2	a4 68 6b 02	49 45 7f 77	49 45 7f 77	58 1b db 1b		f2 7a 59 73	
	9c 9f 5b 6a	de db 39 02	db 39 02 de	4d 4b e7 6b	⊕	c2 96 35 59	=
	7f 35 ea 50	d2 96 87 53	87 53 d2 96	ca 5a ca b0		95 b9 80 f6	
	f2 2b 43 49	89 f1 1a 3b	3b 89 f1 1a	f1 ac a8 e5		f2 43 7a 7f	
3	aa 61 82 68	ac ef 13 45	ac ef 13 45	75 20 53 bb		3d 47 1e 6d	
	8f dd d2 32	73 c1 b5 23	c1 b5 23 73	ec 0b c0 25	⊕	80 16 23 7a	=
	5f e3 4a 46	cf 11 d6 5a	d6 5a cf 11	09 63 cf d0		47 fe 7e 88	
	03 ef d2 9a	7b df b5 b8	b8 7b df b5	93 33 7c dc		7d 3e 44 3b	
4	48 67 4d d6	52 85 e3 f6	52 85 e3 f6	0f 60 6f 5e		ef a8 b6 db	
	6c 1d e3 5f	50 a4 11 cf	a4 11 cf 50	d6 31 c0 b3	⊕	44 52 71 0b	=
	4e 9d b1 58	2f 5e c8 6a	c8 6a 2f 5e	da 38 10 13		a5 5b 25 ad	
	ee 0d 38 e7	28 d7 07 94	94 28 d7 07	a9 bf 6b 01		41 7f 3b 00	
5	e0 c8 d9 85	e1 e8 35 97	e1 e8 35 97	25 bd b6 4c		d4 7c ca 11	
	92 63 b1 b8	4f fb c8 6c	fb c8 6c 4f	d1 11 3a 4c	⊕	d1 83 f2 f9	=
	7f 63 35 be	d2 fb 96 ae	96 ae d2 fb	a9 d1 33 c0		c6 9d b8 15	
	e8 c0 50 01	9b ba 53 7c	7c 9b ba 53	ad 68 8e b0		f8 87 bc bc	
6	f1 c1 7c 5d	a1 78 10 4c	a1 78 10 4c	4b 2c 33 37		6d 11 db ca	
	00 92 c8 b5	63 4f e8 d5	4f e8 d5 63	86 4a 9d d2	⊕	88 0b f9 00	=
	6f 4c 8b d5	a8 29 3d 03	3d 03 a8 29	8d 89 f4 18		a3 3e 86 93	
	55 ef 32 0c	fc df 23 fe	fe fc df 23	6d 80 e8 d8		7a fd 41 fd	
7	26 3d e8 fd	f7 27 9b 54	f7 27 9b 54	14 46 27 34		4e 5f 84 4e	
	0e 41 64 d2	ab 83 43 b5	83 43 b5 ab	15 16 46 2a	⊕	54 5f a6 a6	=
	2e b7 72 8b	31 a9 40 3d	40 3d 31 a9	b5 15 56 d8		f7 c9 4f dc	
	17 7d a9 25	f0 ff d3 3f	3f f0 ff d3	bf ec d7 43		0e f3 b2 4f	
8	5a 19 a3 7a	be d4 0a da	be d4 0a da	00 b1 54 fa		ea b5 31 7f	
	41 49 e0 8c	83 3b e1 64	3b e1 64 83	51 c8 76 1b	⊕	d2 8d 2b 8d	=
	42 dc 19 04	2c 86 d4 f2	d4 f2 2c 86	2f 89 6d 99		73 ba f5 29	
	b1 1f 65 0c	c8 c0 4d fe	fe c8 c0 4d	d1 ff cd ea		21 d2 60 2f	
9	ea 04 65 85	87 f2 4d 97	87 f2 4d 97	47 40 a3 4c		ac 19 28 57	
	83 45 5d 96	ec 6e 4c 90	6e 4c 90 ec	37 d4 70 9f	⊕	77 fa d1 5c	=
	5c 33 98 b0	4a c3 46 e7	46 e7 4a c3	94 e4 3a 42		66 dc 29 00	
	f0 2d ad c5	8c d8 95 a6	a6 8c d8 95	ed a5 a6 bc		f3 21 41 6e	
10	eb 59 8b 1b	e9 cb 3d af	e9 cb 3d af			d0 c9 e1 b6	
	40 2e a1 c3	09 31 32 2e	31 32 2e 09		⊕	14 ee 3f 63	=
	f2 38 13 42	89 07 7d 2c	7d 2c 89 07			f9 25 0c 0c	
	1e 84 e7 d2	72 5f 94 b5	b5 72 5f 94			a8 89 c8 a6	
output	39 02 dc 19						
	25 dc 11 6a						
	84 09 85 0b						
	1d fb 97 32						

12. Rijndael Development Test Vectors

All vectors are in hexadecimal notation with each pair of characters giving a byte value where the left and right characters of each pair provide the bit pattern for the 4 bit group containing the higher and lower numbered bits respectively using the format explained in Section 1.2. The array index for all bytes (groups of two hexadecimal digits) within these test vectors starts at zero on the left and increases from left to right.

Considered instead as bit sequences, with hexadecimal digits numbered from left to right starting from 0, hexadecimal digit *n* gives the value of bits 4*n* to 4*n*+3 in the sequence using the 4-bit notation given in Section 1.2 except that lower numbered bits are now on the left (this arises because bits in bit sequences and bits in bytes are mapped in reverse).

```
These test have been generated by Dr Brian Gladman using the
program aes_vec.cpp  <brg@gladman.uk.net> 24th January 2001.

