Lecture Note 4

The Advanced Encryption Standard (AES)

Sourav Mukhopadhyay

Cryptography and Network Security - MA61027

Exhaustive Search on DES

- Diffie-Hellman (1977):
  - Special purpose hardware;
  - Estimated time 12-hours at a cost of USD 20M.
- Wiener (1993):
  - Special purpose hardware;
  - Estimated time 3.5 hours at a cost of USD 1 M.

Exhaustive Search on DES

- Goldberg-Wagner (1996):
  - FPGA based hardware;
  - Estimated time one year at a cost of USD 45,000.
- Electronic Frontier Foundation (EFF) (1998): DES Cracker
  - Special purpose hardware;
  - Estimated time 3 days at a cost of USD 200,000.

- Only perfectly secure algorithm we have is not very practical for use in the field.
- Necessary to turn back to some of the other algorithms and see what can be done to improve security.
- Because DES was standard it was desired to keep it in the picture and as a result the algorithm **Triple DES** was realised.
- Triple DES is simply applying the DES algorithm three times using two different keys (see figure 9):

$$C = E_{K_1}[D_{K_2}[E_{K_1}[P]]] \tag{1}$$

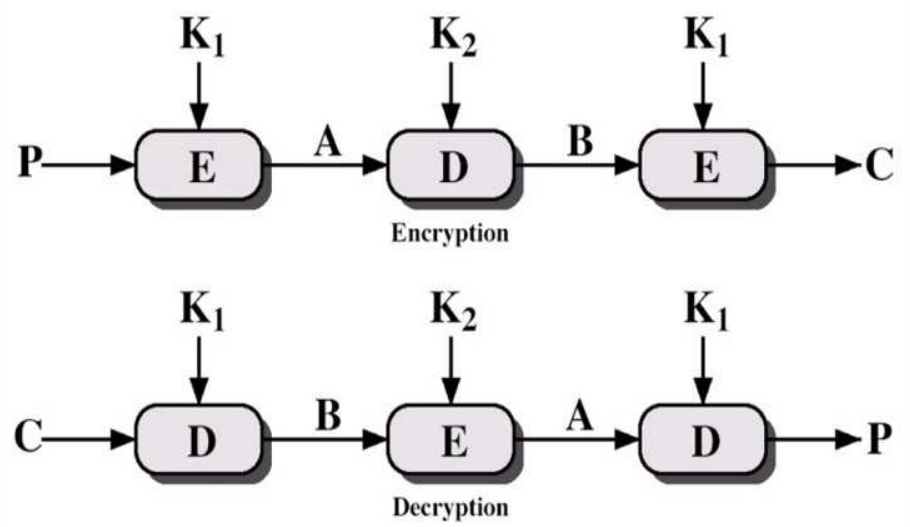


Figure 9: Triple DES.

- The order of encryptions and decryptions determines name EDE or Encrypt-Decrypt-Encrypt.
- Decryption is performed on the second iteration is simply for backward compatibility and offers no extra security to the algorithm.
- This can be seen from:

$$C = E_{K_1}[D_{K_1}[E_{K_1}[P]]] = E_{K_1}[P]. \tag{2}$$

- All of the cryptographic algorithms we have looked at so far have some problems.
- The earlier ciphers can be broken with ease on modern computation systems.
- The DES algorithm was broken in 1998 using a system that costs about $250,000.
- Triple DES turned out to be too slow for efficiency as the DES algorithm was developed for mid-1970's hardware and does not produce efficient software code. Triple DES on the other hand, has three times as many rounds as DES and is correspondingly slower.

- 64 bit block size of triple DES and DES is not very efficient and is questionable when it comes to security.
- What was required was a brand new encryption algorithm. One that would be resistant to all known attacks.
- The National Institute of Standards and Technology (NIST) wanted to help in the creation of a new standard.

- Instead of designing or helping to design a cipher, what NIST did instead, was to set up a contest in which anyone in the world could take part.
- The contest was announced on the 2nd of January 1997 and the idea was to develop a new encryption algorithm that would be used for protecting sensitive, non-classified, U.S. government information.
- The ciphers had to meet a lot of requirements and the whole design had to be fully documented (unlike the DES cipher).

- Once the candidate algorithms had been submitted, several years of scrutinisation in the form of cryptographic conferences took place.
- In the first round of the competition 15 algorithms were accepted and this was narrowed to 5 in the second round.
- The algorithms were tested for efficiency and security both by some of the worlds best publicly renowned cryptographers and NIST itself.
- After all these investigation NIST finally choose an algorithm known as **Rijndael**.

- Rijndael was named after the two Belgian cryptographers who developed and submitted it - Dr. Joan Daemen and Dr. Vincent Rijmen.
- On the 26 November 2001, AES (which is a standardised version of Rijndael) became a FIPS standard (FIPS 197).

