50.042 FCS Summer 2024
Lectures 7 & 8 – Block Ciphers

Felix LOH
Singapore University of Technology and Design

SINGAPORE UNIVERSITY OF TECHNOLOGY AND DESIGN

With selected materials adapted from: *Understanding Cryptography: A Textbook for Students and Practitioners, by C. Paar and J. Pelzl*

Recap of basic ciphers

- Simple ciphers:
  - E.g. shift ciphers like Caesar's cipher
  - Problems: Small keyspace; vulnerable to frequency analysis; ciphers are linear
- Stream ciphers:
  - E.g. one-time pad (OTP)
  - Problems: Generation of the key stream can be impractical; low throughput in general

Block ciphers: motivation

- Simple ciphers have a small keyspace and are vulnerable to frequency analysis
- Stream ciphers generally suffer from low throughput and have issues with the key stream generation
- So we need more efficient ciphers/cryptoalgorithms that can encrypt large data sets
  - Parallelize part of the operations of the cipher
  - Leverage the large word width of modern CPUs

Primitive operations for block ciphers

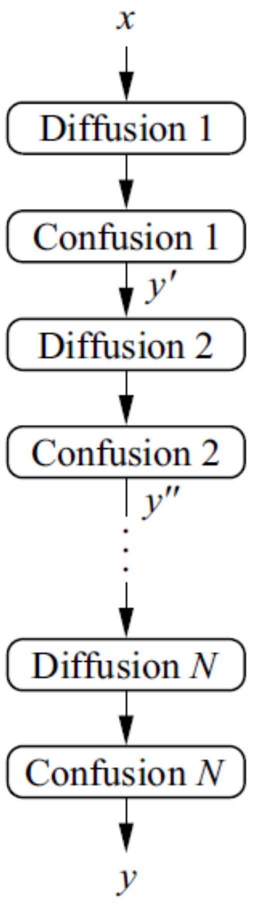
- There are two primitive operations with which strong encryption algorithms can be built
- **Confusion**: An encryption operation where the relationship between the key and ciphertext is obscured
  - Can be achieved with the substitution operation
- **Diffusion**: An encryption operation where the influence of one plaintext symbol is spread over many ciphertext symbols with the goal of hiding the statistical properties of the plaintext
  - Can be achieved with bit permutation operations
  - Modern block ciphers possess excellent diffusion properties – this means that changing one bit of the plaintext results, on average, in the change of half of the bits of the ciphertext

→ spread the influence of one plaintext bit over many ciphertext bits

e.g. $x_1 = 0010\ 1011$ → block cipher → $y_1 = 1011\ 1001$
$x_2 = 0000\ 1011$ $\quad y_2 = 0110\ 1100$ } many difference

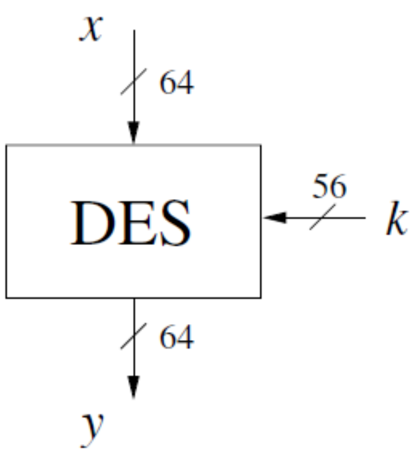
Primitive operations for block ciphers

- **Both** confusion and diffusion operations are utilized in modern block ciphers
- Several iterations of one diffusion operation followed by a confusion operation
- This is also called a product cipher
  - In other words, all modern block ciphers are product ciphers



Overview of the DES algorithm

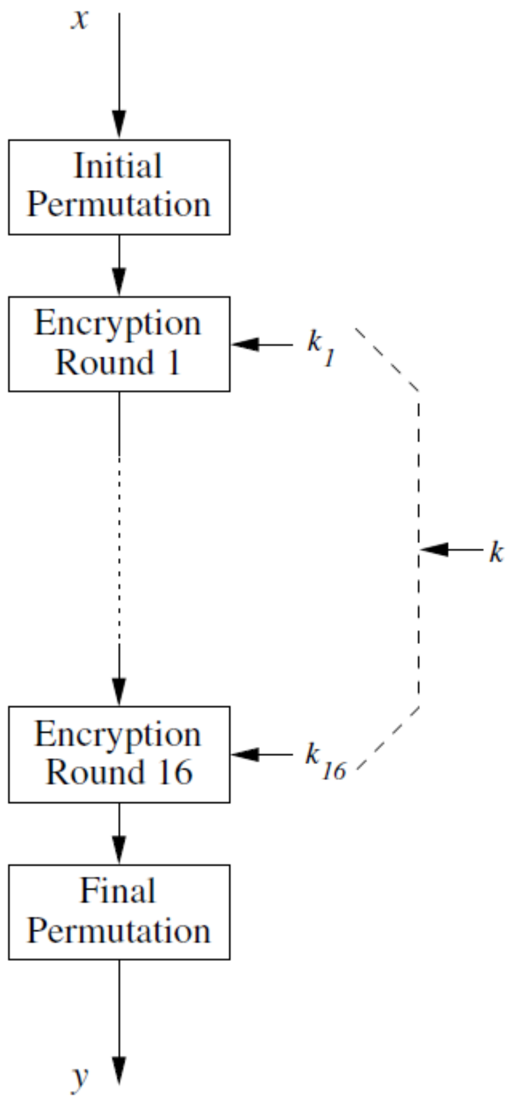
- DES: **D**ata **E**ncryption **S**tandard, a popular cipher released in 1977, developed by IBM with the assistance of the NSA
- Encrypts 64-bit blocks of plaintext, using a 56-bit key
  - DES originally had a 128-bit key, but apparently the NSA convinced IBM to reduce the key length to 56 bits
  - This made DES more vulnerable to brute-force attacks
- DES is a symmetric cipher, i.e. the same key is used for both encryption and decryption
- DES was the cipher of choice of the US government, until it was replaced by the AES cipher in 1999
  - Variants of DES are still in use, e.g. 3DES is commonly used in smart cards and credit cards



$y = DES_k(x)$

Overview of the DES algorithm

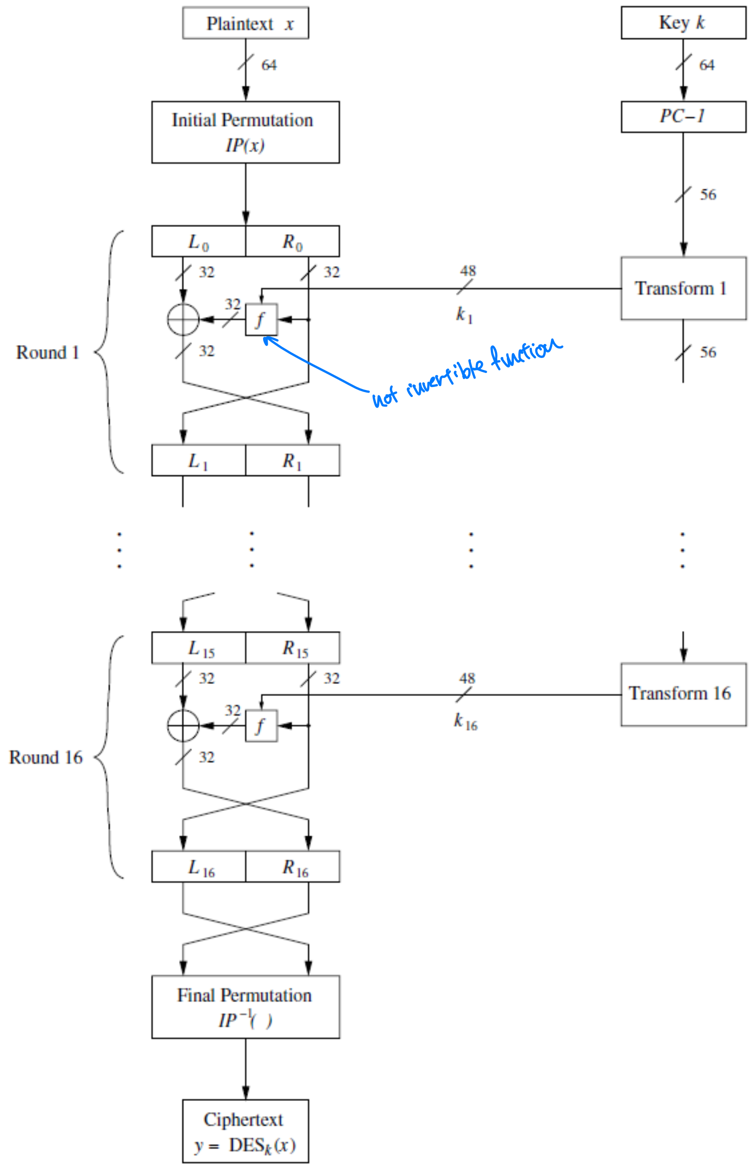
- The encryption of each 64-bit block of plaintext is handled in 16 rounds
- Each round performs the same operation
  - A different round key is used in each round; round key k_i is used in Round i — e.g. round key k_1 is used for Round 1, and so on
  - All round keys k_i are derived from the main key k (this round key derivation is known as *a key schedule*)
- Decryption is carried out in a similar manner; round key k_{16} is used for Round 1 (a decryption round is the same as an encryption round), round key k_{15} is used for Round 2, and so on, with round key k_1 used for Round 16 k_{n-i}



The DES algorithm: encryption

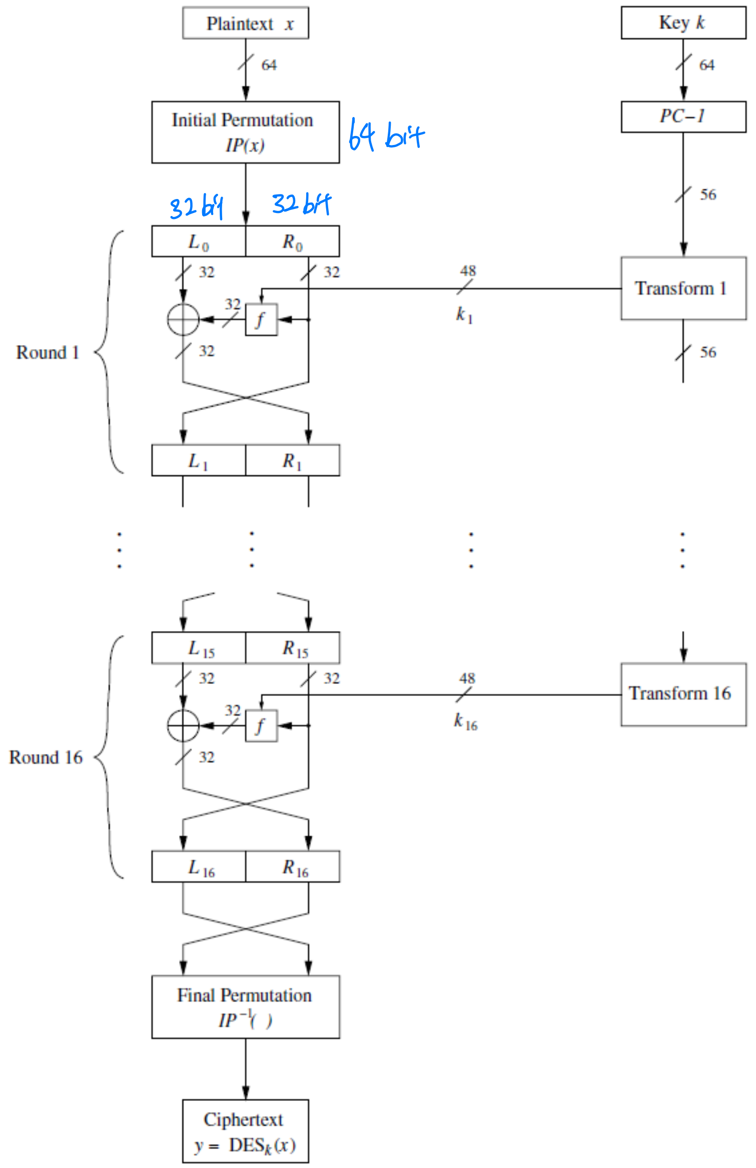
- The structure shown on the right is called a *Feistel network*
  - If designed carefully, Feistel networks can result in strong ciphers
  - One advantage of Feistel networks: encryption and decryption are almost the same process – decryption only requires a reversed key schedule
  - Feistel networks are used in many (but not all) modern block ciphers