LEGEND FOR ENCRYPT (round number r = 0 to 10, 12 or 14)
input:   cipher input
start:   state at start of round[r]
s_box:   state after s_box substitution
s_row:   state after shift row transformation
m_col:   state after mix column transformation
k_sch:   key schedule value for round[r]
output:  cipher output

LEGEND FOR DECRYPT (round number r = 0 to 10, 12 or 14)
KEY SCHEDULE FOR KEY XOR FOLLOWED BY INVERSE MIX COLUMN
iinput:  inverse cipher input
istart:  state at start of round[r]
is_box:  state after inverse s_box substitution
is_row:  state after inverse shift row transformation
ik_sch:  key schedule value for round[r]
ik_add:  state after key addition
ioutput: cipher output

LEGEND FOR DECRYPT (MOD) (round number r = 0 to 10, 12 or 14)
KEY SCHEDULE FOR INVERSE MIX COLUMN FOLLOWED BY KEY XOR
iinput:  inverse cipher input
istart:  state at start of round[r]
is_box:  state after inverse s_box substitution
is_row:  state after inverse shift row transformation
im_col:  state after inverse mix column transformation
ik_sch:  key schedule value for round[r]
ioutput: cipher output

PLAINTEXT:    3243f6a8885a308d313198a2e0370734 (pi * 2^124)
KEY:          2b7e151628aed2a6abf7158809cf4f3c ( e * 2^124)
ENCRYPT
R[ 0].input   3243f6a8885a308d313198a2e0370734
R[ 0].k_sch   2b7e151628aed2a6abf7158809cf4f3c
R[ 1].start   193de3bea0f4e22b9ac68d2ae9f84808
R[ 1].s_box   d42711aee0bf98f1b8b45de51e415230
R[ 1].s_row   d4bf5d30e0b452aeb84111f11e2798e5
R[ 1].m_col   046681e5e0cb199a48f8d37a2806264c
R[ 1].k_sch   a0fafe1788542cb123a339392a6c7605
R[ 2].start   a49c7ff2689f352b6b5bea43026a5049
R[ 2].s_box   49ded28945db96f17f39871a7702533b
R[ 2].s_row   49db873b453953897f02d2f177de961a
R[ 2].m_col   584dcaf11b4b5aacdbe7caa81b6bb0e5
R[ 2].k_sch   f2c295f27a96b9435935807a7359f67f
R[ 3].start   aa8f5f0361dde3ef82d24ad26832469a
R[ 3].s_box   ac73cf7befc111df13b5d6b545235ab8
R[ 3].s_row   acc1d6b8efb55a7b1323cfdf457311b5
R[ 3].m_col   75ec0993200b633353c0cf7cbb25d0dc
R[ 3].k_sch   3d80477d4716fe3e1e237e446d7a883b
R[ 4].start   486c4eee671d9d0d4de3b138d65f58e7
R[ 4].s_box   52502f2885a45ed7e311c807f6cf6a94
```

```
R[ 4].s_row   52a4c89485116a28e3cf2fd7f6505e07
R[ 4].m_col   0fd6daa9603138bf6fc0106b5eb31301
R[ 4].k_sch   ef44a541a8525b7fb671253bdb0bad00
R[ 5].start   e0927fe8c86363c0d9b1355085b8be01
R[ 5].s_box   e14fd29be8fbfbba35c89653976cae7c
R[ 5].s_row   e1fb967ce8c8ae9b356cd2ba974ffb53
R[ 5].m_col   25d1a9adbd11d168b63a338e4c4cc0b0
R[ 5].k_sch   d4d1c6f87c839d87caf2b8bc11f915bc
R[ 6].start   f1006f55c1924cef7cc88b325db5d50c
R[ 6].s_box   a163a8fc784f29df10e83d234cd503fe
R[ 6].s_row   a14f3dfe78e803fc10d5a8df4c632923
R[ 6].m_col   4b868d6d2c4a8980339df4e837d218d8
R[ 6].k_sch   6d88a37a110b3efddbf98641ca0093fd
R[ 7].start   260e2e173d41b77de86472a9fdd28b25
R[ 7].s_box   f7ab31f02783a9ff9b4340d354b53d3f
R[ 7].s_row   f783403f27433df09bb531ff54aba9d3
R[ 7].m_col   1415b5bf461615ec274656d7342ad843
R[ 7].k_sch   4e54f70e5f5fc9f384a64fb24ea6dc4f
R[ 8].start   5a4142b11949dc1fa3e019657a8c040c
R[ 8].s_box   be832cc8d43b86c00ae1d44dda64f2fe
R[ 8].s_row   be3bd4fed4e1f2c80a642cc0da83864d
R[ 8].m_col   00512fd1b1c889ff54766dcdfa1b99ea
R[ 8].k_sch   ead27321b58dbad2312bf5607f8d292f
R[ 9].start   ea835cf00445332d655d98ad8596b0c5
R[ 9].s_box   87ec4a8cf26ec3d84d4c46959790e7a6
R[ 9].s_row   876e46a6f24ce78c4d904ad897ecc395
R[ 9].m_col   473794ed40d4e4a5a3703aa64c9f42bc
R[ 9].k_sch   ac7766f319fadc2128d12941575c006e
R[10].start   eb40f21e592e38848ba113e71bc342d2
R[10].s_box   e9098972cb31075f3d327d94af2e2cb5
R[10].s_row   e9317db5cb322c723d2e895faf090794
R[10].k_sch   d014f9a8c9ee2589e13f0cc8b6630ca6
R[10].output  3925841d02dc09fbdc118597196a0b32
DECRYPT
R[ 0].iinput  3925841d02dc09fbdc118597196a0b32
R[ 0].ik_sch  d014f9a8c9ee2589e13f0cc8b6630ca6
R[ 1].istart  e9317db5cb322c723d2e895faf090794
R[ 1].is_row  e9098972cb31075f3d327d94af2e2cb5
R[ 1].is_box  eb40f21e592e38848ba113e71bc342d2
R[ 1].ik_sch  ac7766f319fadc2128d12941575c006e
R[ 1].ik_add  473794ed40d4e4a5a3703aa64c9f42bc