- Like DES, AES is a symmetric block cipher. However, AES is quite different from DES in a number of ways.
- The algorithm Rijndael allows for a variety of block and key sizes and not just the 64 and 56 bits of DES' block and key size.
- The block and key can in fact be chosen independently from $128, 160, 192, 224, 256$ bits and need not be the same.
- However, the AES standard states that the algorithm can only accept a block size of 128 bits and a choice of three keys - $128, 192, 256$ bits.

- Depending on which version is used, the name of the standard is modified to AES-128, AES-192 or AES-256 respectively.
- As well as these differences AES differs from DES in that it is not a feistel structure.
- Recall that in a feistel structure, half of the data block is used to modify the other half of the data block and then the halves are swapped. In this case the entire data block is processed in parallel during each round using substitutions and permutations.

- A number of AES parameters depend on the key length. For example, if the key size used is 128 then the number of rounds is 10 whereas it is 12 and 14 for 192 and 256 bits respectively.
- At present the most common key size likely to be used is the 128 bit key. This description of the AES algorithm therefore describes this particular implementation.

- Rijndael was designed to have the following characteristics:
  - Resistance against all known attacks.
  - Speed and code compactness on a wide range of platforms.
  - Design Simplicity.
- The input is a single 128 bit block both for decryption and encryption and is known as the **in** matrix (figure 1).

- This block is copied into a **state** array which is modified at each stage of the algorithm and then copied to an output matrix (see figure 2).
- Both the plaintext and key are depicted as a 128 bit square matrix of bytes.
- This key is then expanded into an array of key schedule words (the **w** matrix).
- Ordering of bytes within the **in** matrix is by column. The same applies to the **w** matrix.

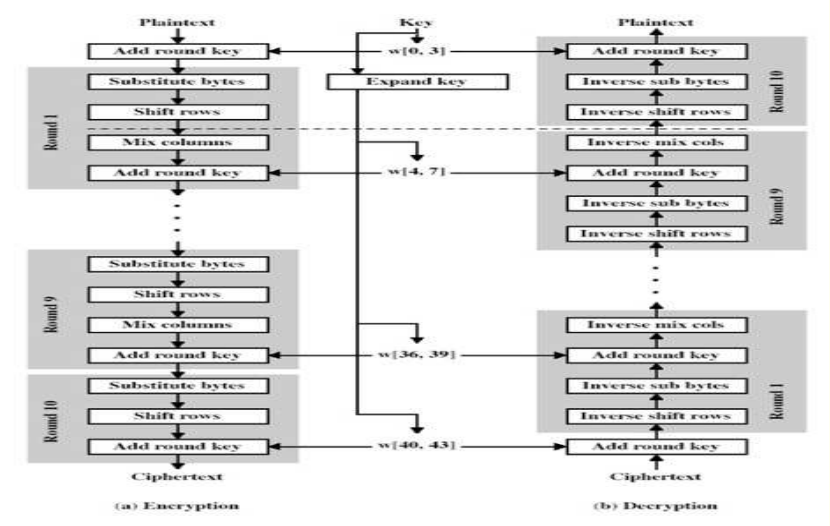


Figure 1: Overall structure of the AES algorithm.

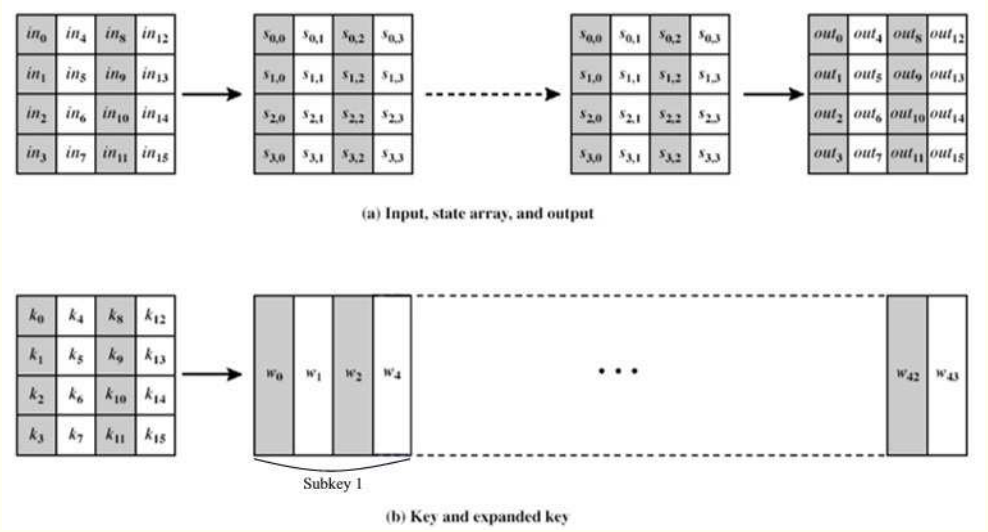


Figure 2: Data structures in the AES algorithm.