- The key k is listed as 64 bits; this is because 8 of those bits are used as parity bits and are **not** actually key bits – thus the key size is 56 bits



The DES algorithm: encryption

1. There is an initial bitwise permutation *IP* of the 64-bit plaintext *x*
2. The plaintext is then split into two 32-bit halves L_0 and R_0, which are fed into the Feistel network, which consists of 16 rounds
3. The right half R_i is used as an input to the *f*-function, whose output is then XOR-ed with the left half L_i
4. The left and right halves are then swapped



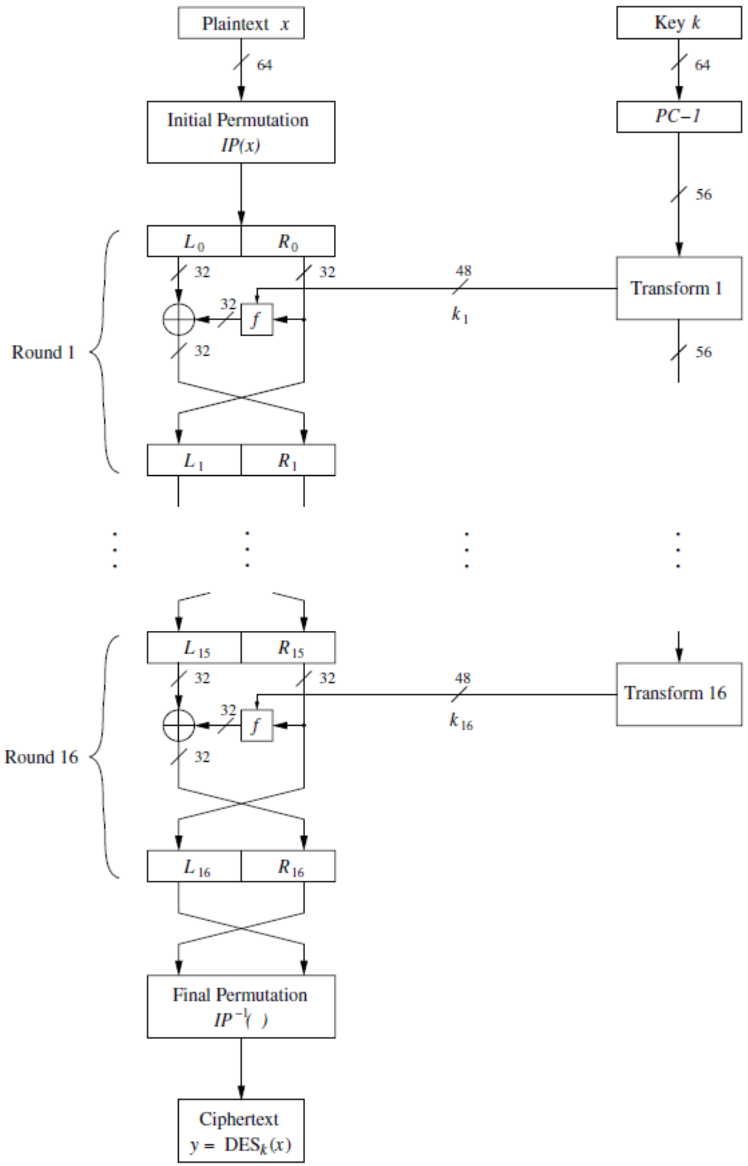
The DES algorithm: encryption

5. Steps 3 and 4 are repeated in the next round, and the operations in each round can be expressed as:

flip

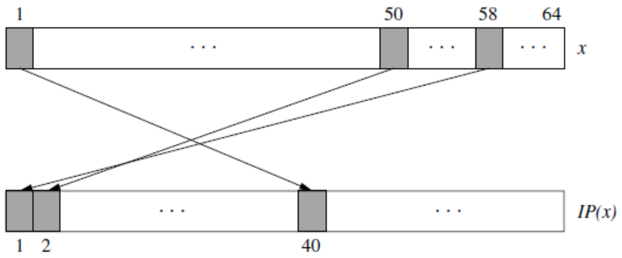
$$\boxed{\begin{aligned} L_i &= R_{i-1} \\ R_i &= L_{i-1} \oplus f(R_{i-1}, k_i), \text{ where } i = 1, \ldots, 16 \end{aligned}}$$

6. After Round 16, the 32-bit halves L_{16} and R_{16} are swapped again
7. There is a final bitwise permutation IP^{-1} of the resulting 64-bit value (IP^{-1} is the inverse of IP); this results in the 64-bit ciphertext output

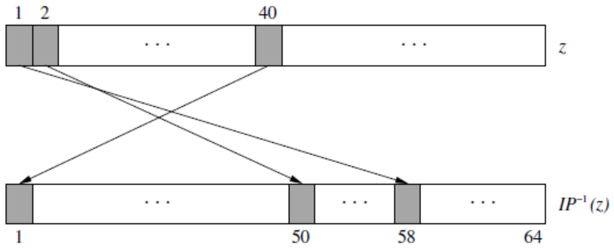


The DES algorithm: initial and final permutation

Initial Permutation *IP*



Final Permutation IP^{-1}



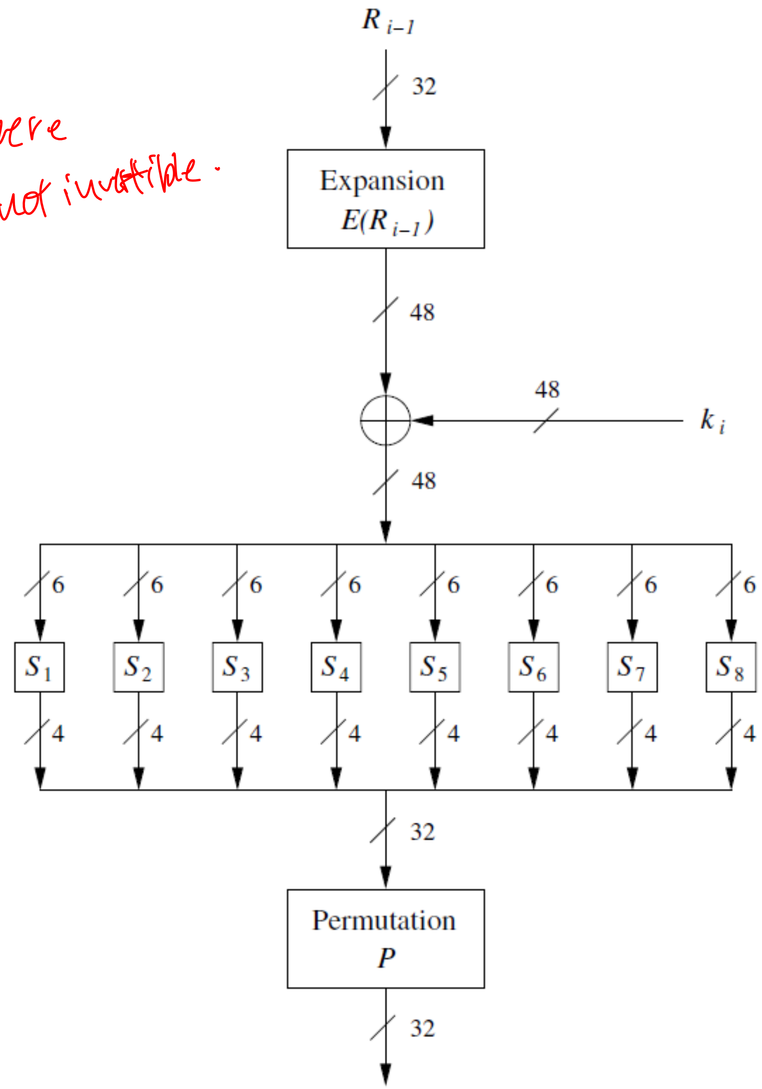
IP							
58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

IP^{-1}							
40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

- *IP*: Bit position 1 of plaintext *x* is mapped to bit position 40; position 58 mapped to 1, etc.
- IP^{-1}: Bit position 40 of *z* (output of Round 16) is mapped to bit position 1 (inverse of *IP*)

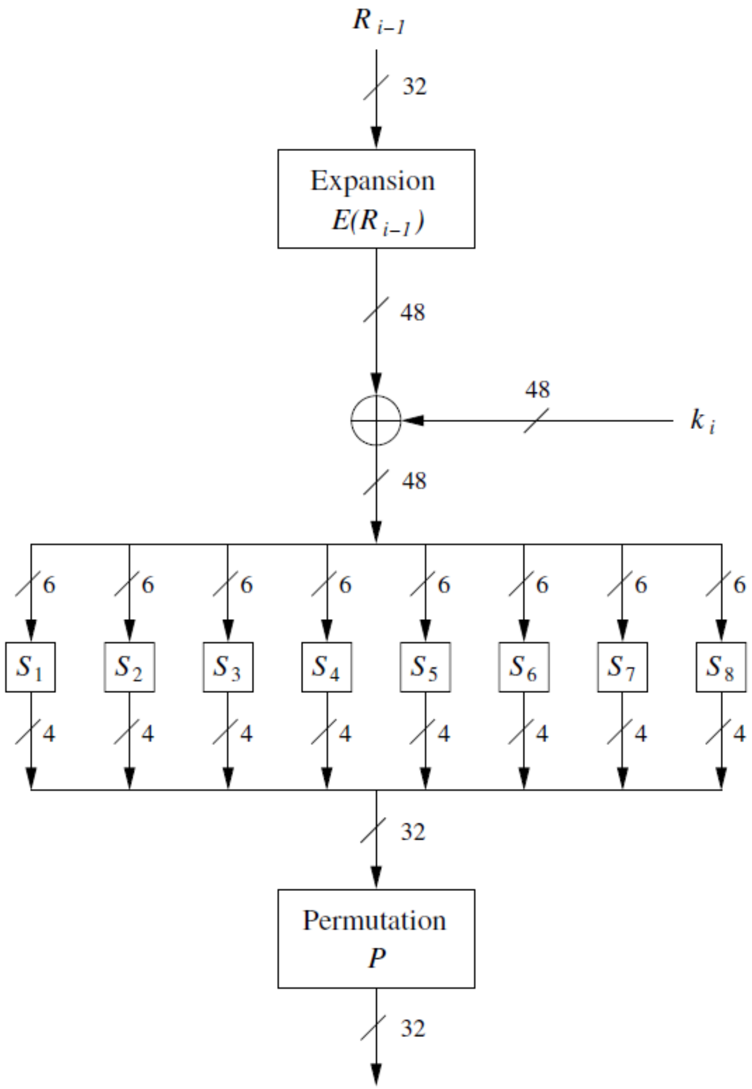
The DES algorithm: *f*-function

1. The 32-bit R_{i-1} input is expanded to 48 bits
   - 16 of the 32 input bits are copied → information loss here ↓ hence f not invertible.
   - Some permutation is also done here
2. The 48-bit result of the expansion is XOR-ed with the round key k_i, then separated into eight 6-bit blocks
3. Each 6-bit block is operated on by a different substitution box (*S*-box), resulting in a 4-bit value
4. The 4-bit blocks are recombined into a 32-bit value and then permuted bitwise using a permutation operation *P*



The DES algorithm: *f*-function

- The two primitive operations, confusion and diffusion, are realized within the *f*-function
- The *S*-boxes provide confusion
  - These *S*-boxes are non-linear, i.e.
    $S(a) \oplus S(b) \neq S(a \oplus b)$
- The permutation operation *P* provides diffusion
  - This is because the 4-bit outputs of the *S*-boxes are permuted such that they affect several different *S*-boxes in the following round



The DES algorithm: *S*-box example

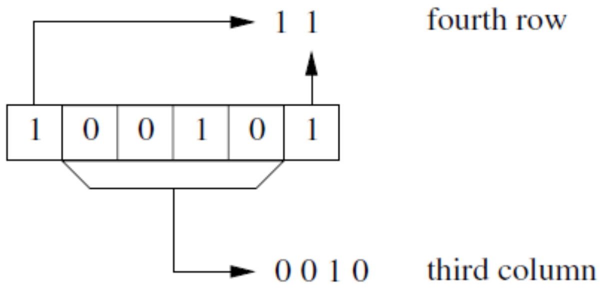
- An *S*-box is basically a lookup table which maps a 6-bit input to a 4-bit output
- E.g. *S*-box S_1 is as follows:

S_1	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	14	04	13	01	02	15	11	08	03	10	06	12	05	09	00	07
1	00	15	07	04	14	02	13	01	10	06	12	11	09	05	03	08
2	04	01	14	08	13	06	02	11	15	12	09	07	03	10	05	00
3	15	12	08	02	04	09	01	07	05	11	03	14	10	00	06	13

