University of Technology
الجامعة التكنولوجية

Computer Science Department
قسم علوم الحاسوب

Data Security 2
أمنية البيانات ٢

Professor: D. Sukaina Hashim
أ.د. سكينة هاشم

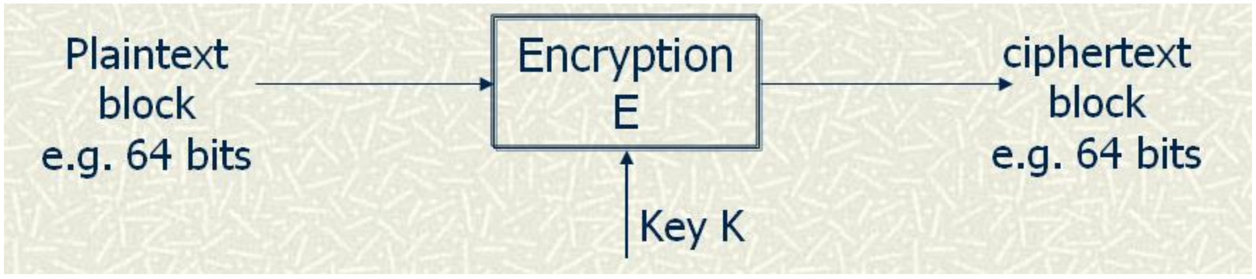
Lectuer:Enas Tariq
أ.م. إيناس طارق

edu.iq

Chapter Four
Data Encryption Standard (DES)

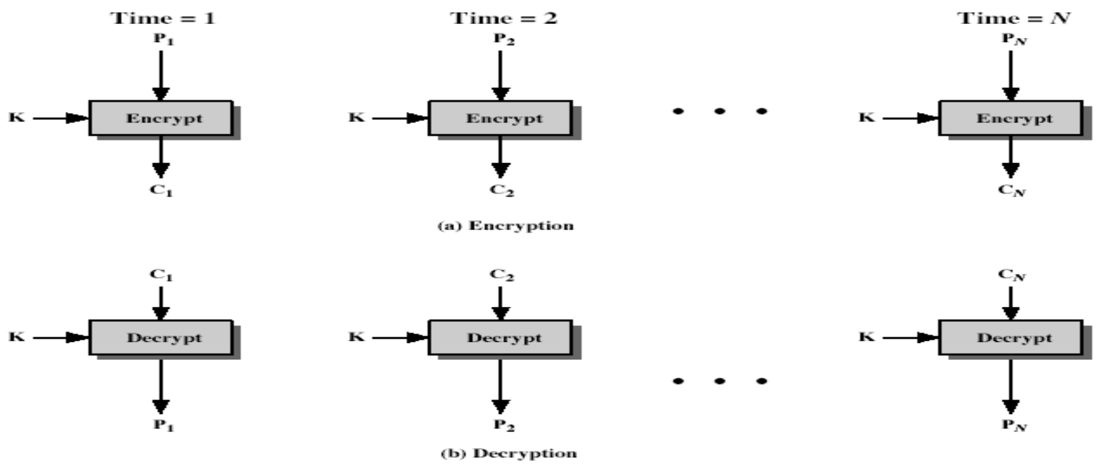
Block Cipher

Block Cipher - An encryption scheme that "the clear text is broken up into blocks of fixed length, and encrypted one block at a time". Usually, a block cipher encrypts a block of clear text into a block of cipher text of the same length. In this case, a block cipher can be viewed as a simple substitute cipher with character size equal to the block size.

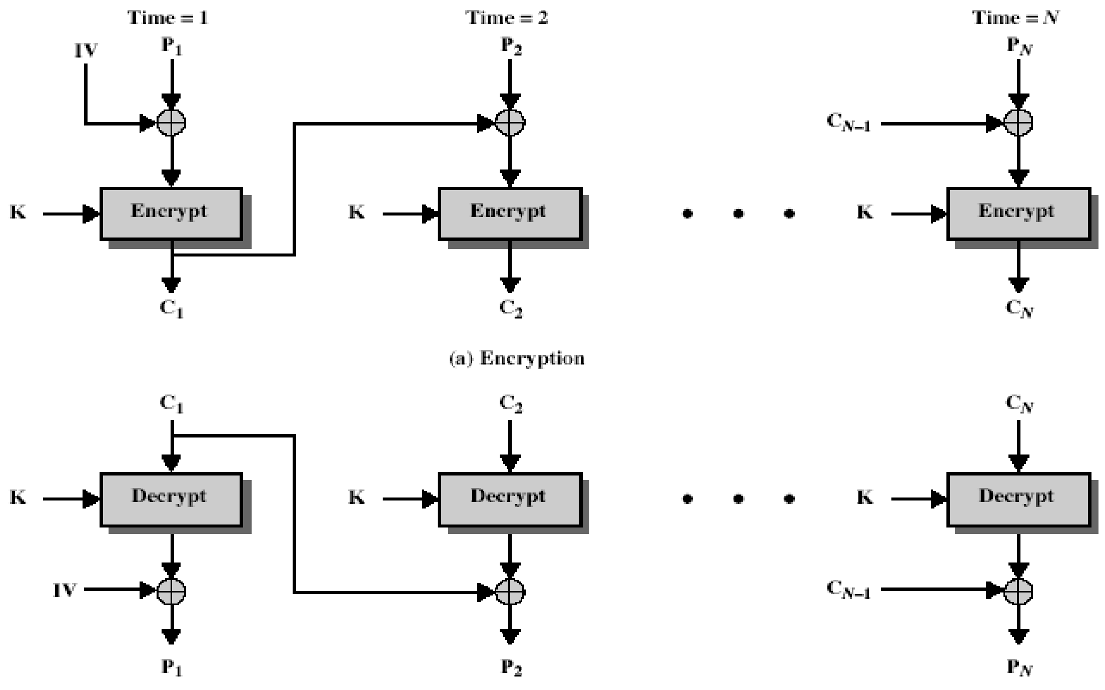


ECB Operation Mode - Blocks of clear text are encrypted independently. ECB stands for Electronic Code Book. Main properties of this mode:

- Identical clear text blocks are encrypted to identical cipher text blocks.
- Re-ordering clear text blocks results in re-ordering cipher text blocks.
- An encryption error affects only the block where it occurs.

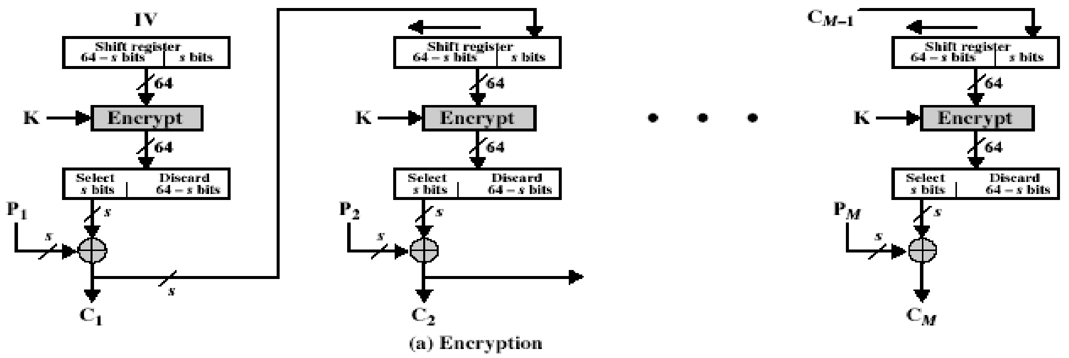


CBC Operation Mode - The previous cipher text block is XORed with the clear text block before applying the encryption mapping. Main properties of this mode:
An encryption error affects only the block where is occurs and one next block.