High-level description of r-round AES

1. Given a plaintext X, initialize **state** to be X and perform an operation **Add round key**, which x-ors the round key with **state**.
2. For each of the first $r-1$ rounds, perform a substitution operation called **SubBytes** on **state** using an S-box; perform a permutation **ShiftRows** on **state**; perform an operation **MixColumns** on **state**; and perform **AddRoundKey**.
3. Perform **SubBytes**; perform **ShiftRows**; and perform **AddRoundKey**.
4. Define the ciphertext Y to be **state**.

- From this high-level description, we can see that structure of the AES is very similar in many respect to the SPN discussed earlier.
- In every round of both these cryptosystems, we have subkey mixing, a substitution step and a permutation step.
- AES is "larger," and it also includes an additional linear transformation (**MixColumns**) in each round.

- All operations in AES are byte-oriented operations, and all variables used are considered to be formed from an appropriate number of bytes.
- The plaintext X consists of 16 bytes.
- **state** is represented as a four by four array of bytes.
- We will often use hexadecimal notation to represent the contents of a byte. Each byte therefore consists of two hexadecimal digits.

Inner Workings of a Round

- The algorithm begins with an **Add round key** stage followed by 9 rounds of four stages and a tenth round of three stages.
- This applies for both encryption and decryption with the exception that each stage of a round the decryption algorithm is the inverse of its counterpart in the encryption algorithm.
- The four stages are as follows:
  1. Substitute bytes
  2. Shift rows

3. Mix Columns
4. Add Round Key

- The tenth round simply leaves out the **Mix Columns** stage. The first nine rounds of the decryption algorithm consist of the following:

1. Inverse Shift rows
2. Inverse Substitute bytes
3. Inverse Add Round Key
4. Inverse Mix Columns

- Again, the tenth round simply leaves out the **Inverse Mix Columns** stage. Each of these stages will now be considered in more detail.

Substitute Bytes

- This stage (known as SubBytes) is simply a table lookup using a 16×16 matrix of byte values called an **s-box**.
- This matrix consists of all the possible combinations of an 8 bit sequence ($2^8 = 16 \times 16 = 256$).
- However, the s-box is not just a random permutation of these values and there is a well defined method for creating the s-box tables.
- The designers of Rijndael showed how this was done unlike the s-boxes in DES for which no rationale was given.

- Our concern will be how **state** is effected in each round.
- For this particular round each byte is mapped into a new byte in the following way: the leftmost nibble of the byte is used to specify a particular row of the s-box and the rightmost nibble specifies a column.
- For example, the byte $\{95\}$ (curly brackets represent hex values in FIPS PUB 197) selects row 9 column 5 which turns out to contain the value $\{2A\}$.
- This is then used to update the **state** matrix. Figure 3 depicts this idea.

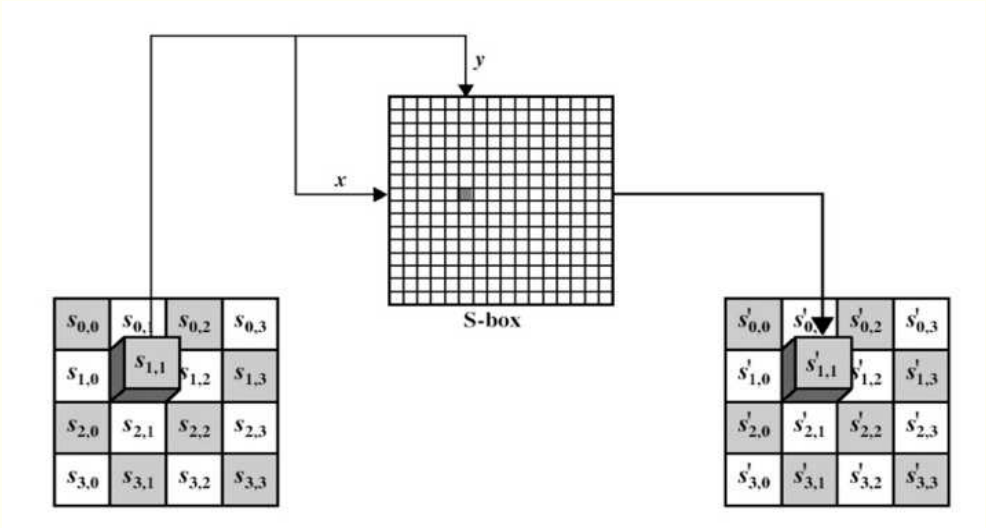
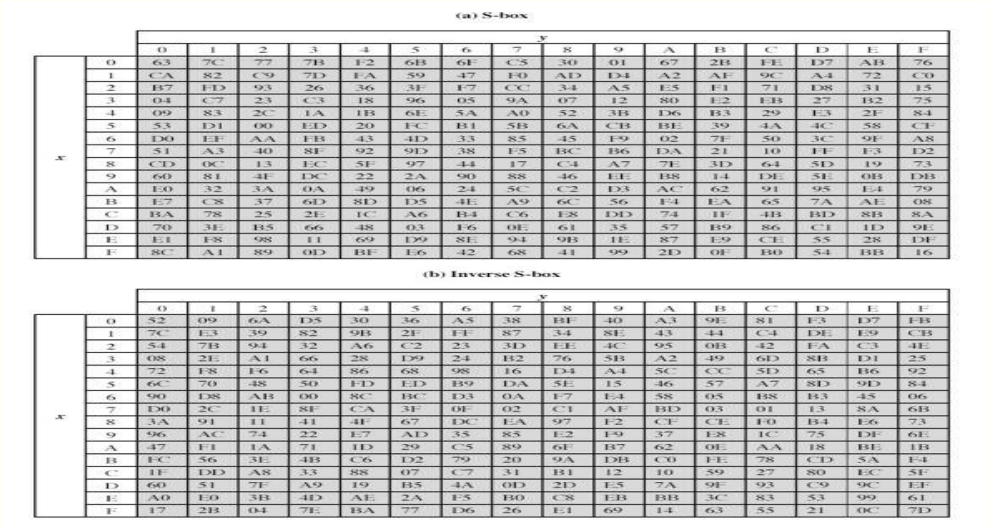