R[ 2].istart  876e46a6f24ce78c4d904ad897ecc395
R[ 2].is_row  87ec4a8cf26ec3d84d4c46959790e7a6
R[ 2].is_box  ea835cf00445332d655d98ad8596b0c5
R[ 2].ik_sch  ead27321b58dbad2312bf5607f8d292f
R[ 2].ik_add  00512fd1b1c889ff54766dcdfa1b99ea
R[ 3].istart  be3bd4fed4e1f2c80a642cc0da83864d
R[ 3].is_row  be832cc8d43b86c00ae1d44dda64f2fe
R[ 3].is_box  5a4142b11949dc1fa3e019657a8c040c
R[ 3].ik_sch  4e54f70e5f5fc9f384a64fb24ea6dc4f
R[ 3].ik_add  1415b5bf461615ec274656d7342ad843
R[ 4].istart  f783403f27433df09bb531ff54aba9d3
R[ 4].is_row  f7ab31f02783a9ff9b4340d354b53d3f
R[ 4].is_box  260e2e173d41b77de86472a9fdd28b25
R[ 4].ik_sch  6d88a37a110b3efddbf98641ca0093fd
R[ 4].ik_add  4b868d6d2c4a8980339df4e837d218d8
R[ 5].istart  a14f3dfe78e803fc10d5a8df4c632923
R[ 5].is_row  a163a8fc784f29df10e83d234cd503fe
R[ 5].is_box  f1006f55c1924cef7cc88b325db5d50c
R[ 5].ik_sch  d4d1c6f87c839d87caf2b8bc11f915bc
R[ 5].ik_add  25d1a9adbd11d168b63a338e4c4cc0b0
R[ 6].istart  e1fb967ce8c8ae9b356cd2ba974ffb53
R[ 6].is_row  e14fd29be8fbfbba35c89653976cae7c
R[ 6].is_box  e0927fe8c86363c0d9b1355085b8be01
R[ 6].ik_sch  ef44a541a8525b7fb671253bdb0bad00
R[ 6].ik_add  0fd6daa9603138bf6fc0106b5eb31301
R[ 7].istart  52a4c89485116a28e3cf2fd7f6505e07
R[ 7].is_row  52502f2885a45ed7e311c807f6cf6a94
R[ 7].is_box  486c4eee671d9d0d4de3b138d65f58e7
R[ 7].ik_sch  3d80477d4716fe3e1e237e446d7a883b
```

```
R[ 7].ik_add  75ec0993200b633353c0cf7cbb25d0dc
R[ 8].istart  acc1d6b8efb55a7b1323cfdf457311b5
R[ 8].is_row  ac73cf7befc111df13b5d6b545235ab8
R[ 8].is_box  aa8f5f0361dde3ef82d24ad26832469a
R[ 8].ik_sch  f2c295f27a96b9435935807a7359f67f
R[ 8].ik_add  584dcaf11b4b5aacdbe7caa81b6bb0e5
R[ 9].istart  49db873b453953897f02d2f177de961a
R[ 9].is_row  49ded28945db96f17f39871a7702533b
R[ 9].is_box  a49c7ff2689f352b6b5bea43026a5049
R[ 9].ik_sch  a0fafe1788542cb123a339392a6c7605
R[ 9].ik_add  046681e5e0cb199a48f8d37a2806264c
R[10].istart  d4bf5d30e0b452aeb84111f11e2798e5
R[10].is_row  d42711aee0bf98f1b8b45de51e415230
R[10].is_box  193de3bea0f4e22b9ac68d2ae9f84808
R[10].ik_sch  2b7e151628aed2a6abf7158809cf4f3c
R[10].ioutput 3243f6a8885a308d313198a2e0370734
DECRYPT (MOD)
R[ 0].iinput  3925841d02dc09fbdc118597196a0b32
R[ 0].ik_sch  d014f9a8c9ee2589e13f0cc8b6630ca6
R[ 1].istart  e9317db5cb322c723d2e895faf090794
R[ 1].is_box  eb2e13d259a1421e8bc3f2841b4038e7
R[ 1].is_row  eb40f21e592e38848ba113e71bc342d2
R[ 1].im_col  8b151cc5e1550d72fda9c248f1a03821
R[ 1].ik_sch  0c7b5a631319eafeb0398890664cfbb4
R[ 2].istart  876e46a6f24ce78c4d904ad897ecc395
R[ 2].is_box  ea4598c5045db0f065965c2d858333ad
R[ 2].is_row  ea835cf00445332d655d98ad8596b0c5
R[ 2].im_col  614646a4cb834255a9444eae0cf6f569
R[ 2].ik_sch  df7d925a1f62b09da320626ed6757324
R[ 3].istart  be3bd4fed4e1f2c80a642cc0da83864d
R[ 3].is_box  5a49190c19e004b1a38c421f7a41dc65
R[ 3].is_row  5a4142b11949dc1fa3e019657a8c040c
R[ 3].im_col  e5433678e75c1f3727f7e30c21feb899
R[ 3].ik_sch  12c07647c01f22c7bc42d2f37555114a
R[ 4].istart  f783403f27433df09bb531ff54aba9d3
R[ 4].is_box  264172253d648b17e8d22e7dfd0eb7a9
R[ 4].is_row  260e2e173d41b77de86472a9fdd28b25
R[ 4].im_col  cfb3e588aa37577c6c8858eb8574ea9a
R[ 4].ik_sch  6efcd876d2df54807c5df034c917c3b9
R[ 5].istart  a14f3dfe78e803fc10d5a8df4c632923
R[ 5].is_box  f1928b0cc1c8d5557cb56fef5d004c32
R[ 5].is_row  f1006f55c1924cef7cc88b325db5d50c
R[ 5].im_col  8f589c8054eb226d9bee760e2205c8de
R[ 5].ik_sch  6ea30afcbc238cf6ae82a4b4b54a338d
R[ 6].istart  e1fb967ce8c8ae9b356cd2ba974ffb53
R[ 6].is_box  e0633501c8b1bee8d9b87fc085926350
R[ 6].is_row  e0927fe8c86363c0d9b1355085b8be01
R[ 6].im_col  c22c8c875791ec22f16e0795ed98c93e
R[ 6].ik_sch  90884413d280860a12a128421bc89739
R[ 7].istart  52a4c89485116a28e3cf2fd7f6505e07
R[ 7].is_box  481db1e767e358ee4d5f4e0dd66c9d38
R[ 7].is_row  486c4eee671d9d0d4de3b138d65f58e7
R[ 7].im_col  d0dec54fadbd9862d30261974c1aaece
R[ 7].ik_sch  7c1f13f74208c219c021ae480969bf7b
R[ 8].istart  acc1d6b8efb55a7b1323cfdf457311b5
R[ 8].is_box  aadd4a9a61d2460382325fef688fe3d2