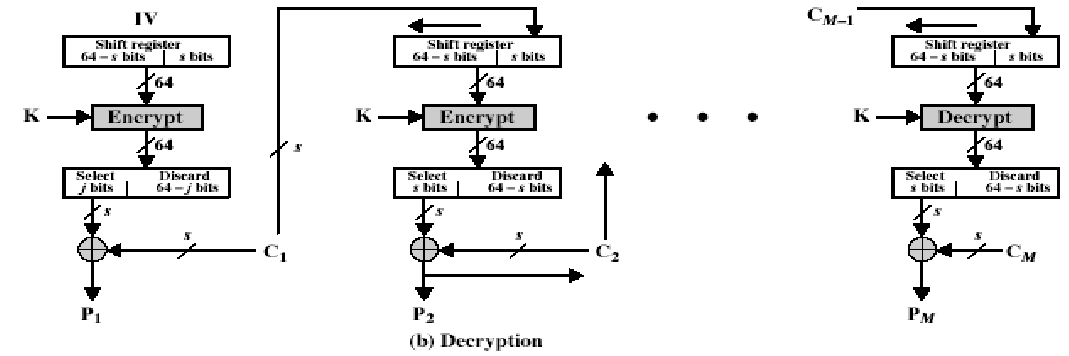


(a) Encryption
(b) Decryption

Cipher FeedBack (CFB) Message is treated as a stream of bits , Bitwise-added to the output of the block cipher , Result is feedback for next stage (hence name)

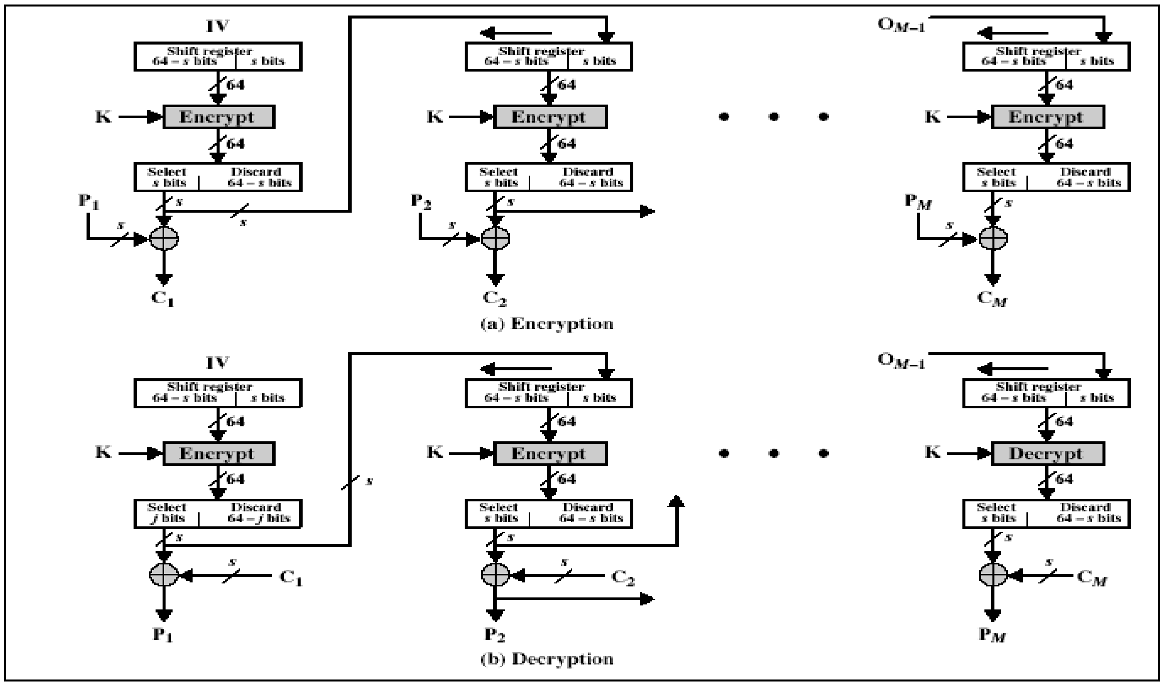


(a) Encryption



(b) Decryption

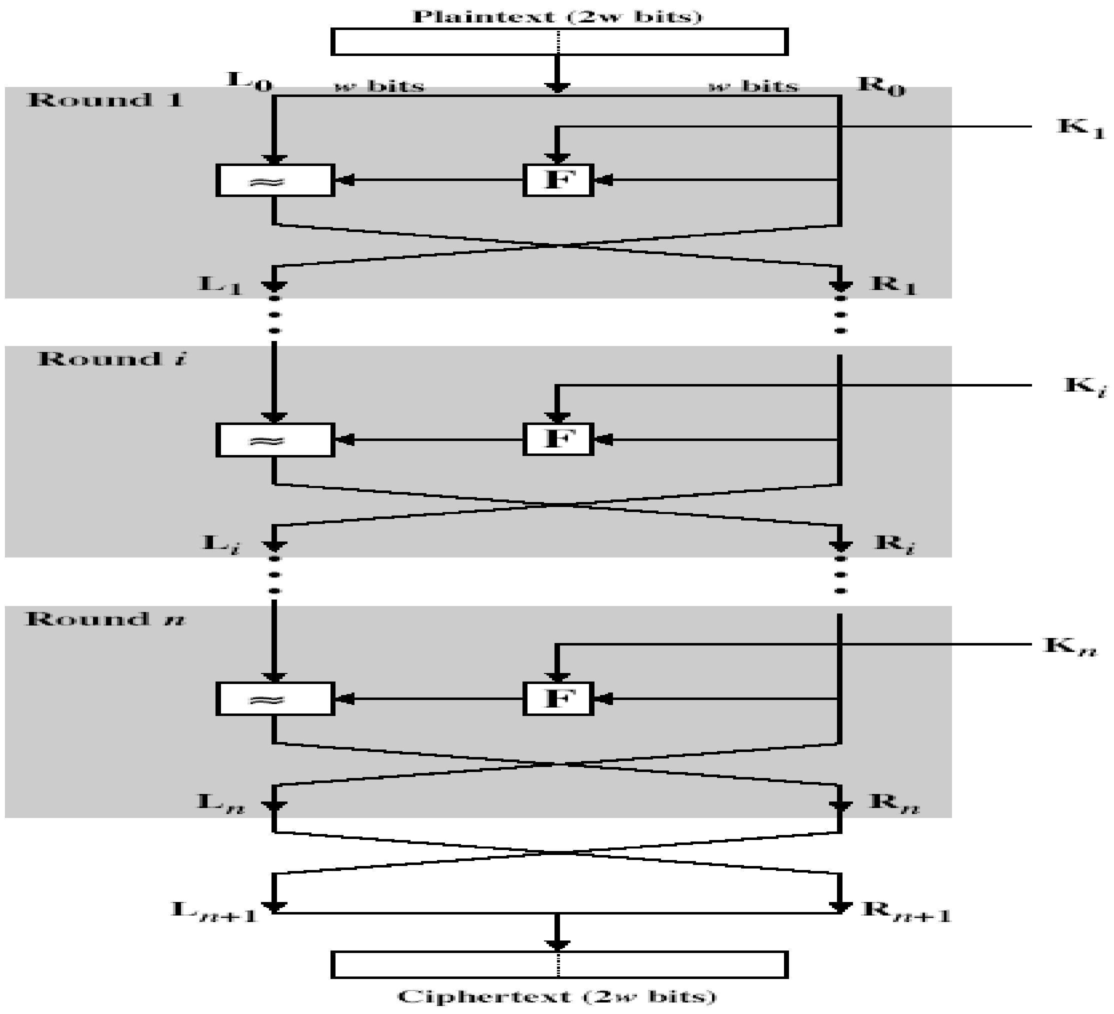
Output Feedback Mode (OFM)- The block cipher is used as a stream cipher, it produces the random key stream.