Figure 3: Substitute Bytes Stage of the AES algorithm.

- The Inverse substitute byte transformation (known as InvSubBytes) makes use of an inverse s-box.
- In this case what is desired is to select the value $\{2A\}$ and get the value $\{95\}$.
- Table 4 shows the two s-boxes and it can be verified that this is in fact the case.
- The s-box is designed to be resistant to known cryptanalytic attacks.

- Specifically, the Rijndael developers sought a design that has a low correlation between input bits and output bits, and the property that the output cannot be described as a simple mathematical function of the input.
- In addition, the s-box has no fixed points (s-box$(a) = a$) and no opposite fixed points (s-box$(a) = \bar{a}$) where $\bar{a}$ is the bitwise compliment of a.
- The s-box must be invertible if decryption is to be possible (Is-box[s-box(a)]$= a$) however it should not be its self inverse i.e. s-box$(a) \neq$ Is-box(a)



(a) S-box

x \ y	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
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

(b) Inverse S-box

x \ 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
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

Figure 4: AES s-boxes both forward and inverse.

Algebraic formulation of AES S-box

- In contrast to the S-boxes in DES, which are apparently "random" substitution, the AES S-box can be defined algebraically.
- AES S-box involves operations in the finite field:

$$F_{2^8} = Z_2[x]/(x^8 + x^4 + x^3 + x + 1).$$

- Let FieldInv denote the multiplicative inverse of a field element.
- Let BinaryToField convert a byte to a field element; and FieldToBinary perform the inverse conversion.

- The field element

$$\sum_{i=0}^{7} a_i x^i$$

corresponds to the byte:

$$a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0,$$

where $a_i \in Z_2 = \{0, 1\}$ for $0 \leq i \leq 7$.

- We now discuss **SubBytes** algorithm where eight input bits $a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0$ are replaced by the eight output bits $b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0$.

Algorithm SubBytes$(a_7a_6a_5a_4a_3a_2a_1a_0)$
external: FieldInv, BinaryToField, FieldToBinary
$z \leftarrow$ BinaryToField$(a_7a_6a_5a_4a_3a_2a_1a_0)$
if $z \neq 0$ **then** $z \leftarrow$ FieldInv(z)
$(a_7a_6a_5a_4a_3a_2a_1a_0) \leftarrow$ FieldToBinary(z)
$(c_7c_6c_5c_4c_3c_2c_1c_0) \leftarrow (01100011)$
comment: all subscripts are to be reduced modulo 8
for $i \leftarrow 0$ **to** 7
do $b_i \leftarrow (a_i + a_{i+4} + a_{i+5} + a_{i+6} + a_{i+7} + c_i) \bmod 2$
return $(b_7b_6b_5b_4b_3b_2b_1b_0)$

Example

- Suppose we begin with (hexadecimal) {53}, which is 01010011 in binary.
- The corresponding field element is:

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

- The multiplicative inverse (in F_{2^8}) can be shown to be

$$x^7 + x^6 + x^3 + x.$$

Therefore, in binary notation, we have

$$(a_7 a_6 a_5 a_4 a_3 a_2 a_1 a_0) = (11001010).$$

- Next, we compute

$$\begin{aligned} b_0 &= a_0 + a_4 + a_5 + a_6 + a_7 + c_0 \bmod 2 \\ &= 0 + 0 + 0 + 1 + 1 + 1 \bmod 2 \\ &= 1 \\ b_1 &= a_1 + a_5 + a_6 + a_7 + a_0 + c_1 \bmod 2 \\ &= 1 + 0 + 1 + 1 + 0 + 1 \bmod 2 \\ &= 0 \\ &\vdots \end{aligned}$$

- The result is: $(b_7 b_6 b_5 b_4 b_3 b_2 b_1 b_0) = (11101101)$, which is $\{ED\}$ in hexadecimal notation.