R[ 8].is_row  aa8f5f0361dde3ef82d24ad26832469a
R[ 8].im_col  85ae82d07b2e8267fd2bbea0be968729
R[ 8].ik_sch  cc7505eb3e17d1ee82296c51c9481133
R[ 9].istart  49db873b453953897f02d2f177de961a
R[ 9].is_box  a49fea49685b50f26b6a7f2b029c3543
R[ 9].is_row  a49c7ff2689f352b6b5bea43026a5049
R[ 9].im_col  ff88559712d686ab047fac4e5546e587
R[ 9].ik_sch  2b3708a7f262d405bc3ebdbf4b617d62
R[10].istart  d4bf5d30e0b452aeb84111f11e2798e5
R[10].is_box  19f48d08a0c648be9af8e32be93de22a
R[10].is_row  193de3bea0f4e22b9ac68d2ae9f84808
R[10].ik_sch  2b7e151628aed2a6abf7158809cf4f3c
R[10].ioutput 3243f6a8885a308d313198a2e0370734
```

```
PLAINTEXT:     3243f6a8885a308d313198a2e0370734 (pi * 2^124)
KEY:           2b7e151628aed2a6abf7158809cf4f3c ( e * 2^188)
               762e7160f38b4da5
ENCRYPT
R[ 0].input    3243f6a8885a308d313198a2e0370734
R[ 0].k_sch    2b7e151628aed2a6abf7158809cf4f3c
R[ 1].start    193de3bea0f4e22b9ac68d2ae9f84808
R[ 1].s_box    d42711aee0bf98f1b8b45de51e415230
R[ 1].s_row    d4bf5d30e0b452aeb84111f11e2798e5
R[ 1].m_col    046681e5e0cb199a48f8d37a2806264c
R[ 1].k_sch    762e7160f38b4da5179d131b3f33c1bd
R[ 2].start    7248f0851340543f5f65c0611735e7f1
R[ 2].s_box    40528c977d092075cf4dbaeff09694a1
R[ 2].s_row    4009baa17d4d9497cf968c75f05220ef
R[ 2].m_col    8026de2a2ed7a16bdd02c5bac2dbc8bc
R[ 2].k_sch    94c4d4359d0b9b09eb25ea6918aea7cc
R[ 3].start    14e20a1fb3dc3a6236272fd3da756f70
R[ 3].s_box    fa9867c06d8680aa05cc1566579da851
R[ 3].s_row    fa8615516dcca8c0059d67aa57988066
R[ 3].m_col    3a83a524fdcdb1487b27b3bafb817e2d
R[ 3].k_sch    f1c158b6cef2990b5a364d3ec73dd637
R[ 4].start    cb42fd92333f28432111fe843cbca81a
R[ 4].s_box    1f2c544fc375341afd82bb5feb65c2a2
R[ 4].s_row    1f75bba2c382c24ffd65541aeb2c345f
R[ 4].m_col    b881fab08dce0f8000d11e19d2b04e80
R[ 4].k_sch    2c183c5e34b69b92bbd517ae75278ea5
R[ 5].start    9499c6eeb9789412bb0409b7a797c025
R[ 5].s_box    22eeb42856bc22c9eaf201a95c88ba3f
R[ 5].s_row    22bc013f56f2ba28ea88b4c95cee22a9
R[ 5].m_col    a57ddda53354b3e231ef51901a541661
R[ 5].k_sch    2f11c39be82c15acc43429f2f082b260
R[ 6].start    8a6c1e3edb78a64ef5db7862ead6a401
R[ 6].s_box    7e5072b2b9bc242fe6b9bcaa87f6497c
R[ 6].s_row    7ebcbc7cb9b949b2e6f6722f875024aa
R[ 6].m_col    e3be257a42b95f5f8ba885eb6be17aa9
R[ 6].k_sch    a0e2c722d5c54987fad48a1c12f89fb0
R[ 7].start    435ce258977c16d8717c0ff77919e519
R[ 7].s_box    1a4a986a88104761a3107668b6d4d9d4
R[ 7].s_row    1a1076d48810d96aa3d49861b64a4768
R[ 7].m_col    a67481fb88b28f9ec3c2ff708683ca1c
R[ 7].k_sch    d6ccb642264e04229f1054d54ad51d52
R[ 8].start    70b837b9aefc8bbc5cd2aba5cc56d74e
R[ 8].s_box    516c9a56e4b03d654ab562064bb10e2f
R[ 8].s_row    51b0622fe4b50e564ab19a654b6c3d06
R[ 8].m_col    24a3547f4fd1b720a3e37b3f19d25780
R[ 8].k_sch    b001974ea2f908fe7435bebc527bba9e
R[ 9].start    94a2c331ed28bfded7d6c5834ba9ed1e
R[ 9].s_box    223a2ec75534081d0ef6a6ecb3d35572
R[ 9].s_row    2234a67255f655c70ed32e1db33a08ec
R[ 9].m_col    ccc9d710399a5bc941dca6d5d733b63f
R[ 9].k_sch    9ee45fd5d43142876430d5c9c6c9dd37
R[10].start    522d88c5edab194e25ec731c11fa6b08
R[10].s_box    00d8c4a65562d42f3fce8f9c822d7f30
R[10].s_row    00628f3055ce7fa63f2dc42f82d8d49c
R[10].m_col    197e378d3af59419e21df0f624d256b2
R[10].k_sch    b2fc638be087d915c9d106341de044b3
R[11].start    ab825406da724d0c2bccf6c239321201
R[11].s_box    6213206f5740e3fef14b42251223c97c
R[11].s_row    6240427c574bc96ff12320fe1213e325
R[11].m_col    3a58225cd5ee24a542298becd72fb38c
R[11].k_sch    79d0917abf194c4d0de52fc6ed62f6d3
R[12].start    4388b3266af768e84fcca42a3a4d455f
R[12].s_box    1ac46df70268459b844b49e580e36ecf
R[12].s_row    1a6849cf024b6ef784e36d9b80c445e5
R[12].k_sch    e3936061fe7324d287a3b5a838baf9e5
R[12].output   f9fb29aefc384a250340d833b87ebc00
DECRYPT
```

```
R[ 0].iinput  f9fb29aefc384a250340d833b87ebc00
R[ 0].ik_sch  e3936061fe7324d287a3b5a838baf9e5
R[ 1].istart  1a6849cf024b6ef784e36d9b80c445e5
R[ 1].is_row  1ac46df70268459b844b49e580e36ecf
R[ 1].is_box  4388b3266af768e84fcca42a3a4d455f