Product Cipher - An encryption scheme that "uses multiple ciphers in which the cipher text of one cipher is used as the clear text of the next cipher". Usually, substitution ciphers and transposition ciphers are used alternatively to construct a product cipher.

Iterated Block Cipher - A block cipher that "iterates a fixed number of times of another block cipher, called round function, with a different key, called round key, for each iteration".

Feistel Cipher - An iterate block cipher that uses the following algorithm:



DES Cipher - A 16-round Feistel cipher with block size of 64 bits. DES stands for Data Encryption Standard.

Data Encryption Standard (DES)

The Data Encryption Standard (DES), known as the Data Encryption Algorithm (DEA) by ANSI and the DEA-1 by the ISO, has been most widely used block cipher in world, especially in financial industry. It encrypts 64-bit data, and uses 56-bit key with 16 48-bit sub-keys.

Description of DES

DES is a block cipher; it encrypts data in 64-bit blocks. A 64-bit block of plaintext goes in one end of the algorithm and a 64-bit block of ciphertext comes out the other end. DES is a symmetric algorithm: The same algorithm and key are used for both encryption and decryption (except for minor differences in the key schedule).

The key length is 56 bits. (The key is usually expressed as a 64-bit number, but every eighth bit is used for parity checking and is ignored. These parity bits are the least- significant bits of the key bytes.) The key can be any 56-bit number and can be changed at any time. All security rests within the key.

At its simplest level, the algorithm is nothing more than a combination of the two basic techniques of encryption: confusion and diffusion. The fundamental building block of DES is a single combination of these techniques (a substitution followed by a permutation) on the text, based on the key. This is known as a round. DES has 16 rounds; it applies the same combination of techniques on the plaintext block 16 times (see Figure 1).

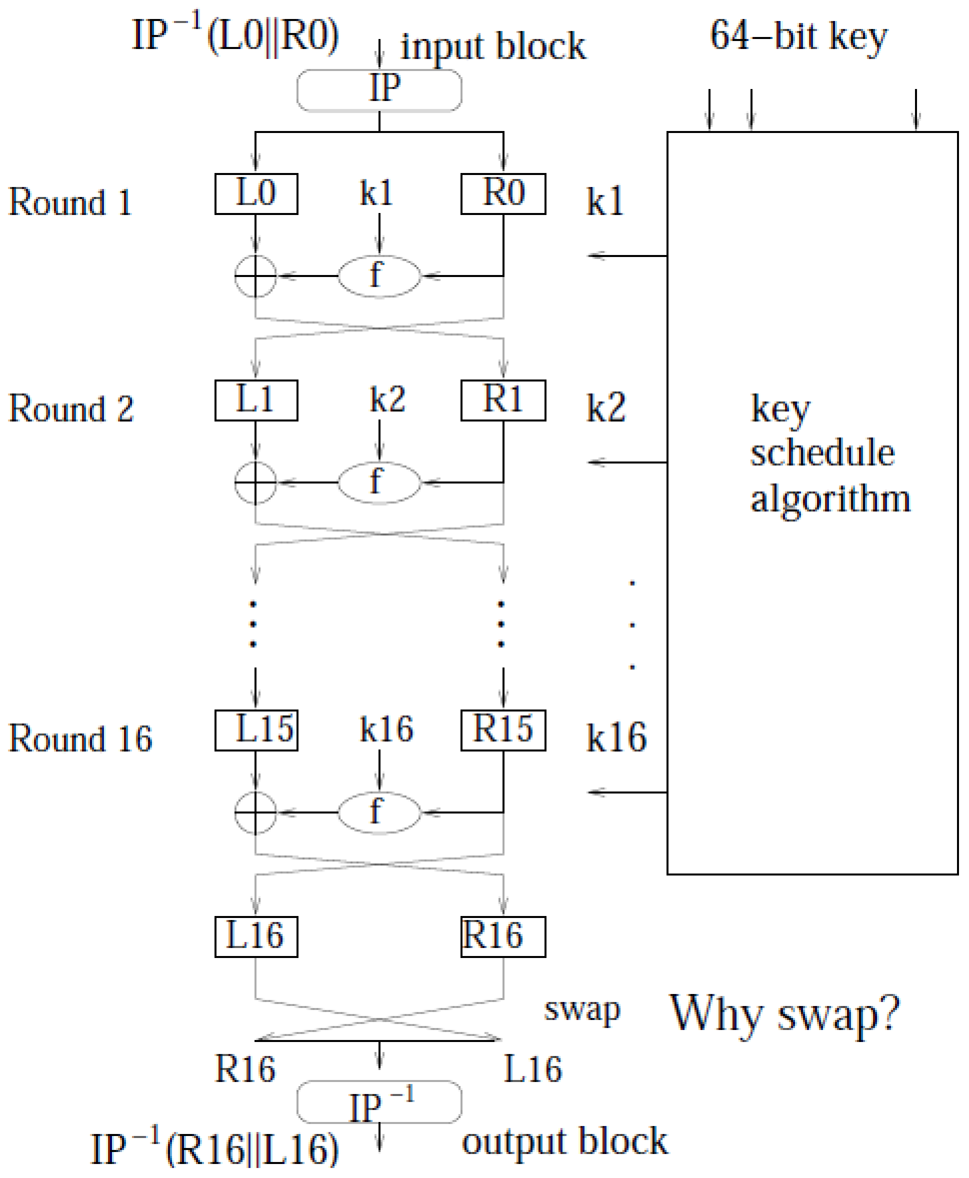

IP^{-1}(L0||R0)
input block
IP
64–bit key
Round 1
L0
k1
R0
k1
f
Round 2
L1
k2
R1
k2
f
key schedule algorithm
Round 16
L15
k16
R15
k16
f
L16
R16
swap
Why swap?
R16
L16
IP^{-1}
IP^{-1}(R16||L16)
output block

Figure 1 DES

Outline of the Algorithm

The basic process in enciphering a 64-bit data block using the DES consists of:

- an initial permutation (IP)
- 16 rounds of a complex key dependent calculation f
- final permutation, being the inverse of IP