- We have the following affine transformation:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ a_4 \\ a_5 \\ a_6 \\ a_7 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

- We perform addition modulo 2 (x-ors) in the above matrix multiplication.

Algorithm InvSubBytes($b_7b_6b_5b_4b_3b_2b_1b_0$)

external: FieldInv, BinaryToField, FieldToBinary

$(d_7d_6d_5d_4d_3d_2d_1d_0) \leftarrow (00000101)$

comment: all subscripts are to be reduced modulo 8

for $i \leftarrow 0$ **to** 7

do $b'_i \leftarrow (b_{i+2} + b_{i+5} + b_{i+7} + d_i) \bmod 2$

$z \leftarrow \mathsf{BinaryToField}(b'_7b'_6b'_5b'_4b'_3b'_2b'_1b'_0)$

if $z \neq 0$ **then** $z \leftarrow \mathsf{FieldInv}(z)$

$(a_7a_6a_5a_4a_3a_2a_1a_0) \leftarrow \mathsf{FieldToBinary}(z)$

return $(a_7a_6a_5a_4a_3a_2a_1a_0)$

- We have the following affine transformation:

$$\begin{pmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

- We perform addition modulo 2 (x-ors) in the above matrix multiplication.

Shift Row Transformation

- This stage (known as ShiftRows) is shown in figure 5.
- Simple permutation an nothing more.
- It works as follow:
  - The first row of **state** is *not* altered.
  - The second row is shifted 1 bytes to the left in a circular manner.
  - The third row is shifted 2 bytes to the left in a circular manner.
  - The fourth row is shifted 3 bytes to the left in a circular manner.

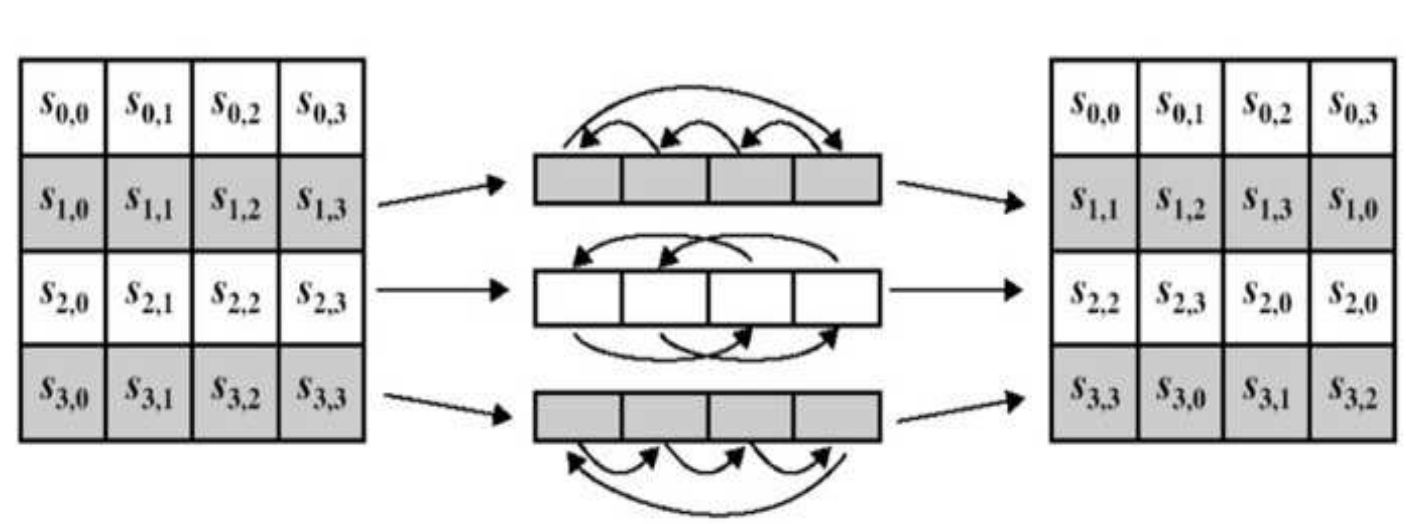


Figure 5: ShiftRows stage.

- The Inverse Shift Rows transformation (known as InvShiftRows) performs these circular shifts in the opposite direction for each of the last three rows (the first row was unaltered to begin with).
- This operation may not appear to do much but if you think about how the bytes are ordered within **state** then it can be seen to have far more of an impact.

- Remember that **state** is treated as an array of four byte columns, i.e. the first column actually represents bytes $1, 2, 3$ and 4.
- A one byte shift is therefore a linear distance of four bytes.
- The transformation also ensures that the four bytes of one column are spread out to four different columns.