R[ 1].ik_sch  79d0917abf194c4d0de52fc6ed62f6d3
R[ 1].ik_add  3a58225cd5ee24a542298becd72fb38c
R[ 2].istart  6240427c574bc96ff12320fe1213e325
R[ 2].is_row  6213206f5740e3fef14b42251223c97c
R[ 2].is_box  ab825406da724d0c2bccf6c239321201
R[ 2].ik_sch  b2fc638be087d915c9d106341de044b3
R[ 2].ik_add  197e378d3af59419e21df0f624d256b2
R[ 3].istart  00628f3055ce7fa63f2dc42f82d8d49c
R[ 3].is_row  00d8c4a65562d42f3fce8f9c822d7f30
R[ 3].is_box  522d88c5edab194e25ec731c11fa6b08
R[ 3].ik_sch  9ee45fd5d43142876430d5c9c6c9dd37
R[ 3].ik_add  ccc9d710399a5bc941dca6d5d733b63f
R[ 4].istart  2234a67255f655c70ed32e1db33a08ec
R[ 4].is_row  223a2ec75534081d0ef6a6ecb3d35572
R[ 4].is_box  94a2c331ed28bfded7d6c5834ba9ed1e
R[ 4].ik_sch  b001974ea2f908fe7435bebc527bba9e
R[ 4].ik_add  24a3547f4fd1b720a3e37b3f19d25780
R[ 5].istart  51b0622fe4b50e564ab19a654b6c3d06
R[ 5].is_row  516c9a56e4b03d654ab562064bb10e2f
R[ 5].is_box  70b837b9aefc8bbc5cd2aba5cc56d74e
R[ 5].ik_sch  d6ccb642264e04229f1054d54ad51d52
R[ 5].ik_add  a67481fb88b28f9ec3c2ff708683ca1c
R[ 6].istart  1a1076d48810d96aa3d49861b64a4768
R[ 6].is_row  1a4a986a88104761a3107668b6d4d9d4
R[ 6].is_box  435ce258977c16d8717c0ff77919e519
R[ 6].ik_sch  a0e2c722d5c54987fad48a1c12f89fb0
R[ 6].ik_add  e3be257a42b95f5f8ba885eb6be17aa9
R[ 7].istart  7ebcbc7cb9b949b2e6f6722f875024aa
R[ 7].is_row  7e5072b2b9bc242fe6b9bcaa87f6497c
R[ 7].is_box  8a6c1e3edb78a64ef5db7862ead6a401
R[ 7].ik_sch  2f11c39be82c15acc43429f2f082b260
R[ 7].ik_add  a57ddda53354b3e231ef51901a541661
R[ 8].istart  22bc013f56f2ba28ea88b4c95cee22a9
R[ 8].is_row  22eeb42856bc22c9eaf201a95c88ba3f
R[ 8].is_box  9499c6eeb9789412bb0409b7a797c025
R[ 8].ik_sch  2c183c5e34b69b92bbd517ae75278ea5
R[ 8].ik_add  b881fab08dce0f8000d11e19d2b04e80
R[ 9].istart  1f75bba2c382c24ffd65541aeb2c345f
R[ 9].is_row  1f2c544fc375341afd82bb5feb65c2a2
R[ 9].is_box  cb42fd92333f28432111fe843cbca81a
R[ 9].ik_sch  f1c158b6cef2990b5a364d3ec73dd637
R[ 9].ik_add  3a83a524fdcdb1487b27b3bafb817e2d
R[10].istart  fa8615516dcca8c0059d67aa57988066
R[10].is_row  fa9867c06d8680aa05cc1566579da851
R[10].is_box  14e20a1fb3dc3a6236272fd3da756f70
R[10].ik_sch  94c4d4359d0b9b09eb25ea6918aea7cc
R[10].ik_add  8026de2a2ed7a16bdd02c5bac2dbc8bc
R[11].istart  4009baa17d4d9497cf968c75f05220ef
R[11].is_row  40528c977d092075cf4dbaeff09694a1
R[11].is_box  7248f0851340543f5f65c0611735e7f1
R[11].ik_sch  762e7160f38b4da5179d131b3f33c1bd
R[11].ik_add  046681e5e0cb199a48f8d37a2806264c
R[12].istart  d4bf5d30e0b452aeb84111f11e2798e5
R[12].is_row  d42711aee0bf98f1b8b45de51e415230
R[12].is_box  193de3bea0f4e22b9ac68d2ae9f84808
R[12].ik_sch  2b7e151628aed2a6abf7158809cf4f3c
R[12].ioutput 3243f6a8885a308d313198a2e0370734
DECRYPT (MOD)
R[ 0].iinput  f9fb29aefc384a250340d833b87ebc00
R[ 0].ik_sch  e3936061fe7324d287a3b5a838baf9e5
R[ 1].istart  1a6849cf024b6ef784e36d9b80c445e5
R[ 1].is_box  43f7a45f6acc45264f4db3e83a88682a
R[ 1].is_row  4388b3266af768e84fcca42a3a4d455f
R[ 1].im_col  bbe8f7fab65b8e7eb6cc47301b3dbef5
R[ 1].ik_sch  d9a8b586e110471147ef67ce092e5dd0
```

```
R[ 2].istart  6240427c574bc96ff12320fe1213e325
R[ 2].is_box  ab72f601dacc12062b32540c39824dc2
R[ 2].is_row  ab825406da724d0c2bccf6c239321201
R[ 2].im_col  a69dafef1b0f45b8b691d6225b664065
R[ 2].ik_sch  a6ff20df4ec13a1e89bc120dd9be94f9
R[ 3].istart  00628f3055ce7fa63f2dc42f82d8d49c
R[ 3].is_box  52ab7308edec6bc525fa884e112d191c
R[ 3].is_row  522d88c5edab194e25ec731c11fa6b08
R[ 3].im_col  5048eec405f4d3330ec50f628b82fa7b
R[ 3].ik_sch  727c48b6500286f40016217f38b8f297
R[ 4].istart  2234a67255f655c70ed32e1db33a08ec
R[ 4].is_box  9428c51eedd6ed31d7a9c3de4ba2bf83
R[ 4].is_row  94a2c331ed28bfded7d6c5834ba9ed1e
R[ 4].im_col  01a4c5a4dc1bddbed4f6482da35227c7
R[ 4].ik_sch  5014a78b38aed3e89e47d248e83e1ac1
R[ 5].istart  51b0622fe4b50e564ab19a654b6c3d06
R[ 5].is_box  70fcab4eaed2d7b95c5637bcccb88ba5
R[ 5].is_row  70b837b9aefc8bbc5cd2aba5cc56d74e
R[ 5].im_col  bcf97774fe6911e3460baa679434892a
R[ 5].ik_sch  a6e901a07679c889e5df3206227ece42
R[ 6].istart  1a1076d48810d96aa3d49861b64a4768