In each round (see Figure 2,3,4,5), the key bits are shifted, and then 48 bits are selected from the 56 bits of the key. The right half of the data is expanded to 48 bits via an expansion permutation, combined with 48 bits of a shifted and permuted key via an XOR, sent through 8 S-boxes producing 32 new bits, and permuted again. These four operations make up Function f. The output of Function f is then combined with the left half via another XOR. The result of these operations becomes the new right half; the old right half becomes the new left half.
If Bi is the result of the ith iteration, Li and Ri are the left and right halves of Bi, Ki is the 48-bit key for round i, and f is the function that does all the substituting and permuting and XORing with the key, then a round looks like:

$$Li = R_{j-1}$$
$$Ri = L_{i-1} \text{ Xor } f(R_{i-1}, K_i)$$

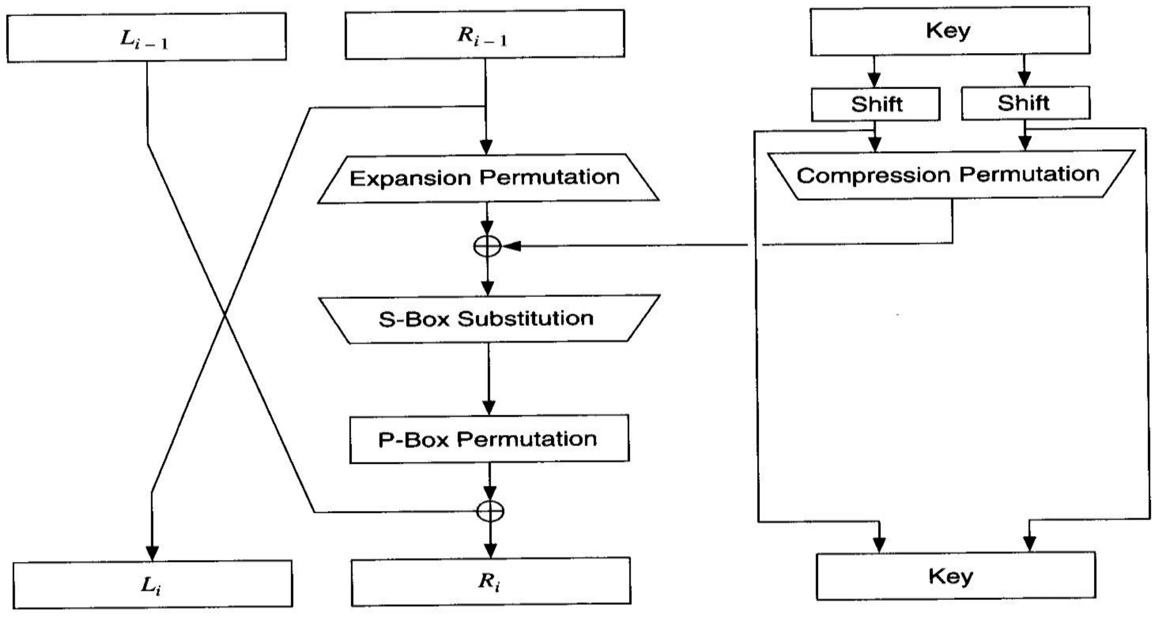


Figure 2 One round of DES

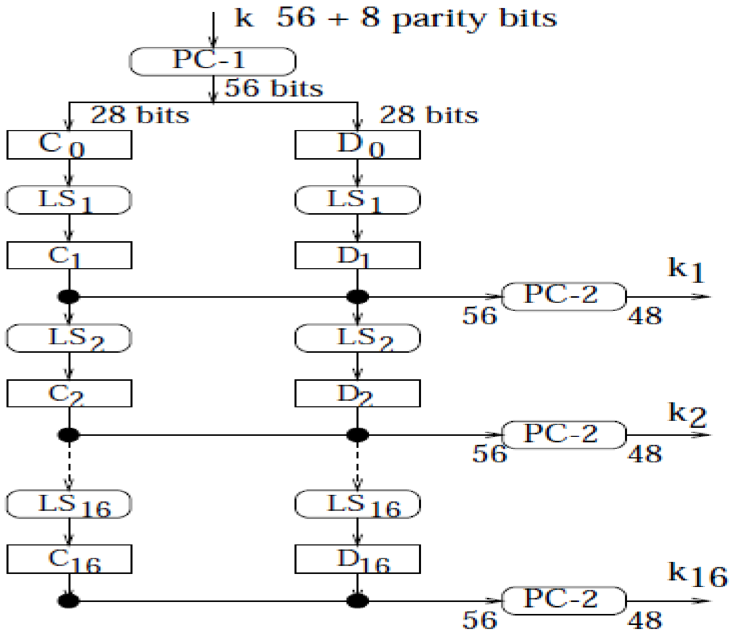


Figure 3. 16th key generation

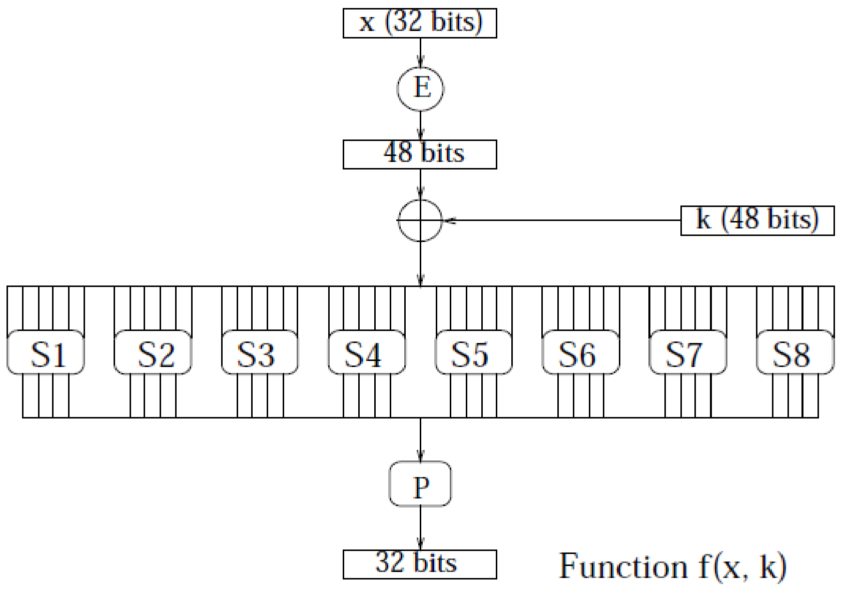
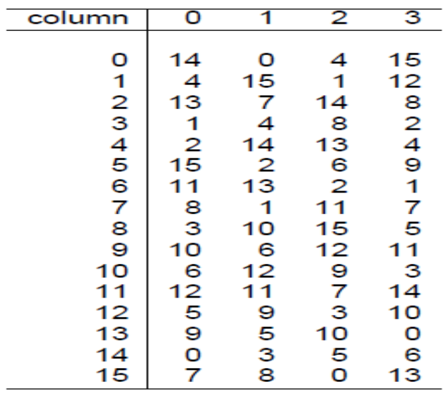


Figure 4. f-function



column	0	1	2	3
0	14	0	4	15
1	4	15	1	12
2	13	7	14	8
3	1	4	8	2
4	2	14	13	4
5	15	2	6	9
6	11	13	2	1
7	8	1	11	7
8	3	10	15	5
9	10	6	12	11
10	6	12	9	3
11	12	11	7	14
12	5	9	3	10
13	9	5	10	0
14	0	3	5	6
15	7	8	0	13

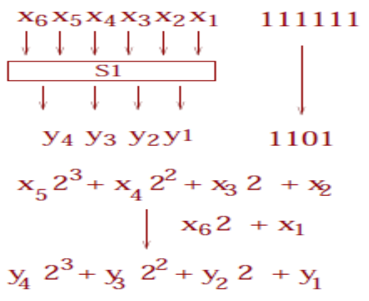

$x_6 x_5 x_4 x_3 x_2 x_1$
111111
S1
$y_4\ y_3\ y_2 y_1$
1101
$x_5 2^3 + x_4 2^2 + x_3 2 + x_2$
$x_6 2 + x_1$
$y_4\ 2^3 + y_3\ 2^2 + y_2\ 2 + y_1$

Figure 5. S-Boxes in F-function

The Initial Permutation

The initial permutation occurs before round 1; it transposes the input block as described in Table 1. This table, like all the other tables in this lecture, should be read left to right, top to bottom. For example, the initial permutation moves bit 58 of the plaintext to bit position 1, bit 50 to bit position 2, bit 42 to bit position 3, and so forth.