Mix Column Transformation

- This stage (known as MixColumn) is basically a substitution but it makes use of arithmetic of GF(2^8).
- Each column is operated on individually. Each byte of a column is mapped into a new value that is a function of all four bytes in the column.
- The transformation can be determined by the following matrix multiplication on **state** (see figure 6):

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

- Each element of the product matrix is the sum of products of elements of one row and one column.
- In this case the individual additions and multiplications are performed in GF(2^8).
- The MixColumns transformation of a single column j ($0 \leq j \leq 3$) of **state** can be expressed as:

$$\begin{aligned} s'_{0,j} &= (2 \bullet s_{0,j}) \oplus (3 \bullet s_{1,j}) \oplus s_{2,j} \oplus s_{3,j} \\ s'_{1,j} &= s_{0,j} \oplus (2 \bullet s_{1,j}) \oplus (3 \bullet s_{2,j}) \oplus s_{3,j} \\ s'_{2,j} &= s_{0,j} \oplus s_{1,j} \oplus (2 \bullet s_{2,j}) \oplus (3 \bullet s_{3,j}) \\ s'_{3,j} &= (3 \bullet s_{0,j}) \oplus s_{1,j} \oplus s_{2,j} \oplus (2 \bullet s_{3,j}) \end{aligned} \quad (1)$$

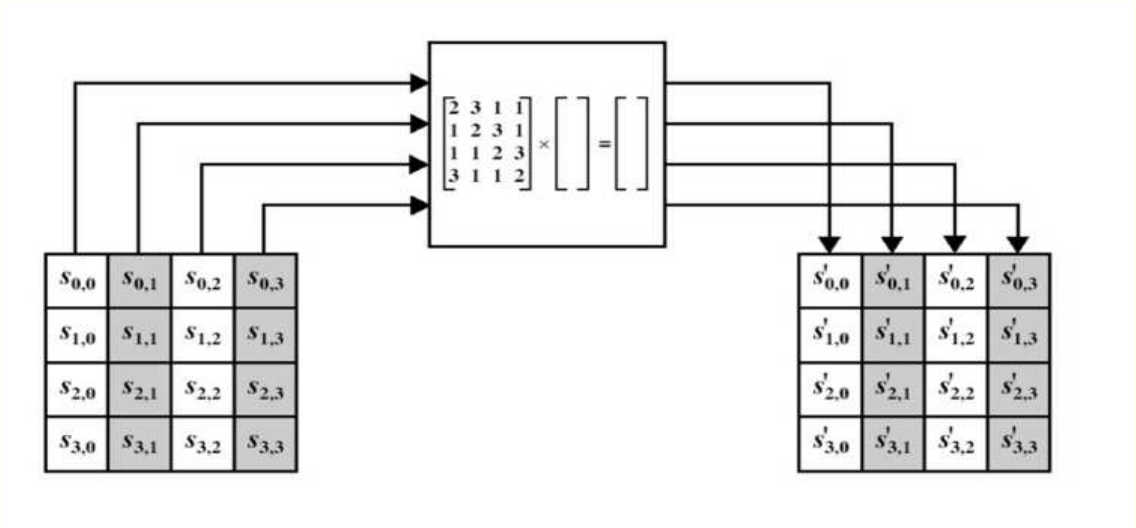


Figure 6: MixColumns stage.

Algorithm MixColumn(c)
external: FieldMult, BinaryToField, FieldToBinary
for $i \leftarrow 0$ **to** 3
  do $t_i \leftarrow \mathsf{BinaryToField}(s_{i,c})$
$u_0 \leftarrow \mathsf{FieldMult}(x, t_0) \oplus \mathsf{FieldMult}(x+1, t_1) \oplus t_2 \oplus t_3$
$u_1 \leftarrow \mathsf{FieldMult}(x, t_1) \oplus \mathsf{FieldMult}(x+1, t_2) \oplus t_3 \oplus t_0$
$u_2 \leftarrow \mathsf{FieldMult}(x, t_2) \oplus \mathsf{FieldMult}(x+1, t_3) \oplus t_0 \oplus t_1$
$u_3 \leftarrow \mathsf{FieldMult}(x, t_3) \oplus \mathsf{FieldMult}(x+1, t_0) \oplus t_1 \oplus t_2$
for $i \leftarrow 0$ **to** 3
  do $s_{i,c} \leftarrow \mathsf{FieldToBinary}(u_i)$

- As and example, lets take the first column of a matrix to be $s_{0,0} = \{87\}, s_{1,0} = \{6E\}, s_{2,0} = \{46\}, s_{3,0} = \{A6\}$.
- This would mean that $s_{0,0} = \{87\}$ gets mapped to the value $s'_{0,0} = \{47\}$ which can be seen by working out the first line of equation (1) with $j = 0$.
- Therefore we have:

$$(02 \bullet 87) \oplus (03 \bullet 6E) \oplus 46 \oplus A6 = 47$$

- So to show this is the case we can represent each Hex number by a polynomial:

$$\{02\} = x; \text{and} \{87\} = x^7 + x^2 + x + 1$$

- Multiply these two together and we get:

$$x \bullet (x^7 + x^2 + x + 1) = x^8 + x^3 + x^2 + x$$

- The degree of this result is greater than 7 so we have to reduce it modulo an irreducible polynomial $m(x)$. The designers of AES chose $m(x) = x^8 + x^4 + x^3 + x + 1$. So it can be seen that

$$(x^8 + x^3 + x^2 + x) \bmod (x^8 + x^4 + x^3 + x + 1) = x^4 + x^2 + 1$$

- This is equal to 0001 0101 in binary. This method can be used to work out the other terms. The result is therefore:

$$
\begin{array}{lcll}
02 \bullet 87 & = & 0001\ 0101 & \\
03 \bullet 6E & = & 1011\ 0010 & \\
46 & = & 0100\ 0110 & \\
A6 & = & 1010\ 0110 & \\
\hline
 & \oplus & 0100\ 0111 & = \{47\}
\end{array}
$$