R[ 6].is_box  437c0f19977ce5587119e2d8795c16f7
R[ 6].is_row  435ce258977c16d8717c0ff77919e519
R[ 6].im_col  35d1c5847e18b5f6949c1be6efea50c9
R[ 6].ik_sch  4b6d79f8c7a1fc44726a69c968ba7463
R[ 7].istart  7ebcbc7cb9b949b2e6f6722f875024aa
R[ 7].is_box  8a787801dbdba43ef5d61e4eea6ca662
R[ 7].is_row  8a6c1e3edb78a64ef5db7862ead6a401
R[ 7].im_col  977794b24c22a78224dbc10a8c7eeb80
R[ 7].ik_sch  b5cb958d1ad01daace5375c3d090c929
R[ 8].istart  22bc013f56f2ba28ea88b4c95cee22a9
R[ 8].is_box  94780925b904c0eebb97c612a79994b7
R[ 8].is_row  9499c6eeb9789412bb0409b7a797c025
R[ 8].im_col  cbf6d3cbdd417ea529ed995c67e0b1e3
R[ 8].ik_sch  d48368691ec3bcead488cd468ccc85bc
R[ 9].istart  1f75bba2c382c24ffd65541aeb2c345f
R[ 9].is_box  cb3ffe1a3311a89221bcfd433c422884
R[ 9].is_row  cb42fd92333f28432111fe843cbca81a
R[ 9].im_col  8ddd6cda3588e03a3c9a779bf8830841
R[ 9].ik_sch  775b798b584448fa39071031af1b8827
R[10].istart  fa8615516dcca8c0059d67aa57988066
R[10].is_box  14dc2f70b3276f1f36750a62dae23ad3
R[10].is_row  14e20a1fb3dc3a6236272fd3da756f70
R[10].im_col  214ae26aeb510c81b40e6c3b3a12f46c
R[10].ik_sch  614358cb961c98167b98e04eca40d483
R[11].istart  4009baa17d4d9497cf968c75f05220ef
R[11].is_box  7240c0f11365e7855f35f03f17485461
R[11].is_row  7248f0851340543f5f65c0611735e7f1
R[11].im_col  393b2568516c66634e0bfc223138a994
R[11].ik_sch  ed847858b1d834cdf64aedd32f1f3171
R[12].istart  d4bf5d30e0b452aeb84111f11e2798e5
R[12].is_box  19f48d08a0c648be9af8e32be93de22a
R[12].is_row  193de3bea0f4e22b9ac68d2ae9f84808
R[12].ik_sch  2b7e151628aed2a6abf7158809cf4f3c
R[12].ioutput 3243f6a8885a308d313198a2e0370734

PLAINTEXT:    3243f6a8885a308d313198a2e0370734 (pi * 2^124)
KEY:          2b7e151628aed2a6abf7158809cf4f3c ( e * 2^252)
              762e7160f38b4da56a784d9045190cfe
ENCRYPT
R[ 0].input   3243f6a8885a308d313198a2e0370734
R[ 0].k_sch   2b7e151628aed2a6abf7158809cf4f3c
R[ 1].start   193de3bea0f4e22b9ac68d2ae9f84808
R[ 1].s_box   d42711aee0bf98f1b8b45de51e415230
R[ 1].s_row   d4bf5d30e0b452aeb84111f11e2798e5
R[ 1].m_col   046681e5e0cb199a48f8d37a2806264c
R[ 1].k_sch   762e7160f38b4da56a784d9045190cfe
R[ 2].start   7248f0851340543f22809eea6d1f2ab2
R[ 2].s_box   40528c977d09207593cd0b873cc0e537
R[ 2].s_row   40090b377dcde59793c08c753c522087
```

```
R[ 2].m_col   a77806acc45fc39a9ff2cf08297fbc23
R[ 2].k_sch   fe80ae78d62e7cde7dd969567416266a
R[ 3].start   59f8a8d41271bf44e22ba65e5d699a49
R[ 3].s_box   cb41c248c9a3081b98f124584cf9b83b
R[ 3].s_row   cba3243bc9f1b84898f9c21b4c410858
R[ 3].m_col   6cc16db771ab8b99e237d3be0b8ef52d
R[ 3].k_sch   e469866217e2cbc77d9a865738838aa9
R[ 4].start   88a8ebd56649405e9fad55e9330d7f84
R[ 4].s_box   c4c2e903333b0958db95fc1ec3d7d25f
R[ 4].s_row   c43bfc5f3395d203dbd7e958c3c2091e
R[ 4].m_col   7df2fd2e136c1c147e162df8d75931a9
R[ 4].k_sch   10fe7d7fc6d001a1bb0968f7cf1f4e9d
R[ 5].start   6d0c8051d5bc1db5c51f450f18467f34
R[ 5].s_box   3cfecdd10365a4d5a6c06e76ad5ad218
R[ 5].s_row   3c656e1803c0d2d1a65acdd5adfea476
R[ 5].m_col   a15cad7f5e2414aea18b19d78acb9a5a
R[ 5].k_sch   6ea9a93c794b62fb04d1e4ac3c526e05
R[ 6].start   cff50443276f7655a55afd7bb699f45f
R[ 6].s_box   8ae6f21acca838fc06be54214eeebfcf
R[ 6].s_row   8aa854cfccbebf1a06eef2fc4ee63821
R[ 6].m_col   77f2c0fcff6b397a2b3008f5b4f0bb4e
R[ 6].k_sch   14611694d2b1173569b87fc2a6a7315f
R[ 7].start   6393d6682dda2e4f4288773712578a11
R[ 7].s_box   fbdcf645d85731842cc4f59ac95b7e82
R[ 7].s_row   fb57f582d8c47e452c5bf684c9dc319a
R[ 7].m_col   63d3c0abc78c2f43c71f17ca5da3c282
R[ 7].k_sch   4af56ef333be0c08376fe8a40b3d86a1
R[ 8].start   2926ae58f432234bf070ff6e569e4423
R[ 8].s_box   a5f7e46abf2326b38c51169fb10b1b26
R[ 8].s_row   a5231626bf511b6a8c0be4b3b1f7269f
R[ 8].m_col   04ffc08de75a6644491e9a1dc2b1b03c
R[ 8].k_sch   3b2524bfe994338a802c4c48268b7d17
R[ 9].start   3fdae4320ece55cec932d655e43acd2b
R[ 9].s_box   75576923ab8bfc8bdd23f6fc6980bdf1
R[ 9].s_row   758bf6f1ab23bd23dd80698b6957fcfc
R[ 9].m_col   6b88011bb6128c3ed8f609982b24c2f3
R[ 9].k_sch   bdc891038e769d0bb91975afb224f30e