The initial permutation and the corresponding final permutation do not improve DES's security, just make DES more complex.

Example:

IP(675a6967 5e5a6b5a) = (ffb2194d 004df6fb)

Note that all numbers are written in hexadecimal as a "short-form" version of the binary actually used, since 1 Hex digit = 4 Binary bits. The digit mapping is:

0=0000 1=0001 2=0010 3=0011 4=0100 5=0101 6=0110 7=0111

8=1000 9=1001 a=1010 b=1011 c=1100 d=1101 e=1110 f=1111

The Key Transformation

Initially, the 64-bit DES key is reduced to a 56-bit key by ignoring every eighth bit. Let us call this operation PC1. This is described in Table 2.

PC2 is the operation which reduces the 56-bits key to a 48-bits subkey for each of the 16 rounds of DES. These subkeys, Ki, are determined in the following manner. PC1 splits the key bits into 2 halves (C and D), each 28-bits. The halves C and D are circularly shifted left by either one or two bits, depending on the round. This shift is given in Table 3. After being shifted, 48 out of the 56 bits are selected. This is done by an operation called compression permutation, it permutes the order of the bits as well as selects a subsets of bits. Table 4 defines the compression permutation.

Example:

```
 keyinit(5b5a5767, 6a56676e)
 PC1(Key)   C=00ffd820,     D=ffec9370
 KeyRnd01  C1=01ffb040,   D1=ffd926f0,   PC2(C,D)=(38 09 1b 26 2f 3a 27 0f)
 KeyRnd02  C2=03ff6080,   D2=ffb24df0,   PC2(C,D)=(28 09 19 32 1d 32 1f 2f)
 KeyRnd03  C3=0ffd8200,   D3=fec937f0,   PC2(C,D)=(39 05 29 32 3f 2b 27 0b)
 KeyRnd04  C4=3ff60800,   D4=fb24dff0,   PC2(C,D)=(29 2f 0d 10 19 2f 1d 3f)
 KeyRnd05  C5=ffd82000,   D5=ec937ff0,   PC2(C,D)=(03 25 1d 13 1f 3b 37 2a)
 KeyRnd06  C6=ff608030,   D6=b24dfff0,   PC2(C,D)=(1b 35 05 19 3b 0d 35
3b)
 KeyRnd07  C7=fd8200f0,   D7=c937ffe0,   PC2(C,D)=(03 3c 07 09 13 3f 39 3e)
 KeyRnd08  C8=f60803f0,   D8=24dfffb0,   PC2(C,D)=(06 34 26 1b 3f 1d 37 38)
 KeyRnd09  C9=ec1007f0,   D9=49bfff60,   PC2(C,D)=(07 34 2a 09 37 3f 38 3c)
 KeyRnd10  C10=b0401ff0,  D10=26fffd90,  PC2(C,D)=(06 33 26 0c 3e 15 3f 38)
 KeyRnd11  C11=c1007fe0, D11=9bfff640,  PC2(C,D)=(06 02 33 0d 26 1f 28 3f)
 KeyRnd12  C12=0401ffb0,  D12=6fffd920,  PC2(C,D)=(14 16 30 2c 3d 37 3a 34)
 KeyRnd13  C13=1007fec0, D13=bfff6490,  PC2(C,D)=(30 0a 36 24 2e 12 2f 3f)
 KeyRnd14  C14=401ffb00,  D14=fffd9260,  PC2(C,D)=(34 0a 38 27 2d 3f 2a 17)
 KeyRnd15  C15=007fec10, D15=fff649b0,  PC2(C,D)=(38 1b 18 22 1d 32 1f 37)
 KeyRnd16  C16=00ffd820,  D16=ffec9370, PC2(C,D)=(38 0b 08 2e 3d 2f 0e 17)
```

Table 1

Initial Permutation

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.

Table 2
Key Permutation

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

Table 3
Number of Key Bits Shifted per Round

Round	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Number	1	1	2	2	2	2	2	2	1	2	2	2	2	2	2	1

Table 4
Compression Permutation

14,	17,	11,	24,	1,	5,	3,	28,	15,	6,	21,	10,
23,	19,	12,	4,	26,	8,	16,	7,	27,	20,	13,	2,
41,	52,	31,	37,	47,	55,	30,	40,	51,	45,	33,	48,
44,	49,	39,	56,	34,	53,	46,	42,	50,	36,	29,	32.

The Expansion Permutation

This operation expands the right half of the data, R_i, from 32 bits to 48 bits. Because this operation changes the order of the bits as well as repeating certain bits, it is known as an expansion permutation. This operation has two purposes: It makes the right half the same size as the key for the XOR operation and it provides a longer result that can be compressed during the substitution operation.
However, neither of those is its main cryptographic purpose.
For each 4-bit input block, the first and fourth bits each represent two bits of the output block, while the second and third bits each represent one bit of the output

block. Table 5 shows which output positions correspond to which input positions. For example, the bit in position 3 of the input block moves to position 4 of the lutput block, and the bit in position 21 of the input block moves to positions 30 and 32 of the output block.

Table 5

Expansion Permutation

32,	1,	2,	3,	4,	5,	4,	5,	6,	7,	8,	9,
8.	9,	10,	11,	12,	13,	12,	13,	14,	15,	16,	17,
16,	17,	18,	19,	20,	21,	20,	21,	22,	23,	24,	25,
24,	25,	26,	27,	28,	29,	28,	29,	30,	31,	32,	1

The S-Box Substitution

After the compressed key is XORed with the expanded block, the 48-bit result moves to a substitution operation. The substitutions are performed by eight substitution boxes, or S-boxes.

Each S-box has a 6-bit input and a 4-bit output, and there are eight different S-boxes. The 48 bits are divided into eight 6-bit sub-blocks. Each separate block is operated on by a separate S-box: The first block is operated on by S-box 1, the second block is operated on by S-box 2, and so on.

Each S-box is a table of 4 rows and 16 columns. Each entry in the box is a 4-bit number. The 6 input bits of the S-box specify under which row and column number to look for the output. Table 6 shows all eight S-boxes.

The input bits specify an entry in the S-box in a very particular manner. Consider an S-box input of 6 bits, labeled bi, b2, b3, b, b, and b6. Bits b, and b6are combined to form a 2-bit number, from 0 to 3, which corresponds to a row in the table. The middle 4 bits, b2 through b5, are combined to form a 4-bit number, from 0 to 15, which corresponds to a column in the table.

For example, assume that the input to the sixth S-box (i.e., bits 31 through 36 of the XOR function) is 110011. The first and last bits combine to form 11, which corrspends to row 3 of the sixth S-box. The middle 4 bits combine to form 1001, which corresponds to the column 9 of the same S-box. The entry under row 3, column 9 of S-box 6 is 14. (Remember to count rows and columns from 0 and not from 1.) The value 1110 is substituted for 110011.

The S-box substitution is the critical step in DES. The algorithm's other operanons are linear and easy to analyze. The S-boxes are nonlinear and, more than any.hing else, give DES its security.

The result of this substitution phase is eight 4-bit blocks which are recombined into a single 32-bit block. This block moves to the next step: the P-box permutation.