- The MSB and LSB of the 6-bit input select the row; the 4 inner bits select the column

decimal notation

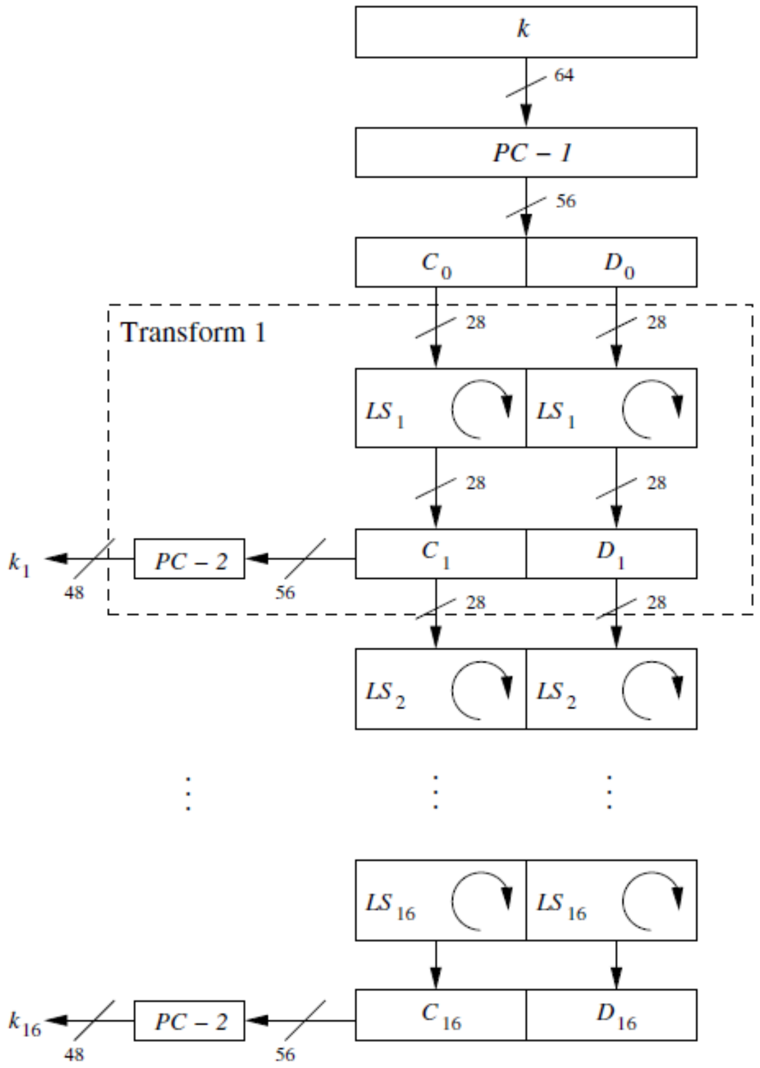
  - Suppose we have a 6-bit input 100101_2 to S_1
  - Choose the 4th row (3_{10}) and 3rd column (2_{10})
  - So the 4-bit output is $08_{10} = 1000_2$



The DES algorithm: key schedule

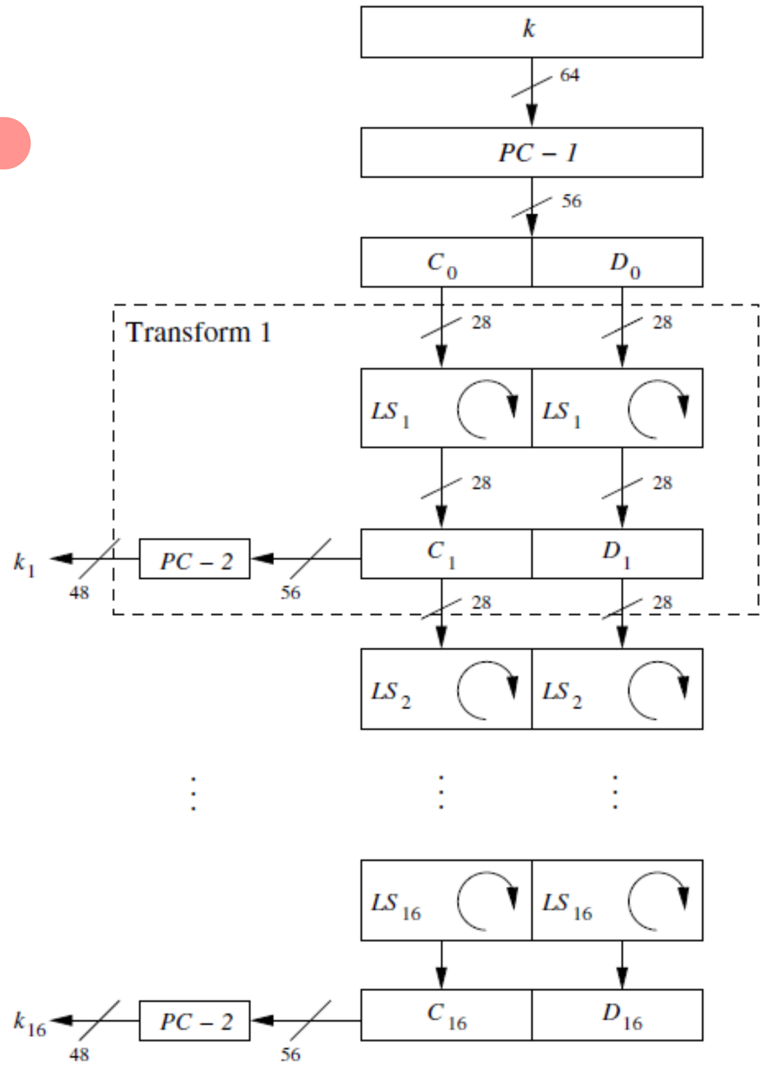
- The key schedule derives 16 round keys k_i, each consisting of 48 bits from the original 56-bit key

1. During the *PC-1* operation, the 64-bit key (including the 8 parity bits) is reduced to 56 bits by discarding every eighth bit (the parity bit) and the remaining bits are permuted
2. The resulting 56-bit value is then split into 28-bit halves C_{i-1} and D_{i-1}, and are passed on to the *Transform i* operation, for i = 1, … 16



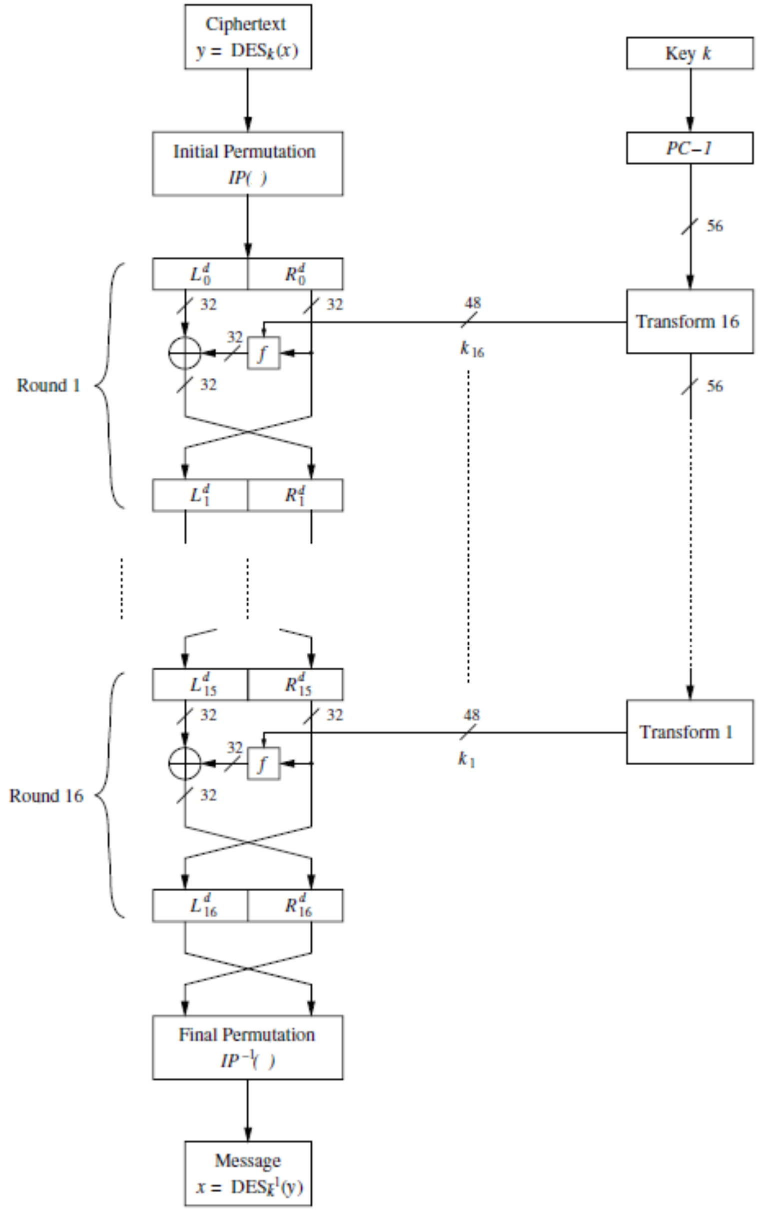
The DES algorithm: key schedule

3. Next, during the 1st phase of the *Transform i* operation, the two halves are each circular left shifted, by one or two bits depending on the round *i*, resulting in C_i and D_i
4. During the 2nd phase of the *Transform i* operation, the 56 bits from the two halves C_i and D_i are permuted using the *PC-2 operation* (also dropping 8 bits), resulting in round key k_i
5. Steps 3 and 4 are repeated for the derivation of the next round key, using the two halves C_i and D_i from the 1st phase of *Transform i*



The DES algorithm: decryption

- Decryption is carried out in a similar manner; there is an initial bitwise permutation *IP* of the ciphertext
- Then subkey k_{16} is used for round 1, round key k_{15} is used for round 2, and so on, until round key k_1 is used for round 16
- This effectively reverses the encryption in a round-by-round manner
    - Round 1 of decryption reverses round 16 of encryption, round 2 of decryption reverses round 15 of encryption, and so on until round 16 of decryption reverses round 1 of encryption
- The final bitwise permutation IP^{-1} of decryption reverses the initial bitwise permutation *IP* of encryption; this results in the original plaintext *x*



The DES algorithm: decryption

- Let's show that the decryption process reverses the encryption in a round-by round manner
  - Note that the *f*-function itself is **not** invertible

let $x = (L_0, R_0) \leftarrow$ one 64-bit block
y is the ciphertext

after permutation of y,

$(L_0^d, R_0^d) = IP(y) = IP(IP^{-1}(R_{16}, L_{16}))$
$= (R_{16}, L_{16})$
$\Rightarrow L_0^d = R_{16}, R_0^d = L_{16} = R_{15}$

- After the initial permutation *IP* of the ciphertext *y*, we have

$$(L_0^d, R_0^d) = IP(y) = IP(IP^{-1}(R_{16}, L_{16})) = (R_{16}, L_{16})$$

- So, we have

$$L_0^d = R_{16} \text{ and } R_0^d = L_{16} = R_{15}$$

after 1 round of decryption:

$L_1^d = R_0^d = R_{15}$

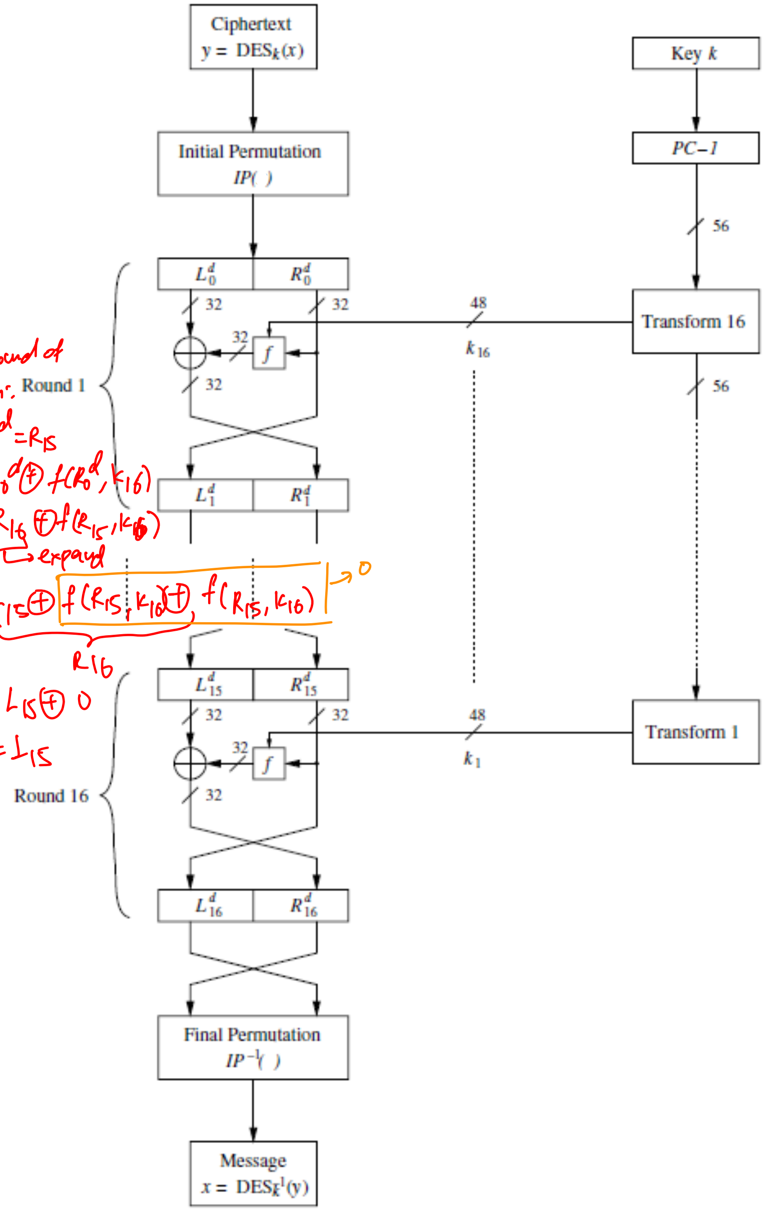
$R_1^d = L_0^d \oplus f(R_0^d, k_{16})$

$= \underbrace{R_{16}}_{\text{expand}} \oplus f(R_{15}, k_{16})$

$= \underbrace{L_{15} \oplus f(R_{15}, k_{16})}_{R_{16}} \oplus f(R_{15}, k_{16})$ → 0

$= L_{15} \oplus 0$

$= L_{15}$

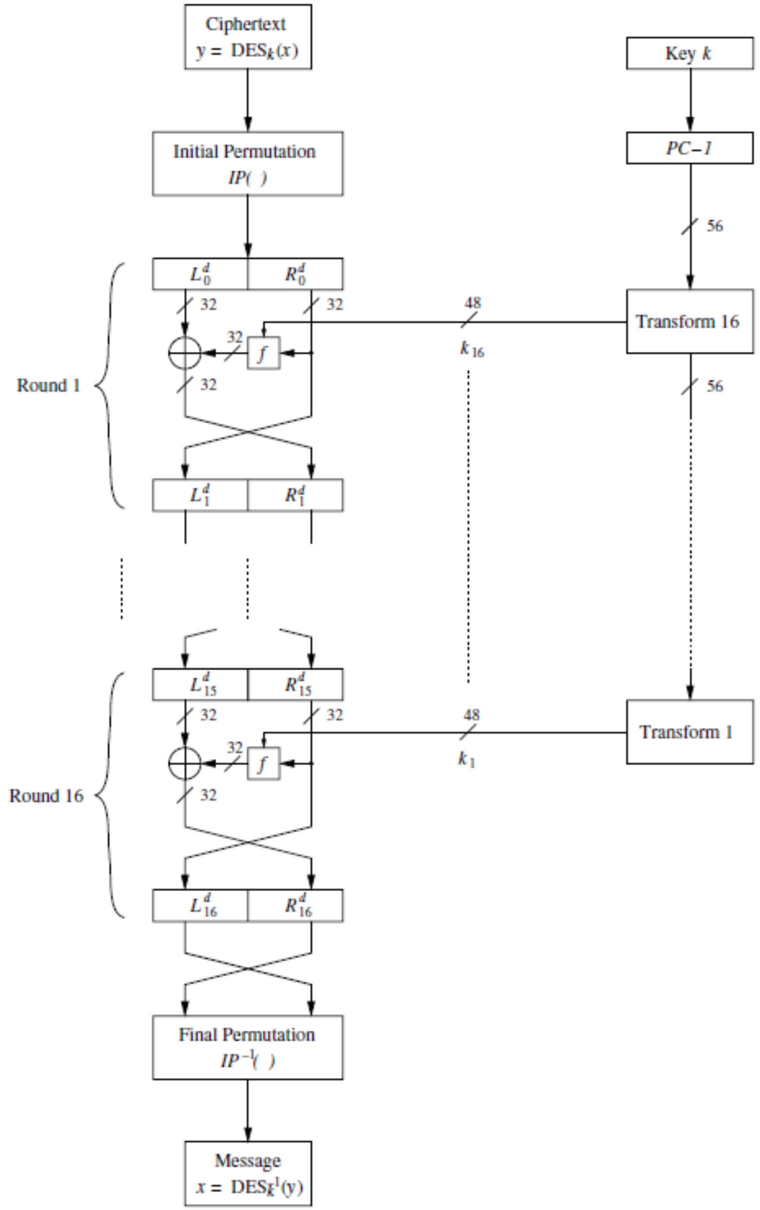


The DES algorithm: decryption

- Now the output (L_1^d, R_1^d) of round 1 of decryption can be expressed as follows:

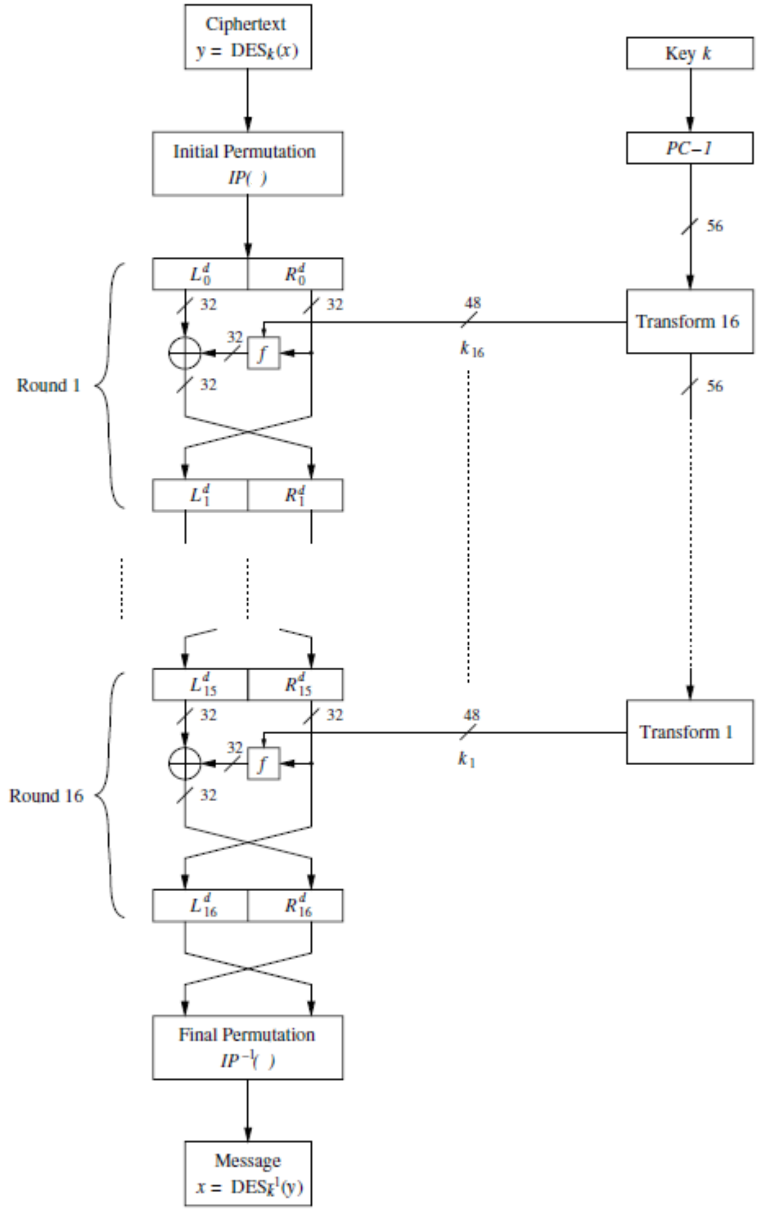
$$L_1^d = R_0^d = R_{15}$$

$$\begin{aligned} R_1^d &= L_0^d \oplus f(R_0^d, k_{16}) \\ &= R_{16} \oplus f(R_{15}, k_{16}) \\ &= L_{15} \oplus f(R_{15}, k_{16}) \oplus f(R_{15}, k_{16}) \\ &= L_{15} \oplus 0 \\ &= L_{15} \end{aligned}$$



The DES algorithm: decryption

- Thus, the output of round 1 of decryption is
  $(L_1^d, R_1^d) = (R_{15}, L_{15})$
- R_{15} and L_{15} are effectively the input bits to round 16 of <u>encryption</u>; we have shown that round 1 of decryption reverses round 16 of encryption
- So we can express the decryption process iteratively as:
  $L_i^d = R_{16-i}$
  $R_i^d = L_{16-i}$, where $i = 0, 1,\ldots, 16$



The DES algorithm: decryption

- Particularly, after round 16 of decryption, we have:

  $L_{16}{}^{d} = R_0$

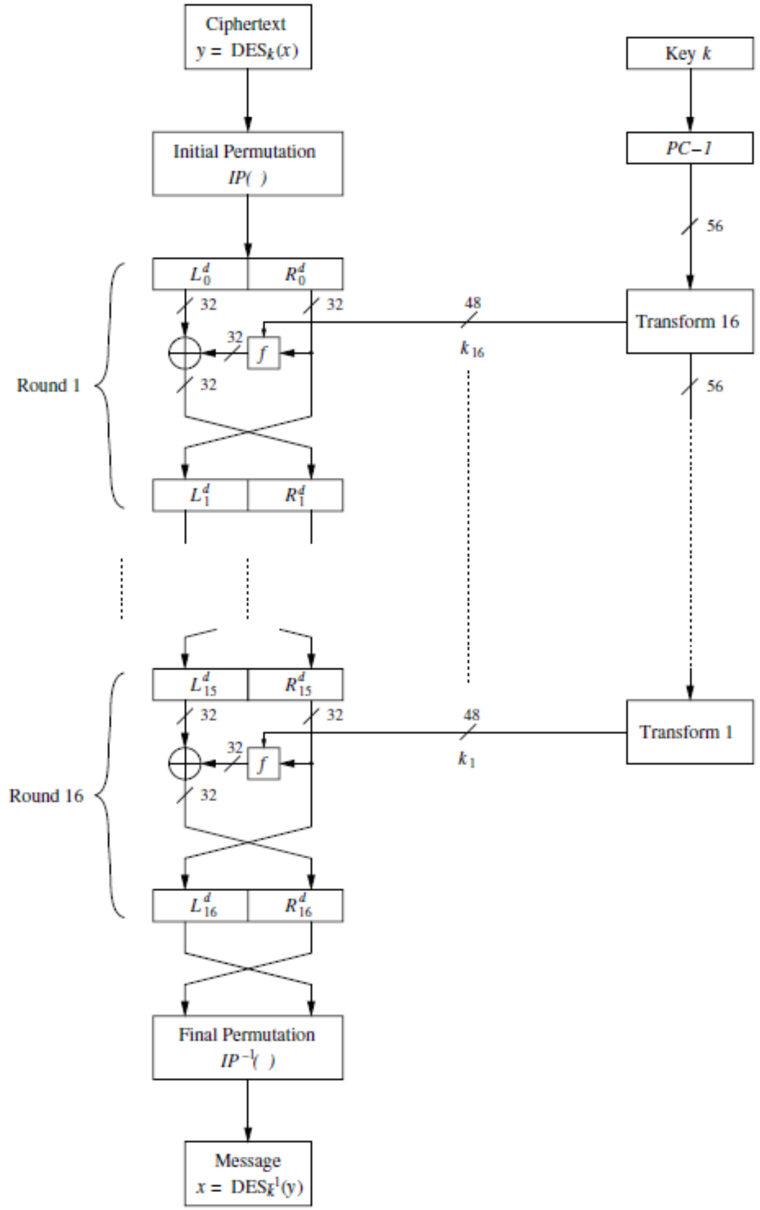
  $R_{16}{}^{d} = L_0$

- Thus, after the final bitwise permutation IP^{-1}, we have

  $IP^{-1}(R_{16}{}^{d}, L_{16}{}^{d}) = IP^{-1}(L_0, R_0)$

  $= IP^{-1}(IP(x))$

  $= x$



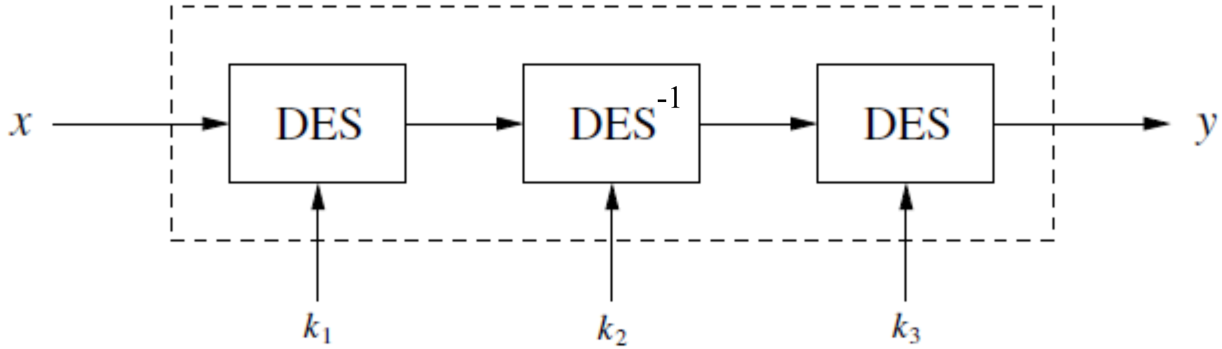
The DES algorithm: security analysis

- The DES block cipher is now vulnerable to brute-force attacks (i.e. exhaustive key search) due to the small key length of 56 bits