R[10].start   d64090183864113561ef7c37990031fd
R[10].s_box   f60960ad07438296efdf109aee63c754
R[10].s_row   f643105407dfc7adef636096ee09829a
R[10].m_col   761469fa1e5da150961fed1ec4fb4d8d
R[10].k_sch   1d288f88f4bcbc027490f04a521b8d5d
R[11].start   6b3ce672eae11d52e28f1d5496e0c0d0
R[11].s_box   7feb8e4087f8a4009873a42090e1ba70
R[11].s_row   7ff8a4708773ba4098e18e0090eba420
R[11].m_col   3913443d7af45bdb9dc87edc998a48a4
R[11].k_sch   bd67cc4f331151448a0824eb382cd7e5
R[12].start   8474887249e50a9f17c05a37a1a69f41
R[12].s_box   5f92c4403bd967dbf0babe9a3224db83
R[12].s_row   5fd9be833bbadb40f024c4db3292679a
R[12].m_col   f3ac7f9b3862ecac88343146343edb8c
R[12].k_sch   4c26568fb89aea8dcc0a1ac79e11979a
R[13].start   bf8a291480f80621443e2b81aa2f4c16
R[13].s_box   087ea5facd416ffd1bb2f10cac152947
R[13].s_row   0841f147cdb229fa1b15a5fdac7e6f0c
R[13].m_col   65c579269f3338385138437ca2ed18e6
R[13].k_sch   b6e544f785f415b30ffc315837d0e6bd
R[14].start   d3203dd11ac72d8b5ec47224953dfe5b
R[14].s_box   66b7273ea2c6d83d581c40362a27bb39
R[14].s_row   66c64039a21cbb3e5827273d2ab7d836
R[14].k_sch   7ca82c15c432c6980838dc5f96294bc5
R[14].output  1a6e6c2c662e7da6501ffb62bc9e93f3
DECRYPT
R[ 0].iinput  1a6e6c2c662e7da6501ffb62bc9e93f3
R[ 0].ik_sch  7ca82c15c432c6980838dc5f96294bc5
R[ 1].istart  66c64039a21cbb3e5827273d2ab7d836
R[ 1].is_row  66b7273ea2c6d83d581c40362a27bb39
R[ 1].is_box  d3203dd11ac72d8b5ec47224953dfe5b
R[ 1].ik_sch  b6e544f785f415b30ffc315837d0e6bd
R[ 1].ik_add  65c579269f3338385138437ca2ed18e6
```

```
R[ 2].istart  0841f147cdb229fa1b15a5fdac7e6f0c
R[ 2].is_row  087ea5facd416ffd1bb2f10cac152947
R[ 2].is_box  bf8a291480f80621443e2b81aa2f4c16
R[ 2].ik_sch  4c26568fb89aea8dcc0a1ac79e11979a
R[ 2].ik_add  f3ac7f9b3862ecac88343146343edb8c
R[ 3].istart  5fd9be833bbadb40f024c4db3292679a
R[ 3].is_row  5f92c4403bd967dbf0babe9a3224db83
R[ 3].is_box  8474887249e50a9f17c05a37a1a69f41
R[ 3].ik_sch  bd67cc4f331151448a0824eb382cd7e5
R[ 3].ik_add  3913443d7af45bdb9dc87edc998a48a4
R[ 4].istart  7ff8a4708773ba4098e18e0090eba420
R[ 4].is_row  7feb8e4087f8a4009873a42090e1ba70
R[ 4].is_box  6b3ce672eae11d52e28f1d5496e0c0d0
R[ 4].ik_sch  1d288f88f4bcbc027490f04a521b8d5d
R[ 4].ik_add  761469fa1e5da150961fed1ec4fb4d8d
R[ 5].istart  f643105407dfc7adef636096ee09829a
R[ 5].is_row  f60960ad07438296efdf109aee63c754
R[ 5].is_box  d64090183864113561ef7c37990031fd
R[ 5].ik_sch  bdc891038e769d0bb91975afb224f30e
R[ 5].ik_add  6b88011bb6128c3ed8f609982b24c2f3
R[ 6].istart  758bf6f1ab23bd23dd80698b6957fcfc
R[ 6].is_row  75576923ab8bfc8bdd23f6fc6980bdf1
R[ 6].is_box  3fdae4320ece55cec932d655e43acd2b
R[ 6].ik_sch  3b2524bfe994338a802c4c48268b7d17
R[ 6].ik_add  04ffc08de75a6644491e9a1dc2b1b03c
R[ 7].istart  a5231626bf511b6a8c0be4b3b1f7269f
R[ 7].is_row  a5f7e46abf2326b38c51169fb10b1b26
R[ 7].is_box  2926ae58f432234bf070ff6e569e4423
R[ 7].ik_sch  4af56ef333be0c08376fe8a40b3d86a1
R[ 7].ik_add  63d3c0abc78c2f43c71f17ca5da3c282
R[ 8].istart  fb57f582d8c47e452c5bf684c9dc319a
R[ 8].is_row  fbdcf645d85731842cc4f59ac95b7e82
R[ 8].is_box  6393d6682dda2e4f428873712578a11
R[ 8].ik_sch  14611694d2b1173569b87fc2a6a7315f
R[ 8].ik_add  77f2c0fcff6b397a2b3008f5b4f0bb4e
R[ 9].istart  8aa854cfccbebf1a06eef2fc4ee63821
R[ 9].is_row  8ae6f21acca838fc06be54214eeebfcf
R[ 9].is_box  cff50443276f7655a55afd7bb699f45f
R[ 9].ik_sch  6ea9a93c794b62fb04d1e4ac3c526e05
R[ 9].ik_add  a15cad7f5e2414aea18b19d78acb9a5a
R[10].istart  3c656e1803c0d2d1a65acdd5adfea476
R[10].is_row  3cfecdd10365a4d5a6c06e76ad5ad218
R[10].is_box  6d0c8051d5bc1db5c51f450f18467f34
R[10].ik_sch  10fe7d7fc6d001a1bb0968f7cf1f4e9d
R[10].ik_add  7df2fd2e136c1c147e162df8d75931a9
R[11].istart  c43bfc5f3395d203dbd7e958c3c2091e
R[11].is_row  c4c2e903333b0958db95fc1ec3d7d25f
R[11].is_box  88a8ebd56649405e9fad55e9330d7f84
R[11].ik_sch  e469866217e2cbc77d9a865738838aa9
R[11].ik_add  6cc16db771ab8b99e237d3be0b8ef52d
R[12].istart  cba3243bc9f1b84898f9c21b4c410858
R[12].is_row  cb41c248c9a3081b98f124584cf9b83b