Table 6 -Boxes

S-box 1:

14,	4,	13,	1,	2,	15,	11,	8,	3,	10,	6,	12,	5,	9,	0,	7,
0,	15,	7,	4,	14,	2,	13,	1,	10,	6,	12,	11,	9,	5,	3,	8,
4,	1,	14,	8,	13,	6,	2,	11,	15,	12,	9,	7,	3,	10,	5,	0,
15,	12,	8,	2,	4,	9,	1,	7,	5,	11,	3,	14,	10,	0,	6,	13,

S-box 2:

15,	1,	8,	14,	6,	11,	3,	4,	9,	7,	2,	13,	12,	0,	5,	10,
3,	13,	4,	7,	15,	2,	8,	14,	12,	0,	1,	10,	6,	9,	11,	5,
0,	14,	7,	11,	10,	4,	13,	1,	5,	8,	12,	6,	9,	3,	2,	15,
13,	8,	10,	1,	3,	15,	4,	2,	11,	6,	7,	12,	0,	5,	14,	9,

S-box 3:

10,	0,	9,	14,	6,	3,	15,	5,	1,	13,	12,	7,	11,	4,	2,	8,
13,	7,	0,	9,	3,	4,	6,	10,	2,	8,	5,	14,	12,	11,	15,	1,
13,	6,	4,	9,	8,	15,	3,	0,	11,	1,	2,	12,	5,	10,	14,	7,
1,	10,	13,	0,	6,	9,	8,	7,	4,	15,	14,	3,	11,	5,	2,	12,

S-box 4:

7,	13,	14,	3,	0,	6,	9,	10,	1,	2,	8,	5,	11,	12,	4,	15,
13,	8,	11,	5,	6,	15,	0,	3,	4,	7,	2,	12,	1,	10,	14,	9,
10,	6,	9,	0,	12,	11,	7,	13,	15,	1,	3,	14,	5,	2,	8,	4,
3,	15,	0,	6,	10,	1,	13,	8,	9,	4,	5,	11,	12,	7,	2,	14,

S-box 5:

2,	12,	4,	1,	7,	10,	11,	6,	8,	5,	3,	15,	13,	0,	14,	9,
14,	11,	2,	12,	4,	7,	13,	1,	5,	0,	15,	10,	3,	9,	8,	6,
41,	2,	1,	11,	10,	13,	7,	8,	15,	9,	12,	5,	6,	3,	0,	14,
11,	8,	12,	7,	1,	14,	2,	13,	6,	15,	0,	9,	10,	4,	5,	3,

S-box 6:

12,	1,	10,	15,	9,	2,	6,	8,	0,	13,	3,	4,	14,	7,	5,	11,
10,	15,	4,	2,	7,	12,	9,	5,	6,	1,	13,	14,	0,	11,	3,	8,
9,	14,	15,	5,	2,	8,	12,	3,	7,	0,	4,	10,	1,	13,	11,	6,
4,	3,	2,	12,	9,	5,	15,	10,	11,	14,	1,	7,	6,	0,	8,	13,

S-box 7:

4,	11,	2,	14,	15,	0,	8,	13,	3,	12,	9,	7,	5,	10,	6,	1,
13,	0,	11,	7,	4,	9,	1,	10,	14,	3,	5,	12,	2,	15,	8,	6,
1,	4,	11,	13,	12,	3,	7,	14,	10,	15,	6,	8,	0,	5,	9,	2,
6,	11,	13,	8,	1,	4,	10,	7,	9,	5,	0,	15,	14,	2,	3,	12,

S-box 8:

13,	2,	8,	4,	6,	15,	11,	1,	10,	9,	3,	14,	5,	0,	12,	7,
1,	15,	13,	8,	10,	3,	7,	4,	12,	5,	6,	11,	0,	14,	9,	2,
7,	11,	4,	1,	9,	12,	14,	2,	0,	6,	10,	13,	15,	3,	5,	8,
-2,	1,	14,	7,	4,	10,	8,	13,	15,	12,	9,	0,	3,	5,	6,	11

Example:

S(18 09 12 3d 11 17 38 39) = 5fd25e03

The P-Box Permutation

The 32-bit output of the S-box substitution is permuted according to a P-box. This permutation maps each input bit to an output position; no bits are used twice and no bits are ignored. Table 7 shows the position to which each bit moves. For example, bit 21 moves to bit 4. while bit 4 moves to bit 3 1.

Table 7

P-Box Permutation

16, 7, 20, 21, 29, 12, 28, 17, 1, 15, 23, 26, 5, 18, 31, 10,
2, 8, 24, 14, 32, 27, 3, 9, 19, 13, 30, 6, 22, 11, 4, 25

Finally, the result of the P-box permutation is XORed with the left half of the initial 64-bit block. Then the left and right halves are switched and another round begins.

The Final Permutation

The final permutation is the inverse of the initial permutation and is described in Table 8. Note that the left and right halves are not exchanged after the last round of DES; instead the concatenated block $R_{16}L_{16}$ is used as the input to the final permutation. There's nothing going on here; exchanging the halves and shifting around the permutation would yield exactly the same result. This is so that the algorithm can be used to both encrypt and decrypt.

Table 8

Final Permutation

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.

Decrypting DES

After all the substitutions, permutations, XORs, and shifting around, you might think that the decryption algorithm is completely different and just as confusing as the encryption algorithm. On the contrary, the various operations were chosen to produce a very useful property: The same algorithm works for both encryption and decryption.

With DES it is possible to use the same function to encrypt or decrypt a block. The only difference is that the keys must be used in the reverse order. That is, if the encryption keys for each round are K1, K2, K3, . . . , K16, then the decryption keys are K16, K15, K14, . . . , K1,. The algorithm that generates the key used for each round is circular as well. The key shift is a right shift and the number of positions shifted is 0, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1.