  much less than 2^{80}

  - Keyspace is too small for today's security requirements
  - Given at least one pair of plaintext-ciphertext *(x, y)*, an attacker can test all 2^{56} possible keys, by checking whether $DES_{k_i}^{-1}(y) = x$, for $i = 0, 1, 2, 3, \ldots, 2^{56} - 1$
- No practical analytical attacks found
- Use the 3DES variant as a replacement for DES

3DES algorithm



- DES is no longer secure, but its variant 3DES is still viable and in current use
- 3DES uses three 56-bit keys k_1, k_2, k_3 and computes the 64-bit block of ciphertext *y* as follows:

  $y = DES_{k_3}(\ DES^{-1}_{k_2}(\ DES_{k_1}(x)\ )\ )$

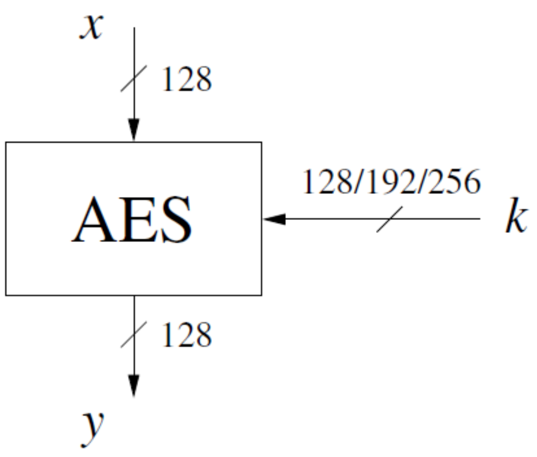
- If $k_1 = k_2 = k_3$, then 3DES effectively behaves like DES (allows for backward compatibility with legacy systems)

3DES algorithm

- 3DES is commonly used in credit cards and smart cards, for authentication of transactions
  - The 3DES block cipher is being utilized as a MAC in these situations
- 3DES increases the effective key length to 112 bits (as opposed to the 56-bit key length of DES)
  - Improved security over DES
  - Note: The effective key length of 3DES is **not** increased to 168 bits, because of the *meet-in-the-middle* attack

Overview of the AES algorithm

- AES: **A**dvanced **E**ncryption **S**tandard, introduced in 2001
  - This cipher is the result of an NIST call for proposals for a new cipher to replace DES
- Encrypts 128-bit blocks of plaintext, with a choice of key lengths of 128, 192 or 256 bits
  - To date, there are no known attacks better than brute-force attacks against AES
- AES is a symmetric cipher, i.e. the same key *k* is used for both encryption and decryption
  - Unlike DES, AES does not utilize a Feistel network
- AES is the current cipher of choice of the US government
  - Used by US federal agencies to encrypt classified documents
- Also used in protocols like IPsec, TLS, SSH



$y = AES_k(x)$

Overview of the AES algorithm

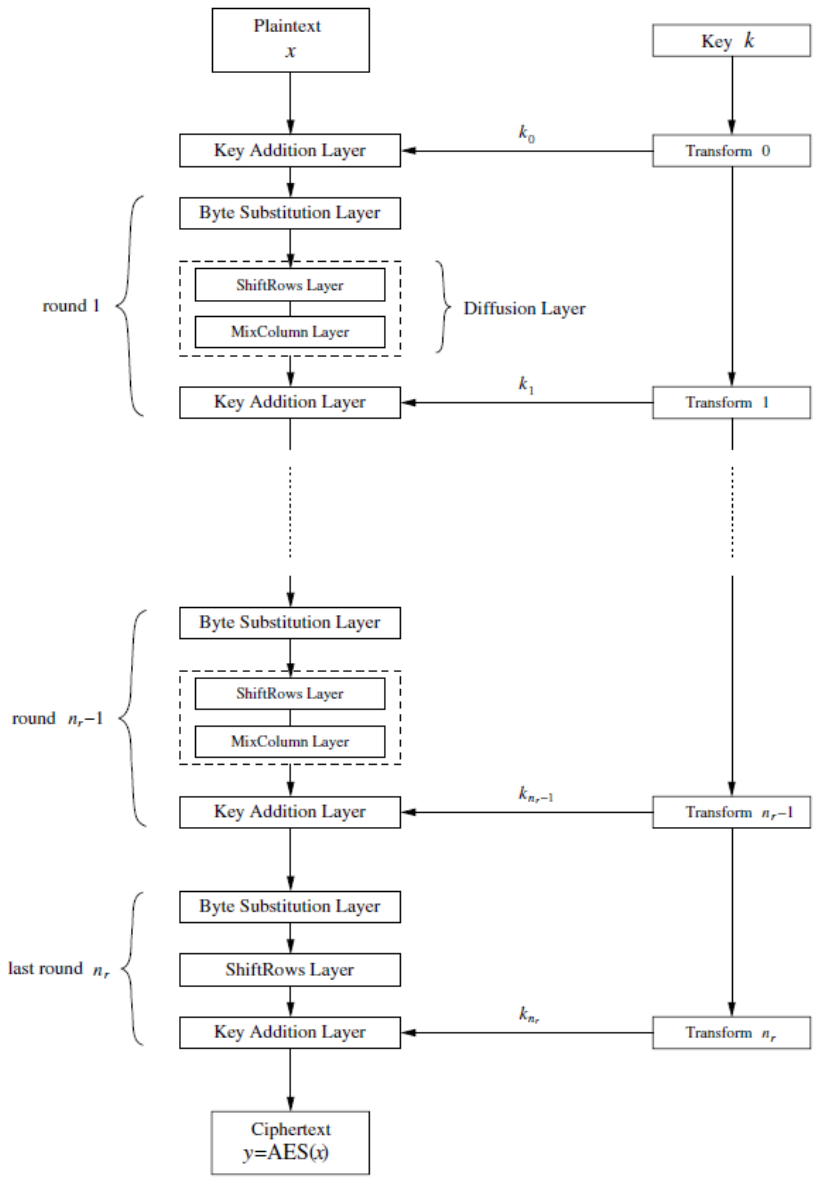
- The encryption of each 128-bit block of plaintext is handled in several rounds; the number of rounds needed, n_r, depends on the key length:
  - 128-bit key length — $n_r = 10$
  - 192-bit key length — $n_r = 12$
  - 256-bit key length — $n_r = 14$
- Each round of AES encrypts all 128 bits of the data
  - By contrast, DES only encrypts 32 bits of the 64-bit data block in each round
- Similar to DES, each round of AES uses a round key k_i which is derived from the key k using a key schedule
- AES encryption consists of *layers*, each of which manipulates all 128 bits of the data path in some manner (the data path is known as the *state*)
- Decryption requires a reversed key schedule, and all layers utilized during the encryption process need to be inverted

The AES algorithm: encryption

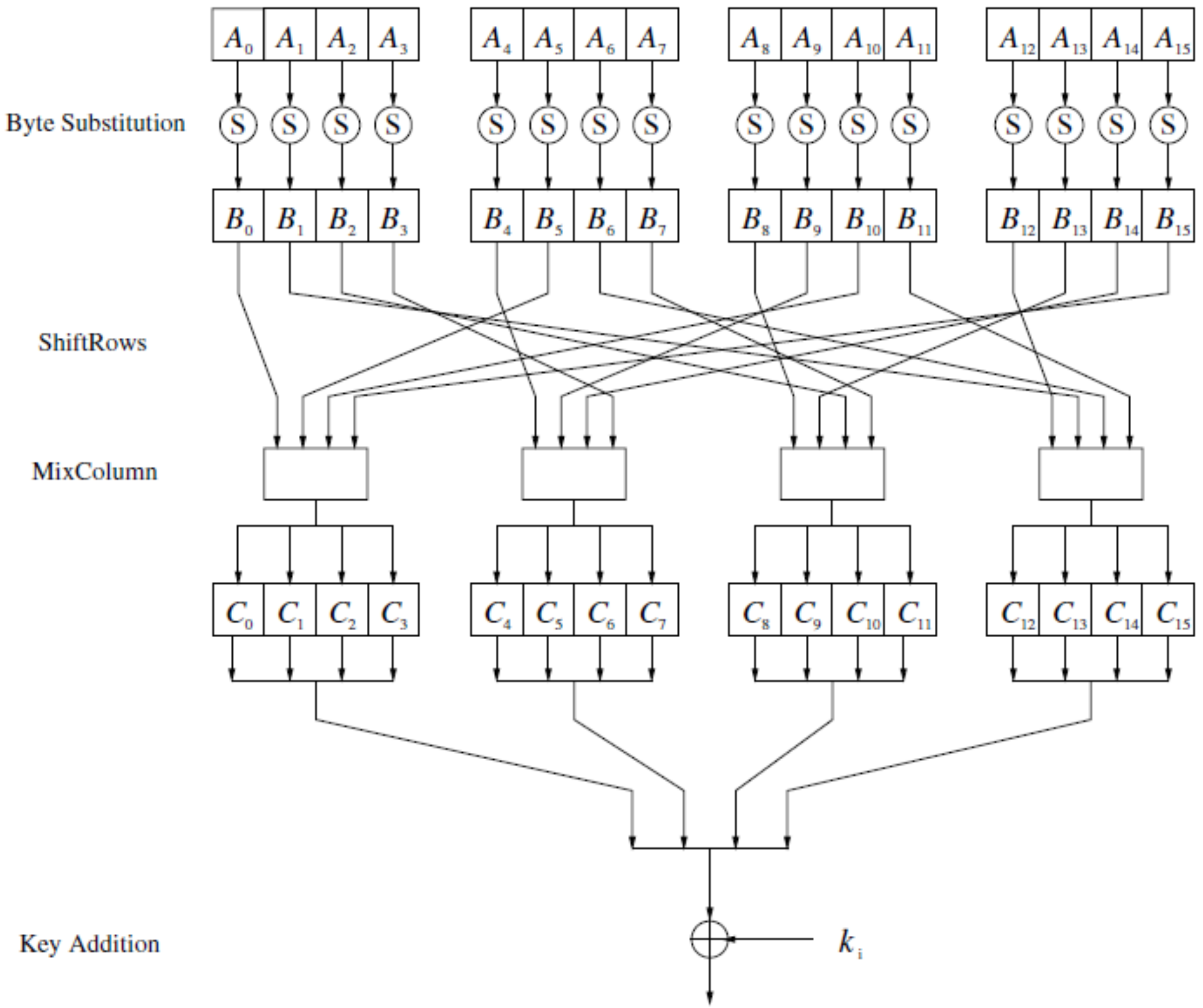
- A brief description of the three layers utilized in AES encryption is as follows
- **Key Addition layer**: A 128-bit round key is XOR-ed to the state
- **Byte Substitution layer (S-box)**: Each element of the state is non-linearly transformed using lookup tables with special mathematical properties
  - This introduces confusion to the data
- **Diffusion layer**: Provides diffusion over all state bits, and it consists of two sublayers (both sublayers are linear operations)
  - **ShiftRows sublayer**: Permutes the data on a byte level
  - **MixColumn sublayer**: A matrix operation which combines blocks of 4 bytes