- There is infact an easier way to do multiplication modulo $m(x)$.
- If we were multiplying by $\{02\}$ then all we have to do is a 1-bit left shift followed by a conditional bitwise XOR with (00011011) if the leftmost bit of the original value (prior to the shift) was 1.
- Multiplication by other numbers can be seen to be repeated application of this method.
- Stallings goes into more detail on why this works but we will not be too concerned with it here.

- What is important to note however is that a multiplication operation has been reduced to a shift and an XOR operation.
- This is one of the reasons why AES is a very efficient algorithm to implement.
- The InvMixColumns is defined by the following matrix multiplication:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

- This first matrix of equation can be shown to be the inverse of the first matrix in equation .
- If we label these $\mathbf{A}$ and $\mathbf{A}^{-1}$ respectively and we label state before the mix columns operation as $\mathbf{S}$ and after as $\mathbf{S}'$, we can see that:

$$\mathbf{AS} = \mathbf{S}'$$

- Therefore

$$\begin{aligned} & \mathbf{A}^{-1}\mathbf{S}' \\ = \ & \mathbf{A}^{-1}\mathbf{AS} = \mathbf{S} \end{aligned}$$

Add Round Key Transformation

- In this stage (known as AddRoundKey) the 128 bits of **state** are bitwise XORed with the 128 bits of the round key.
- The operation is viewed as a columnwise operation between the 4 bytes of a **state** column and one word of the round key.
- This transformation is as simple as possible which helps in efficiency but it also effects every bit of **state**.

- The AES key expansion algorithm takes as input a 4-word key and produces a linear array of 44 words. Each round uses 4 of these words as shown in figure 2.
- Each word contains 32 bytes which means each subkey is 128 bits long. Figure 7 show pseudocode for generating the expanded key from the actual key.

Add Round Key Transformation

```
KeyExpansion (byte key[16], word w[44])
{
  word temp
  for (i = 0; i < 4; i++)    w[i] = (key[4 * i], key[4 * i + 1], key[4 * i + 2], key[4 * i + 3]);
  for (i = 4; i < 44; i++)
  {
    temp = w[i − 1];
    if (i mod 4 = 0)    temp = SubWord (RotWord(temp)) ⊕ Rcon[i/4];
    w[i] = w[1 − 4] ⊕ temp;
  }
}
```

Figure 7: Key expansion pseudocode.

- The key is copied into the first four words of the expanded key.
- The remainder of the expanded key is filled in four words at a time.
- Each added word $\mathbf{w}[i]$ depends on the immediately preceding word, $\mathbf{w}[i-1]$, and the word four positions back $\mathbf{w}[i-4]$.
- In three out of four cases, a simple XOR is used.
- For a word whose position in the $\mathbf{w}$ array is a multiple of 4, a more complex function is used.

Algorithm KeyExpansion(key)

external: RotWord, SubWord

$Rcon[1] \leftarrow 01000000$

$Rcon[2] \leftarrow 02000000$

$Rcon[3] \leftarrow 04000000$

$Rcon[4] \leftarrow 08000000$

$Rcon[5] \leftarrow 10000000$

$Rcon[6] \leftarrow 20000000$

$Rcon[7] \leftarrow 40000000;$

$Rcon[8] \leftarrow 80000000$

$Rcon[9] \leftarrow 1B000000;\ Rcon[10] \leftarrow 36000000$

for $i \leftarrow 0$ **to** 3
  do $w[i] \leftarrow (key[4i], key[4i+1], key[4i+2], key[4i+3])$
for $i \leftarrow 4$ **to** 43 **do**
  $temp \leftarrow w[i-1]$
  if $i \equiv 0 \pmod 4$
    then $temp \leftarrow SubWord(RotWord(temp)) \oplus Rcon[i/4]$
  $w[i] \leftarrow w[i-4] \oplus temp$
end do
return $(w[0], w[1], \cdots, w[43])$

- Figure 8 illustrates the generation of the first eight words of the expanded key using the symbol g to represent that complex function.
- The function g consists of the following subfunctions:
  1. **RotWord** performs a one-byte circular left shift on a word. This means that an input word $[b_0, b_1, b_2, b_3]$ is transformed into $[b_1, b_2, b_3, b_0]$.
  2. **SubWord** performs a byte substitution on each byte of its input word, using the s-box described earlier.
  3. The result of steps 1 and 2 is XORed with round constant, Rcon$[j]$.