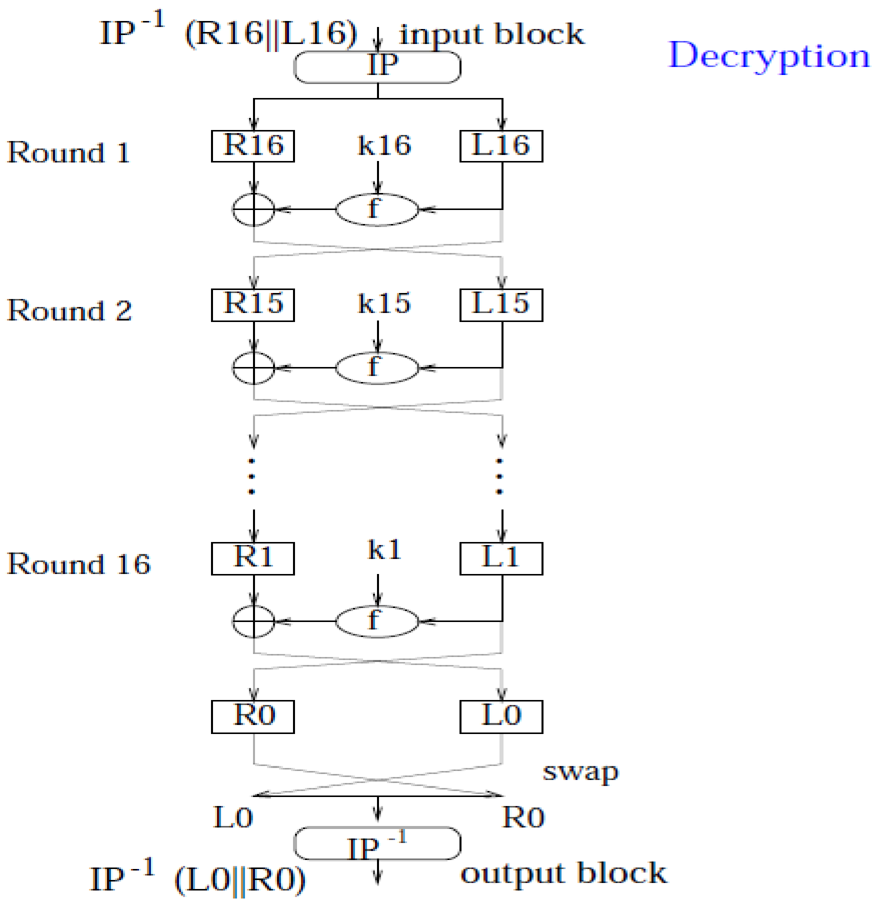
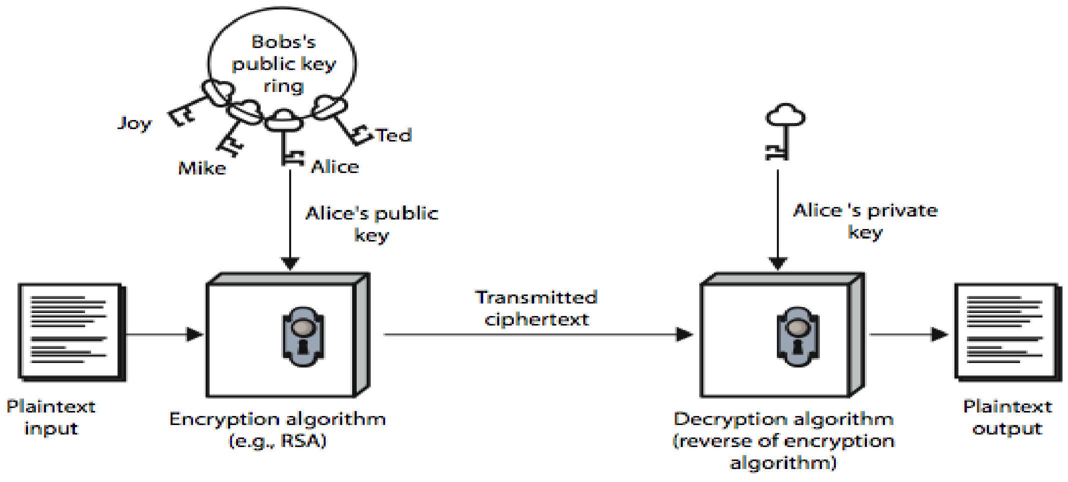


Figure 6. DES Decryption

Exponential Cipher

Public-Key Cryptography

- public-key/two-key/asymmetric cryptography involves the use of two keys:
  - a public-key, which may be known by anybody, and can be used to encrypt messages, and verify signatures
  - a private-key, known only to the recipient, used to decrypt messages, and sign (create) signatures
- is asymmetric because
  - those who encrypt messages or verify signatures cannot decrypt messages or create signatures

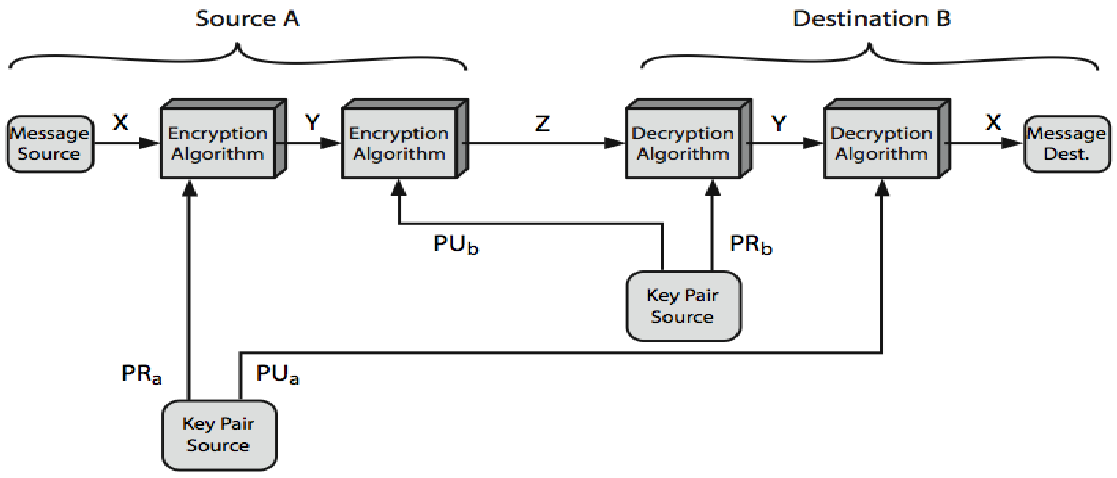


(a) Encryption

Public-Key Characteristics

- Public-Key algorithms rely on two keys where:
  - it is computationally infeasible to find decryption key knowing only algorithm & encryption key

- it is computationally easy to en/decrypt messages when the relevant (en/decrypt) key is known
- either of the two related keys can be used for encryption, with the other used for decryption (for some algorithms)



Public-Key Applications

- ➢ can classify uses into 3 categories:
  - encryption/decryption (provide secrecy)
  - digital signatures (provide authentication)
  - key exchange (of session keys)
- ➢ some algorithms are suitable for all uses, others are specific to one

Security of Public Key Schemes

- ➢ like private key schemes brute force exhaustive search attack is always theoretically possible
- ➢ but keys used are too large (>512bits)

Chapter Five

Exponentiation Ciphers

- We will consider two kinds of exponentiation ciphers developed by the following people:

$$\left\langle \begin{array}{l} \text{Pohlig and Hellman} \\ \text{Rivest, Shamir, and Adleman (RSA)} \end{array} \right.$$

- Both schemes encipher a message block $M \in [0, n-1]$ by computing the exponential $C = M^e \bmod n$,

- where e and n are the key to the enciphering transformation.
- M is restored by the same operation, but using a different exponent d for the key: $M = C^d \bmod n$.

- Enciphering and deciphering can be implemented using the fast exponentiation algorithm:

$$C = fast\_exp(M, e, n)$$

$$M = fast\_exp(C, d, n)$$

- Thm: Given e, d, M such that $ed \bmod \phi(n)$

  $= 1, M \in [0, n-1], gcd(M, n) = 1,$

  Then $(M^e \bmod n)^d \bmod n = M.$

- Note that by symmetry, enciphering and deciphering are commutative and mutual inverses; thus,

$(M^d \bmod n)^e \bmod n = M^{de} \bmod n = M$

- Given $\phi(n)$, it is easy to generate a pair (e, d) such that $ed \bmod \phi(n) = 1$. This is done by first choosing d relatively prime to $\phi(n)$, and then computing e as
- $e = inv(d, \phi(n))$
- Because e and d are symmetric, we could also pick e and compute $d = inv(e, \phi(n))$.
- Given e, it is easy to compute d (or vice versa) if $\phi(n)$ is known. But if e and n can be released without giving away $\phi(n)$ or d, then the deciphering transformation can be kept secret, while the enciphering transformation is made public.

- It is the ability to hide $\phi(n)$ that distinguishes the two schemes.

Pohlig-Hellman Scheme

- The modulus is chosen to be a large prime p.

To encipher:

$$C = M^e \bmod p$$

To decipher:

$$M = C^d \bmod p$$

- Because p is prime, $\phi(p) = p - 1$.