The AES algorithm: encryption

- Each round of encryption uses all three layers, except the last round n_r, which does not utilize the MixColumn sublayer



The AES algorithm: encryption (single round)



The AES algorithm: encryption (single round)

- It is helpful to visualize the 16 byte data path (i.e. input state matrix) $A = (A_0, A_1, \ldots, A_{15})$ that is moving through the AES encryption process as a 4 × 4 state matrix, with each of the bytes $A_0, \ldots, A_{15}$ arranged as follows:

A_0	A_4	A_8	A_{12}
A_1	A_5	A_9	A_{13}
A_2	A_6	A_{10}	A_{14}
A_3	A_7	A_{11}	A_{15}

- AES operates on elements, columns or rows of the current state matrix

The AES algorithm: encryption (single round)

1. The 16 byte (i.e. 128 bits) input state matrix $A = (A_0, A_1, \ldots, A_{15})$ is fed byte-wise into the Byte Substitution layer
2. Then, the resulting 16 byte state matrix $B = (B_0, B_1, \ldots, B_{15})$ is permuted byte-wise in ShiftRows sublayer
3. The result of the ShiftRows sublayer is then mixed by combining selected blocks of 4 bytes in the MixColumns sublayer
4. Lastly, the output of the ShiftRows sublayer, the state matrix $C = (C_0, C_1, \ldots, C_{15})$ is XOR-ed with the round key k_i

The AES algorithm: Byte Substitution layer

- This layer is basically a row of 16 parallel S-boxes
  - Each S-box has an 8-bit input and an 8-bit output (i.e. one byte input/output)
  - All 16 S-boxes are identical, unlike DES which uses 8 different S-boxes
- Each S-box substitutes the input byte A_i with another byte B_i
  - So each S-box performs the operation $B_i = S(A_i)$
  - The substitution is a bijective mapping; each of the 2^8 possible different byte values is one-to-one mapped to another byte value
  - This allows the S-boxes to be inverted/reversed (unlike the S-boxes for DES), which is necessary for decryption
- The S-boxes are the only non-linear elements in AES
  - So $ByteSub(A) + ByteSub(B) \neq ByteSub(A + B)$ for any two arbitrary state matrices A and B

The AES algorithm: Byte Substitution layer

- S-box substitution values for an input byte (x, y):

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

- E.g. If A_0 = 0xF3, then $B_0 = S(A_0) = S($0xF3$)$ = 0x0D

The AES algorithm: Byte Substitution layer

- The AES S-box implementation of one-to-one byte mappings is **not** based on some randomly chosen set of mappings (unlike DES)
- The S-box implementation is actually a two step mathematical transformation (non-linear)
  - The 1st step is a Galois field inversion for the Galois extension field $GF(2^8)$
  - The 2nd step is an affine mapping

- ***Note: We will discuss Galois fields (which includes extension fields) and operations in these fields (addition, subtraction, multiplication and inversion) during next week's lectures on modular arithmetic***

The AES algorithm: ShiftRows sublayer

- The ShiftRows sublayer is part of the Diffusion layer
  - Note: The Diffusion layer is linear, i.e. $Diff(A) + Diff(B) = Diff(A + B)$ for any two arbitrary state matrices A and B

- The ShiftRows sublayer cyclically shifts the rows of the input state matrix to the right
  - First row: no shift
  - Second row: cyclic shift by 3 bytes to the right
  - Third row: cyclic shift by 2 bytes to the right
  - Fourth row: cyclic shift by 1 byte to the right

The AES algorithm: ShiftRows sublayer

- In other words, the input state matrix $B = (B_0, B_1, \ldots, B_{15})$ results in the following output:

$$B = \begin{array}{|c|c|c|c|}\hline B_0 & B_4 & B_8 & B_{12} \\ \hline B_1 & B_5 & B_9 & B_{13} \\ \hline B_2 & B_6 & B_{10} & B_{14} \\ \hline B_3 & B_7 & B_{11} & B_{15} \\ \hline \end{array} \rightarrow B_{SR} = \begin{array}{|c|c|c|c|}\hline B_0 & B_4 & B_8 & B_{12} \\ \hline B_5 & B_9 & B_{13} & B_1 \\ \hline B_{10} & B_{14} & B_2 & B_6 \\ \hline B_{15} & B_3 & B_7 & B_{11} \\ \hline \end{array} \begin{array}{l} \text{no shift} \\ \longrightarrow \text{three positions right shift} \\ \longrightarrow \text{two positions right shift} \\ \longrightarrow \text{one position right shift} \end{array}$$

The AES algorithm: MixColumn sublayer

- The MixColumn sublayer is a linear transformation which mixes each column of the input state matrix
  - It is also part of the Diffusion layer
  - The MixColumn operation is the major diffusion element in AES, since every input byte influences four output bytes

- The operation performed is $C = MixColumn(B_{SR})$, where B_{SR} is the input state matrix after the ShiftRows operation is executed
- Each 4-byte column of B_{SR} is treated as a vector and multiplied by a fixed constant 4×4 matrix. Multiplication and addition of the coefficients is done in the Galois extension field $GF(2^8)$

The AES algorithm: MixColumn sublayer

- E.g. For the first column of C and B_{SR} respectively:

$$\begin{pmatrix} C_0 \\ C_1 \\ C_2 \\ C_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} B_0 \\ B_5 \\ B_{10} \\ B_{15} \end{pmatrix}$$

- Each state byte C_i and B_i with $i = 0, \ldots, 15$, is an 8-bit value that represents an element from the Galois extension field GF(2^8)
- Here, column (C_0, C_1, C_2, C_3) is the result of a matrix-vector multiplication of the fixed constant 4 × 4 matrix with the first column of B_{SR}
  - Additions in the matrix-vector multiplication are $GF(2^8)$ additions (these are basically bitwise XOR operations)
  - Multiplications in the matrix-vector multiplication are $GF(2^8)$ multiplications

The AES algorithm: Key Addition layer

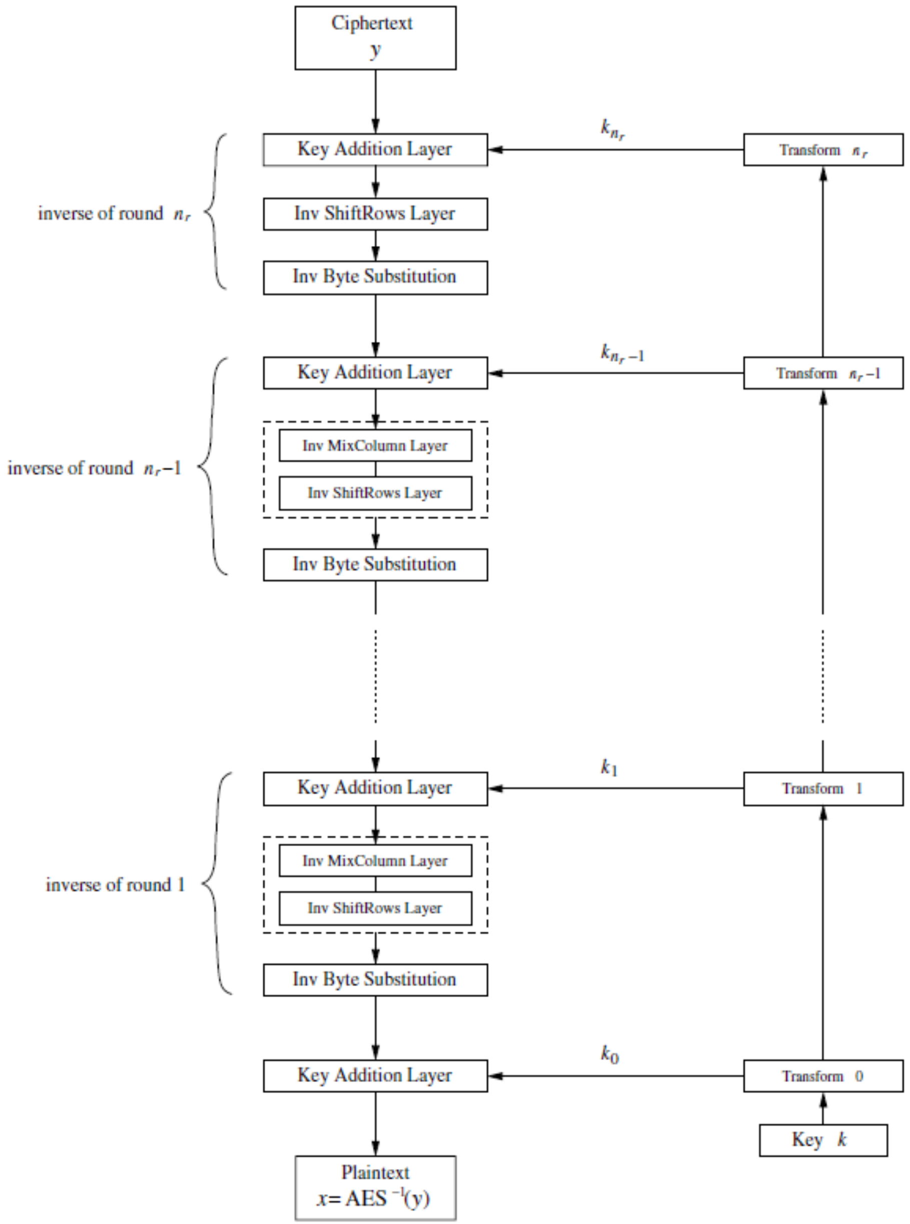
- The two inputs to this layer are the 16-byte state matrix *C* and the round key k_i (also 16 bytes), where *i* is the *i*th round
- The round keys are derived using the key schedule
- The two inputs are combined using a bitwise XOR operation
  - The is equivalent to addition in the Galois extension field $GF(2^8)$
  - Recall that bitwise XOR operation is also equivalent to addition modulo 2, which means that it is also addition in the Galois field $GF(2)$

The AES algorithm: key schedule

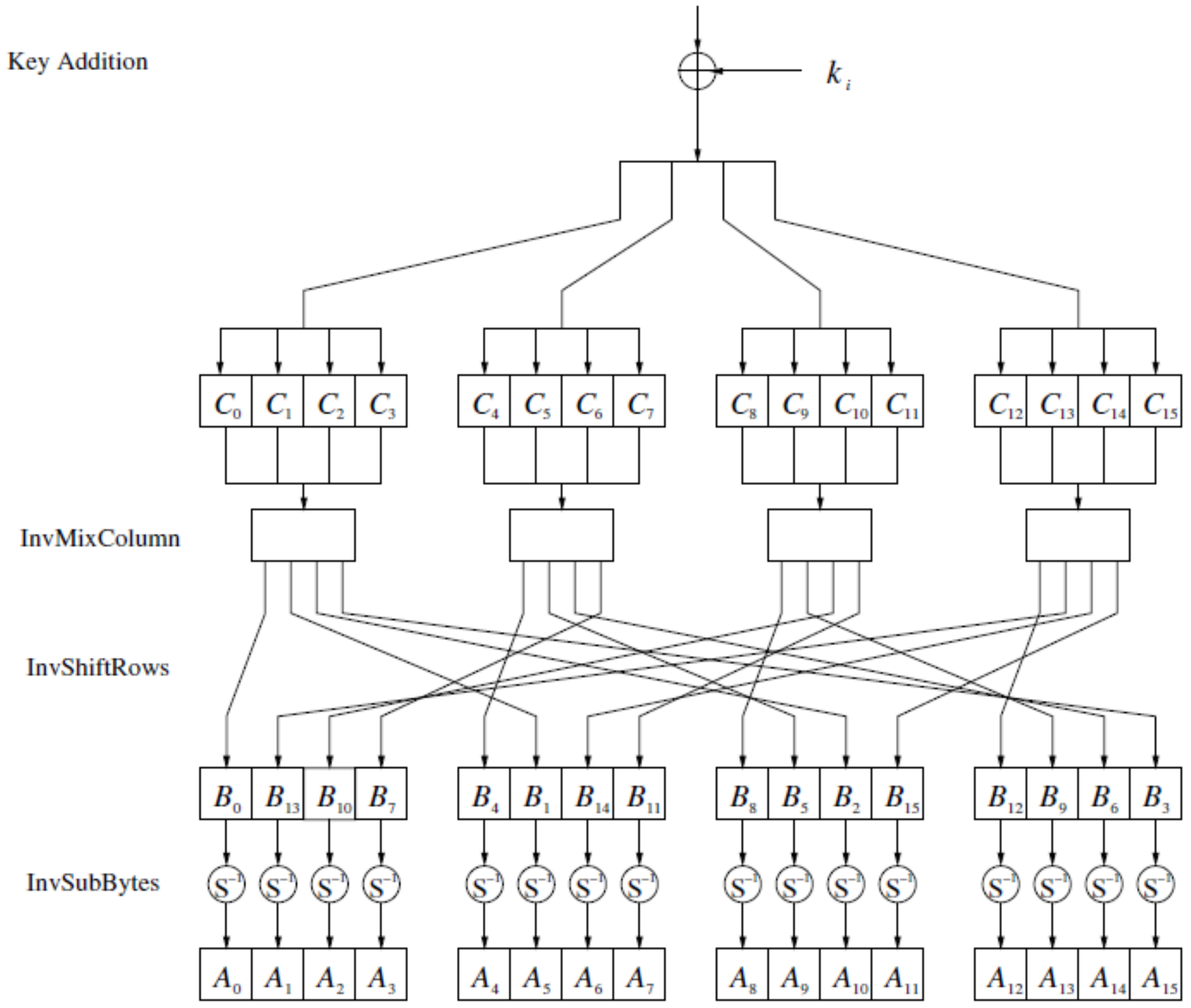
- The key schedule takes the original input key (128-bit, 192-bit or 256-bit) and derives the round keys k_i for i = 0, 1, …, n_r
  - 128-bit key length — n_r = 10 (11 round keys)
  - 192-bit key length — n_r = 12 (13 round keys)
  - 256-bit key length — n_r = 14 (15 round keys)
- The round keys are computed iteratively
  - i.e. round key k_i must first be computed in order to derive round key k_{i+1}, and so on
- Different key schedules for the three different AES key sizes (128-bit, 192-bit or 256-bit)