R[12].is_box  59f8a8d41271bf44e22ba65e5d699a49
R[12].ik_sch  fe80ae78d62e7cde7dd969567416266a
R[12].ik_add  a77806acc45fc39a9ff2cf08297fbc23
R[13].istart  40090b377dcde59793c08c753c522087
R[13].is_row  40528c977d09207593cd0b873cc0e537
R[13].is_box  7248f0851340543f22809eea6d1f2ab2
R[13].ik_sch  762e7160f38b4da56a784d9045190cfe
R[13].ik_add  046681e5e0cb199a48f8d37a2806264c
R[14].istart  d4bf5d30e0b452aeb84111f11e2798e5
R[14].is_row  d42711aee0bf98f1b8b45de51e415230
R[14].is_box  193de3bea0f4e22b9ac68d2ae9f84808
R[14].ik_sch  2b7e151628aed2a6abf7158809cf4f3c
R[14].ioutput 3243f6a8885a308d313198a2e0370734
DECRYPT (MOD)
R[ 0].iinput  1a6e6c2c662e7da6501ffb62bc9e93f3
R[ 0].ik_sch  7ca82c15c432c6980838dc5f96294bc5
R[ 1].istart  66c64039a21cbb3e5827273d2ab7d836
R[ 1].is_box  d3c7725b1ac4fed15e3d3d8b95202d24
```

```
R[ 1].is_row  d3203dd11ac72d8b5ec47224953dfe5b
R[ 1].im_col  5c148cdbe9ad281737e8988b0c467d3a
R[ 1].ik_sch  54557d9c241f01ed2cfd3d76a0381236
R[ 2].istart  0841f147cdb229fa1b15a5fdac7e6f0c
R[ 2].is_box  bff82b16803e4c14442f2921aa8a0681
R[ 2].is_row  bf8a291480f80621443e2b81aa2f4c16
R[ 2].im_col  f3d1755fcfc7f7a01dd4c9d0737030ec
R[ 2].ik_sch  ac08cbdcf47d2ce0edf00d0b41e25776
R[ 3].istart  5fd9be833bbadb40f024c4db3292679a
R[ 3].is_box  84e55a4149c09f7217a6889fa1740a37
R[ 3].is_row  8474887249e50a9f17c05a37a1a69f41
R[ 3].im_col  15d50dc7f739c6319003b29b1c2e8b60
R[ 3].ik_sch  6a2da9b7704a7c7108e23c9b8cc52f40
R[ 4].istart  7ff8a4708773ba4098e18e0090eba420
R[ 4].is_box  6be11dd0ea8fc072e2e0e652963c1d54
R[ 4].is_row  6b3ce672eae11d52e28f1d5496e0c0d0
R[ 4].im_col  463156e25faa2091f6ee417d421bd8e7
R[ 4].ik_sch  b07246b65875e73c198d21ebac125a7d
R[ 5].istart  f643105407dfc7adef636096ee09829a
R[ 5].is_box  d6647cfd38ef3118610090359940113 7
R[ 5].is_row  d64090183864113561ef7c37990031fd
R[ 5].im_col  0c6e760ab14468e5a5282961ed70ef27
R[ 5].ik_sch  79e580fb1a67d5c678a840ea842713db
R[ 6].istart  758bf6f1ab23bd23dd80698b6957fcfc
R[ 6].is_box  3fced62b0e32cd32c93ae4cee4da5555
R[ 6].is_row  3fdae4320ece55cec932d655e43acd2b
R[ 6].im_col  1f9ba2155756bae0cdf3226404685d09
R[ 6].ik_sch  bab8b433e807a18a41f8c6d7b59f7b96
R[ 7].istart  a5231626bf511b6a8c0be4b3b1f7269f
R[ 7].is_box  2932ff23f4704458f09eae4b5626236e
R[ 7].is_row  2926ae58f432234bf070ff6e569e4423
R[ 7].im_col  6dbe0329bb462b784e9463a8355362ab
R[ 7].ik_sch  96e9f6ab6382553d62cf952cfc8f5331
R[ 8].istart  fb57f582d8c47e452c5bf684c9dc319a
R[ 8].is_box  63da77112d888a684257d64f12932e37
R[ 8].is_row  6393d6682dda2e4f428873712578a11
R[ 8].im_col  72d05bb79e01aaa3af1195a1ba818560
R[ 8].ik_sch  f8780f7852bf15b9a9ff675df467bd41
R[ 9].istart  8aa854cfccbebf1a06eef2fc4ee63821
R[ 9].is_box  cf6ffd5f275af443a5990455b6f5767b
R[ 9].is_row  cff50443276f7655a55afd7bb699f45f
R[ 9].im_col  025a3316f6ab7147a7170dc433be626b
R[ 9].ik_sch  3e3f5d0ef56ba396014dc0119e40c61d
R[10].istart  3c656e1803c0d2d1a65acdd5adfea476
R[10].is_box  6dbc4534d51f7f51c54680b5180c1d0f
R[10].is_row  6d0c8051d5bc1db5c51f450f18467f34
R[10].im_col  7127e7019952c8c220979bbc9e5ad302
R[10].ik_sch  b51c1b5eaac71ac1fb4072e45d98da1c
R[11].istart  c43bfc5f3395d203dbd7e958c3c2091e
R[11].is_box  8849558466ad7fd59f0deb5e33a840e9
R[11].is_row  88a8ebd56649405e9fad55e9330d7f84
R[11].im_col  b12fee6e02a546d06cdfa19cd34c0e54
R[11].ik_sch  7a8cca55cb54fe98f42663879f0d060c
R[12].istart  cba3243bc9f1b84898f9c21b4c410858
R[12].is_box  5971a649122b9ad4e269a8445df8bf5e
R[12].is_row  59f8a8d41271bf44e22ba65e5d699a49
R[12].im_col  8687d60a6216e408c247e4509a8a887f
R[12].ik_sch  c68edd3d1fdb019f51876825a6d8a8f8
R[13].istart  40090b377dcde59793c08c753c522087
R[13].is_box  72409eb213802a85221ff03f6d4854ea
R[13].is_row  7248f0851340543f22809eea6d1f2ab2
R[13].im_col  393b2568516c666387338cee750cfd6e
R[13].ik_sch  ed847858b1d834cd3f729d1f6b2b658b
R[14].istart  d4bf5d30e0b452aeb84111f11e2798e5
R[14].is_box  19f48d08a0c648be9af8e32be93de22a
R[14].is_row  193de3bea0f4e22b9ac68d2ae9f84808
R[14].ik_sch  2b7e151628aed2a6abf7158809cf4f3c
R[14].ioutput 3243f6a8885a308d313198a2e0370734
```