- The round constant is a word in which the three rightmost bytes are always 0.
- Thus the effect of an XOR of a word with Rcon is to only perform an XOR on the leftmost byte of the word.
- The round constant is different for each round and is defined as Rcon$[j]$ = (RC$[J]$, 0,0,0), with RC[1]= 1, RC$[j]$= 2• RC$[j-1]$ and with multiplication defined over the field GF(2^8).
- The key expansion was designed to be resistant to know cryptanalytic attacks.

- The inclusion of a round-dependent round constant eliminates the symmetry, or similarity, between the way in which round keys are generated in different rounds.
- Figure 9 give a summary of each of the rounds.
- The ShiftRows column is depicted here as a linear shift which gives a better idea how this section helps in the encryption.

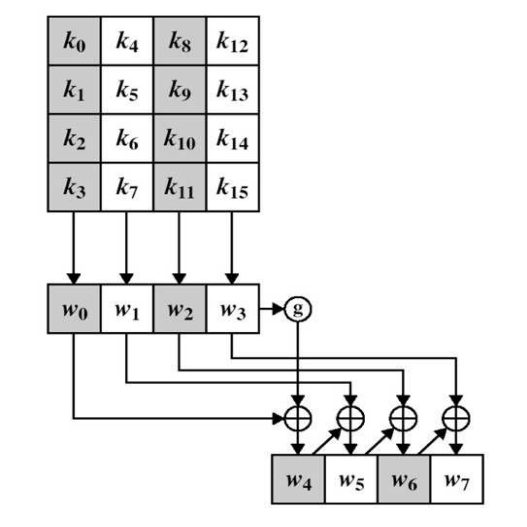


Figure 8: AES key expansion.

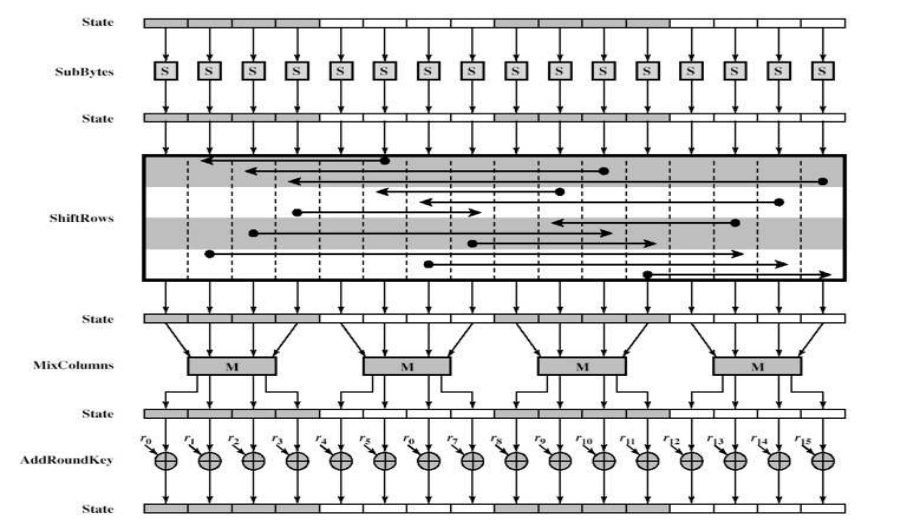


Figure 9: AES encryption round.

Analysis of AES

- Obviously, the AES is secure against all known attacks. Various aspects of its design incorporate specific features that help provide security against specific attacks.
- For example, the use of the finite field inversion operation in the construction of the S-box yields linear approximation and difference distribution tables in which the entries are close to uniform. This provides security against differential and linear attacks.

- As well, the linear transformation, MixColumns, makes it impossible to find differential and linear attacks that involve "few" active S-boxes (the designers refer to this feature as the *wide trail strategy*).
- There are apparently no known attacks on AES that are faster than exhaustive search.
- The "best" attacks on AES apply to variants of the cipher in which the number of rounds is reduced, and are not effective for 10-round AES.

Equivalent Inverse Cipher

- As can be seen from figure 1 the decryption ciphers is not identical to the encryption ciphers.
- However the form of the key schedules is the same for both.
- This has the disadvantage that two separate software or firmware modules are needed for applications that require both encryption and decryption.
- As well as that, decryption is slightly less efficient to implement.

- However, encryption was deemed more important than decryption for two reasons:
  1. For the CFB and OFB cipher mode (which we have seen before but will study in more detail next) only encryption is used.
  2. As with any block cipher, AES can be used to construct a message authentication code (to be described later), and for this only encryption is used.
- However, if desired it is possible to create an **equivalent inverse cipher**.

- This means that decryption has the same structure as the encryption algorithms.
- However, to achieve this, a change of key schedule is needed.
- We will not be concerned with this alternate form but you should be aware that is exists.