The AES algorithm: decryption

- AES is not based on a Feistel network, so all layers need to be inverted for the decryption process (and executed in reverse order)
  - Byte Substitution layer becomes Inv Byte Substitution layer
  - ShiftRows sublayer becomes Inv ShiftRows
  - MixColumn sublayer becomes Inv MixColumn
  - Key Addition layer remains unchanged, because the XOR operation is its own inverse
- A reversed key schedule is also required



The AES algorithm: decryption (single round)



Block cipher modes

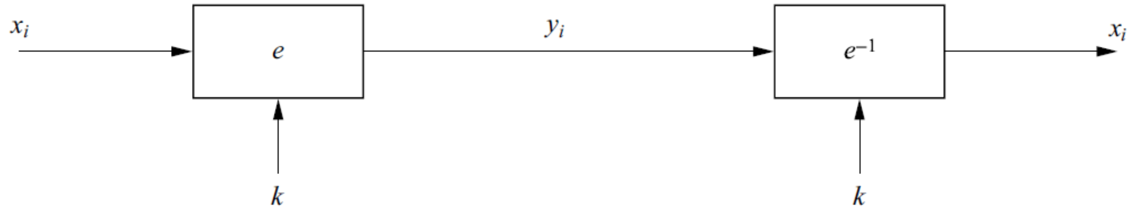
- So far, we have discussed the DES and AES block ciphers with respect to their operations on <u>one block of the plaintext</u>
  - What about encrypting a long plaintext consisting of multiple blocks?
- It turns out that there are several ways to encrypt a long plaintext with a block cipher
- One obvious way: Apply the block cipher to each block of the plaintext individually (after padding)
  - This is the **E**lectronic **C**ode **B**ook mode (ECB)

Block cipher modes

- Other modes include:
  - **C**ipher **B**lock **C**haining mode (CBC)
  - **C**ipher **F**eed**b**ack mode (CFB)
  - **O**utput **F**eed**b**ack mode (OFB)
  - **C**oun**t**e**r** mode (CTR)
- ECB and CBC are actual block ciphers
- CFB, OFB and CTR use a block cipher as a building block for a *stream* cipher
  - These modes use the block cipher to generate a key stream s_i
- Let's discuss each of these five modes in more detail

Block cipher modes: ECB

- ECB mode is the most straightforward way to encrypt a message using a block cipher
  - Requires that the length of the plaintext be an exact multiple of the input block size of the block cipher used (so padding is needed)
- Encryption and decryption in ECB mode:



Let *e()* be a block cipher in ECB mode with block size *b* and key *k*, and x_i and y_i be bitstrings of length *b*

Encryption: $y_i = e_k(x_i)$, *for* $i \geq 1$

Decryption: $x_i = e_k^{-1}(y_i) = e_k^{-1}(\ e_k(x_i)\ )$, *for* $i \geq 1$

Block cipher modes: comments on ECB

- Advantages of ECB:
  - Can be parallelized, suitable for high-speed implementations
  - Block synchronization between the sending and receiving parties is not necessary
- Disadvantage of ECB:
  - Encrypts in a very deterministic manner; i.e. assuming the key does not change, encrypting a plaintext block always result in the same ciphertext block

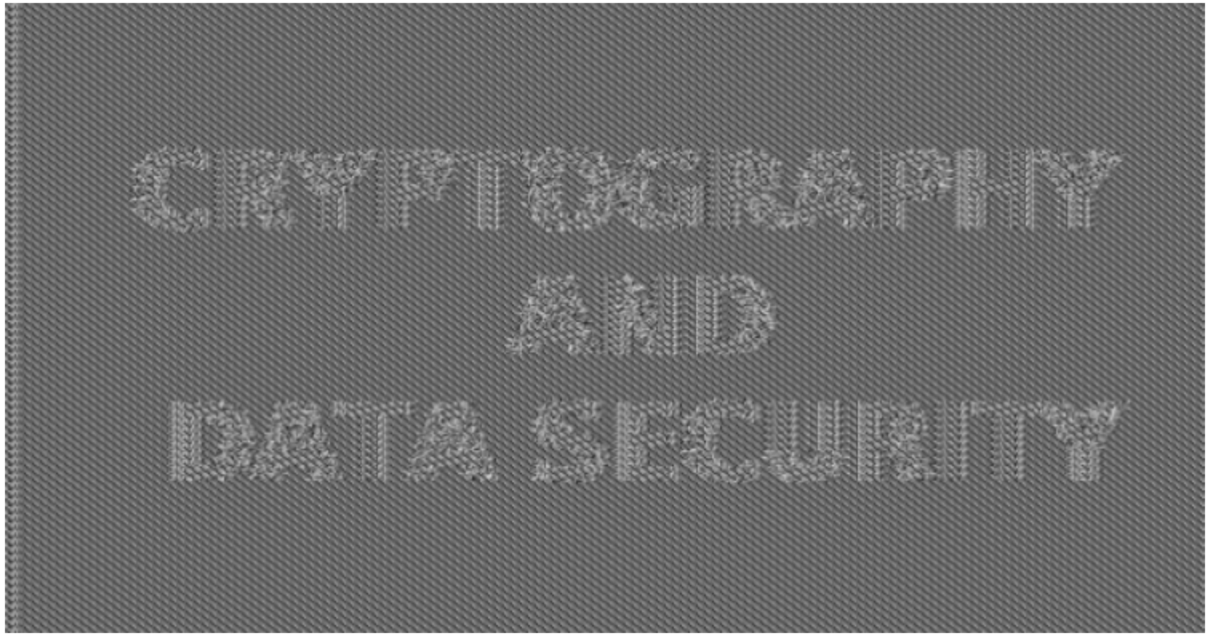
Block cipher modes: comments on ECB

- A block cipher in ECB mode encrypts in a very deterministic manner
- As such, it is vulnerable to traffic analysis – an attacker can recognize if the same plaintext message has been sent twice by simply looking at the ciphertext (note: the attacker does **not** know the contents of the plaintext)
- Also, an attacker can *reorder* the ciphertext blocks transmitted by the sending party, which might lead to a valid (but different) plaintext
- So a block cipher in ECB mode is vulnerable to substitution attacks
  - Attacker manipulates the ciphertext by replacing some ciphertext blocks with other ciphertext blocks that are observed and/or manipulated by the attacker
  - Note that this is **not** an attack that breaks the block cipher itself – this is a message *integrity* issue (recall Lectures 3 & 4)

Block cipher modes: comments on ECB

- A block cipher in ECB mode encrypts in a very deterministic manner
  - Identical plaintext values are mapped to the same ciphertext values (on a per-block basis)
- E.g. Encrypting a bitmap image using AES in ECB mode with a 256-bit key:

→

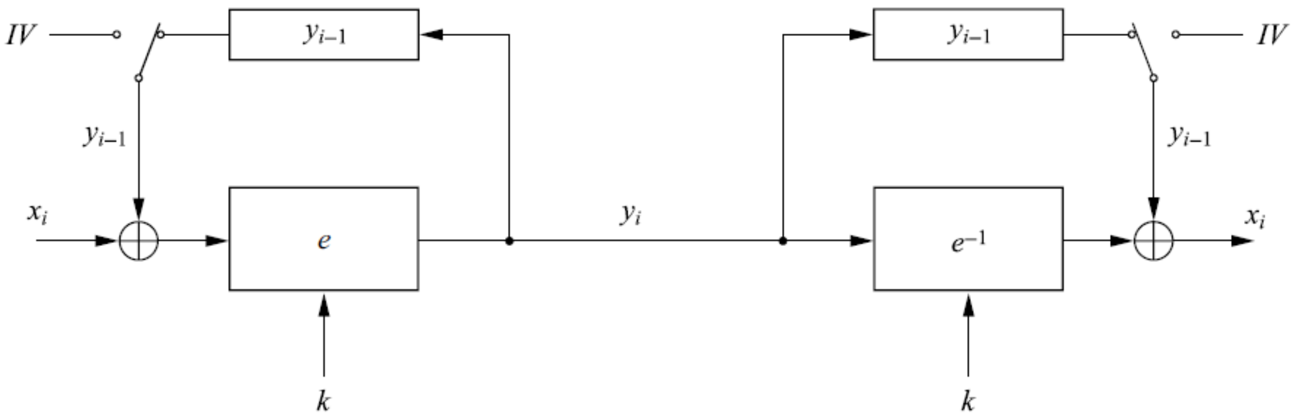


Block cipher modes: CBC

- CBC mode addresses the security issue of the ECB mode
- Encryption is randomized using an initialization vector (*IV*)
  - The *IV* is typically randomly generated and is used only once (ideally), thus it is also known a *nonce* (i.e. “number only used once”)
  - Note that the *IV* does **not** need to be kept secret; it can be transmitted over an insecure channel between the sending and receiving parties
  - Choose a new *IV* each time the message is encrypted
- The ciphertext block y_i depends not only on plaintext block x_i, it also depends on all previous plaintext blocks $x_{i-1}, \ldots, x_1$

Block cipher modes: CBC

- Encryption and decryption in CBC mode:



Let *e()* be a block cipher in CBC mode with block size *b* and key *k*, and x_i and y_i be bitstrings of length *b*, and *IV* be a nonce of length *b*

number used once.

Encryption (first block): $y_1 = e_k(x_1 \oplus IV)$

Encryption (subsequent blocks): $y_i = e_k(x_i \oplus y_{i-1})$, *for* $i \geq 2$

Decryption (first block): $x_1 = e_k^{-1}(y_1) \oplus IV$

Decryption (subsequent blocks): $x_i = e_k^{-1}(y_i) \oplus y_{i-1}$, *for* $i \geq 2$

Block cipher modes: comments on CBC

- Assuming the *IV* is changed every time encryption is done, then substitution attacks will no longer work
  - Encrypting the same plaintext twice results in different ciphertexts
- However, the attacker can still cause mischief: he can substitute the original ciphertext blocks with garbage blocks (though he can no longer effectively manipulate the original ciphertext blocks)

Block cipher modes: comments on CBC

- Basic point is that encryption itself is insufficient – message integrity also needs to be protected; hence the need for hashes and MACs as mentioned earlier
- On that note, the CBC mode *could* be used to generate a MAC – Alice can transmit the <u>last</u> ciphertext block y_n as a MAC (instead of all of the y_i blocks), along with the plaintext message x
  - This is known as CBC-MAC
  - To verify message integrity, Bob repeats the **encryption** process of the block cipher in CBC mode on x to obtain y_n', then compares y_n' with the received y_n
  - <u>**However**</u>, this is not a really secure way to generate a MAC

Block cipher modes: comments on CBC

- Why is the CBC-MAC mode not a secure MAC function?
- Let $x = (x_1, x_2)$ and $h_i = e_k(h_{i-1} \oplus x_i)$, with $h_0 = IV$
  So $h_1 = e_k(h_0 \oplus x_1) = e_k(IV \oplus x_1)$, and $h_2 = e_k(h_1 \oplus x_2)$
- Now let's craft an alternate plaintext $x' = (x_1', x_2')$ such that:
  $x_1' = IV \oplus h_1 \oplus x_2$ and $x_2' = h_2 \oplus h_1 \oplus x_2$
- Then $h_1' = e_k(h_0 \oplus x_1') = e_k(IV \oplus x_1') = e_k(IV \oplus IV \oplus h_1 \oplus x_2) = e_k(h_1 \oplus x_2)$
  and $h_2' = e_k(h_1' \oplus x_2') = e_k[e_k(h_1 \oplus x_2) \oplus h_2 \oplus h_1 \oplus x_2]$
  $= e_k[e_k(h_1 \oplus x_2) \oplus e_k(h_1 \oplus x_2) \oplus h_1 \oplus x_2] = e_k(h_1 \oplus x_2) = h_2$
- So, we have $x' \neq x$ but $h_2' = h_2$ and we obtain a collision; this MAC function does not meet the second preimage resistance requirement

CBC-MAC → another example of 2nd preimage collision.

let $x = x_1$ → one block message

with $h_i = e_k(h_{i-1} \oplus x_i)$, $h_0 = IV$

$h_1 = e_k(h_0 \oplus x_1)$

$= e_k(IV \oplus x_1)$

lets make alternate msg: $x' = (x_1', x_2')$ → 2 block message.

with $x_1' = x_1$

$x_2' = IV \oplus h_1 \oplus x_1$

$\Rightarrow h_1' = e_k(h_0 \oplus x_1') = e_k(IV \oplus x_1) = h_1$

$h_2' = e_k(h_1' \oplus x_2') = e_k(\cancel{h_1} \oplus IV \oplus \cancel{h_1} \oplus x_1)$

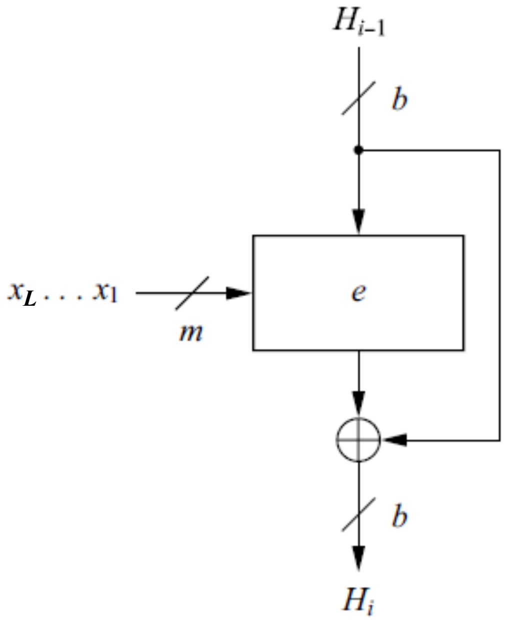
$= e_k(IV \oplus x_1) = h_1$

we have $h_2' = h_1$

→ MAC are the same for x and x'.

Block cipher modes: comments on CBC

- A better way to use a block cipher as a hash function (Davies-Meyer):



- This function sequentially utilizes the plaintext blocks x_i as the keys k_i (with a b-to-m mapping function, where m is the key length of the block cipher)
- $h_i = h_{i\text{-}1} \oplus e_{x_i}(h_{i\text{-}1})$, where $e_{x_i}()$ is the encryption function of the block cipher

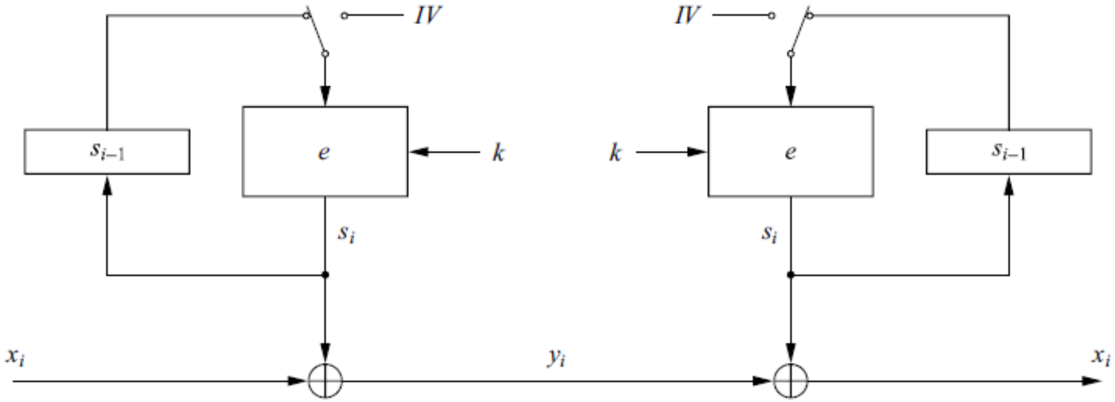
Block cipher modes: OFB

output feedback mode.

- OFB mode uses a block cipher as a building block for a stream cipher
  - It uses the block cipher to generate a key stream s_i
  - The key stream is **not** generated bitwise; it is generated in a block-wise manner
- An *IV* (should be a nonce) is encrypted using the block cipher and the cipher output is the first set of *b* key stream bits
- The block cipher output is also fed back into the block cipher to generate the next block of *b* key stream bits
- This process is repeated as necessary
- The actual encryption of the plaintext is handled by the XOR function, just like in a regular stream cipher – the plaintext bits are XOR-ed with the key stream bits

Block cipher modes: OFB

- Encryption and decryption in OFB mode:



Let *e()* be a block cipher with block size *b* and key *k*, with x_i, y_i and s_i being bitstrings of length *b*, and let *IV* be a nonce of length *b*

Encryption (first block): $s_1 = e_k(IV)$, $y_1 = s_1 \oplus x_1$

Encryption (subsequent blocks): $s_i = e_k(s_{i-1})$, $y_i = s_i \oplus x_i$, *for* $i \geq 2$

Decryption (first block): $s_1 = e_k(IV)$, $x_1 = s_1 \oplus y_1$

Decryption (subsequent blocks): $s_i = e_k(s_{i-1})$, $x_i = s_i \oplus y_i$, *for* $i \geq 2$

Block cipher modes: comments on OFB

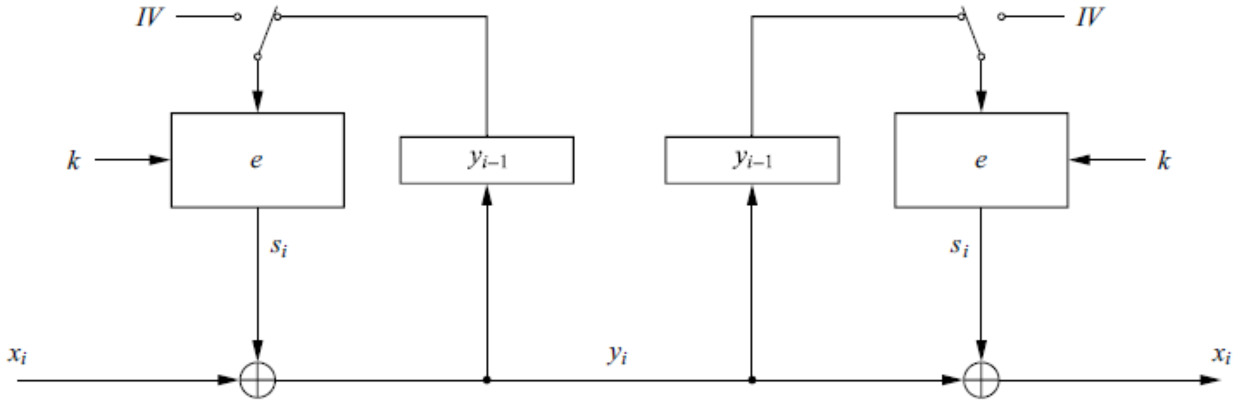
- Like CBC mode, the encryption in OFB mode is nondeterministic
  - i.e. encrypting the same plaintext twice results in different ciphertexts
- Still need to protect message integrity, as before
- Block cipher computations are independent of the plaintext bits, so several blocks of the key stream bits can be computed in advance

Block cipher modes: CFB

- Like OFB mode, CFB mode also uses a block cipher as a building block for a stream cipher; the key stream is generated block-wise
- An *IV* is encrypted using the block cipher and the cipher output is the first set of b key stream bits (just like OFB)
- The difference now is that ciphertext output y_1 (which is the output of the **stream cipher**) is fed back into the block cipher to generate the next block of b key stream bits
- The next ciphertext output y_2 is then fed back into the block cipher to generate the following block of b key stream bits
- This process is repeated as necessary

Block cipher modes: CFB

- Encryption and decryption in CFB mode:



Let *e()* be a block cipher with block size *b* and key *k*, with x_i and y_i being bitstrings of length *b*, and let *IV* be a nonce of length *b*

Encryption (first block): $y_1 = e_k(IV) \oplus x_1$

Encryption (subsequent blocks): $y_i = e_k(y_{i-1}) \oplus x_i$, *for* $i \geq 2$

Decryption (first block): $x_1 = e_k(IV) \oplus y_1$

Decryption (subsequent blocks): $x_i = e_k(y_{i-1}) \oplus y_i$, *for* $i \geq 2$

Block cipher modes: comments on CFB

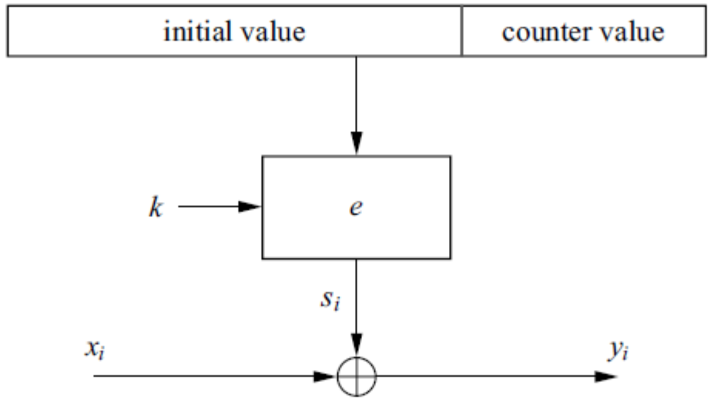
- The encryption in CFB mode is nondeterministic, like OFB
  - i.e. encrypting the same plaintext twice results in different ciphertexts
- Still need to protect message integrity, as always

Block cipher modes: CTR

- CTR mode also uses a block cipher as a building block for a stream cipher; the key stream is generated block-wise
- The input to the block cipher is a counter whose value changes every time the block cipher computes a new key stream block
- Care needs to be taken with the initialization of the input to the block cipher, to avoid using the same input value twice
  - This can be achieved by concatenating some *IV* (that is a nonce with a length smaller than the block size of the block cipher) with the counter value
  - Length of the *IV* and counter value is equal to the block size
  - E.g. Assuming AES is used as the block cipher, choose 96 bits for the *IV* and 32 bits for the counter value, for a total of 128 bits
  - Counter value is initialized to zero
  - Each time a block is encrypted, the *IV* stays the same while the counter is incremented

Block cipher modes: CTR

- Encryption in CTR mode:



Let *e()* be a block cipher with block size *b* and key *k*, with x_i and y_i being bitstrings of length *b*, and let $IV \,||\, CTR_i$ denote the concatenation of the initialization value *IV* with the counter value CTR_i; this concatenation is a bitstring of length *b*

Encryption: $y_i = e_k(IV \,||\, CTR_i) \oplus x_i$, *for* $i \geq 1$

Decryption: $x_i = e_k(IV \,||\, CTR_i) \oplus y_i$, *for* $i \geq 1$

Block cipher modes: comments on CTR

- The encryption in CTR mode is nondeterministic, like OFB and CFB
  - i.e. encrypting the same plaintext twice results in different ciphertexts
- Still need to protect message integrity
- CTR mode can be parallelized, unlike OFB and CFB
  - E.g. two block cipher devices running in parallel – the first device encrypts with $IV \,||\, CTR_1$ and the second one encrypts with $IV \,||\, CTR_2$
  - Then after they are done, the first device encrypts with $IV \,||\, CTR_3$ and the second one encrypts with $IV \,||\, CTR_4$, and so on
  - Good for applications with high throughput requirements