- Thus the scheme can only be used for conventional encryption, where e and d are both kept secret.
- Ex. Let $p = 11$, $\phi(p) = 10$. Choose $d = 7$ and compute $e = inv(7, 10) = 3$. Suppose $M = 5$. Then M is enciphered as:

$$C = M^{e} \bmod p = 5^{3} \bmod 11 = 4.$$

Similarly, C is deciphered as:

$$C^{d} \bmod p = 4^{7} \bmod 11 = 5 = M .$$

Security Concern

- A cryptanalyst may deduce p by observing the sizes of plaintext and ciphertext blocks.
- Under a known-plaintext attack, a cryptanalyst can compute e (and thereby d) given a pair (M, C):
- $e = \log M^{C}$
- Pohlig and Hellman show that if $(p - 1)$ has only small prime factors, it is possible to compute the logarithm in $O(\log 2p)$ time, which is unsatisfactory even for large values of p.
- They recommend picking $p = 2p' + 1$, where p' is also a large prime.

RSA description and algorithm

RSA stands for Rivest, Shamir, and Adleman, they are the inventors of the RSA cryptosystem. RSA is one of the algorithms used in PKI (Public Key Infrastructure), asymmetric key encryption scheme. RSA is a block chiper, it encrypt message in blocks (block by block). The common size for the key length now is 1024 bits for P and Q, therefore N is 2048 bits, if the implementation (the library) of RSA is fast enough, we can double the key size.

Key Generation Algorithm

1. Generate two large random primes, *p* and *q*, of approximately equal size such that their product n = pq is of the required bit length, e.g. 1024 bits. [See note 1].
2. Compute n = pq and (φ) phi = (p-1)(q-1).
3. Choose an integer *e*, 1 < e < phi, such that gcd(e, phi) = 1. [See note 2].
4. Compute the secret exponent *d*, 1 < d < phi, such that ed ≡ 1 (mod phi). [See note 3].
5. The public key is (n, e) and the private key is (n, d). Keep all the values d, p, q and phi secret.

- n is known as the *modulus*.
- e is known as the *public exponent* or *encryption exponent* or just the *exponent*.
- d is known as the *secret exponent* or *decryption exponent*.

Encryption

Sender A does the following:-

1. Obtains the recipient B's public key (n, e).
2. Represents the plaintext message as a positive integer *m* [see note 4].
3. Computes the ciphertext $c = m^e \bmod n$.
4. Sends the ciphertext *c* to B.

Decryption

Recipient B does the following:-

1. Uses his private key (n, d) to compute $m = c^d \bmod n$.
2. Extracts the plaintext from the message representative *m*.

A very simple example of RSA encryption

This is an extremely simple example using numbers you can work out on a pocket calculator (those of you over the age of 35 45 can probably even do it by hand).

1. Select primes p=11, q=3.
2. n = pq = 11.3 = 33

   phi = (p-1)(q-1) = 10.2 = 20
3. Choose e=3

   Check gcd(e, p-1) = gcd(3, 10) = 1 (i.e. 3 and 10 have no common factors except 1),

   and check gcd(e, q-1) = gcd(3, 2) = 1

   therefore gcd(e, phi) = gcd(e, (p-1)(q-1)) = gcd(3, 20) = 1
4. Compute d such that $ed \equiv 1 \pmod{phi}$

   i.e. compute $d = e^{-1} \bmod phi = 3^{-1} \bmod 20$

   i.e. find a value for d such that phi divides (ed-1)

   i.e. find d such that 20 divides 3d-1.

Simple testing (d = 1, 2, ...) gives d = 7

Check: ed-1 = 3.7 - 1 = 20, which is divisible by phi.

5. Public key = (n, e) = (33, 3)

   Private key = (n, d) = (33, 7).

This is actually the smallest possible value for the modulus n for which the RSA algorithm works.

Now say we want to encrypt the message m = 7,

$c = m^e \bmod n = 7^3 \bmod 33 = 343 \bmod 33 = 13$.

Hence the ciphertext c = 13.

To check decryption we compute

$m' = c^d \bmod n = 13^7 \bmod 33 = 7$.

Note that we don't have to calculate the full value of 13 to the power 7 here. We can make use of the fact that

a = bc mod n = (b mod n).(c mod n) mod n

so we can break down a potentially large number into its components and combine the results of easier, smaller calculations to calculate the final value.

One way of calculating m' is as follows:-

$m' = 13^7 \bmod 33 = 13^{(3+3+1)} \bmod 33 = 13^3.13^3.13 \bmod 33$

$= (13^3 \bmod 33).(13^3 \bmod 33).(13 \bmod 33) \bmod 33$

= (2197 mod 33).(2197 mod 33).(13 mod 33) mod 33

= 19.19.13 mod 33 = 4693 mod 33

= 7.

Example 1:

Using small numbers for clarity, here are results of an example run:

enter prime p: 47

enter prime q: 71

n = p*q   = 3337

(p-1)*(q-1) = 3220

Guess a large value for public key e then we can work down from there.

enter trial public key e: 79

trying e = 79

Use private key d: 1019

Publish e: 79

and n: 3337

cipher = char^e (mod n) -----------------   char = cipher^d (mod n)

Example 2:

1) Generate two large prime numbers, p and q To make the example easy to follow I am going to use small numbers, but this is not secure. To find random primes, we start at a random number and go up ascending odd numbers until we find a prime. Lets have:

p = 7

q = 19

2) Let n = pq   ---------------------- n = 7 * 19 = 133

3) Let PHi = (p - 1)(q - 1)

PHi = (7 - 1)(19 - 1)   = 6 * 18   = 108

4) Choose a small number, e coprime to PHi

e coprime to PHi, means that the largest number that can exactly divide both e and m (their greatest common divisor, or GCD) is 1. Euclid's algorithm is used to find the GCD of two numbers, but the details are omitted here.

e = 2 => GCD(e, 108) = 2 (no)

e = 3 => GCD(e, 108) = 3 (no)

e = 4 => GCD(e, 108) = 4 (no)

e = 5 => GCD(e, 108) = 1 (yes!)

5) Find d, such that de % m = 1

This is equivalent to finding d which satisfies de = 1 + nPHi where n is any integer. We can rewrite this as d = (1 + nPHi) / e. Now we work through values of n until an integer solution for e is found:

n = 0 => d = 1 / 5 (no)

n = 1 => d = 109 / 5 (no)

n = 2 => d = 217 / 5 (no)

n = 3 => d = 325 / 5

= 65 (yes!)

To do this with big numbers, a more sophisticated algorithm called extended Euclid must be used.

Public Key

n = 133 && e = 5

Secret Key

n = 133 && d =65

Encryption

The message must be a number less than the smaller of p and q. However, at this point we don't know p or q, so in practice a lower bound on p and q must be published. This can be somewhat below their true value and so isn't a major security concern. For this example, lets use the message "6".

C = Pe % n

= 65 % 133

= 7776 % 133

= 62

Decryption

This works very much like encryption, but involves a larger exponentiation, which is broken down into several steps.

P = Cd % n

= 6265 % 133

= 62 * 6264 % 133

= 62 * (622)32 % 133

= 62 * 384432 % 133

= 62 * (3844 % 133)32 % 133

= 62 * 12032 % 133

We now repeat the sequence of operations that reduced 6265 to 12032 to reduce the exponent down to 1.

= 62 * 3616 % 133

= 62 * 998 % 133

= 62 * 924 % 133

= 62 * 852 % 133

= 62 * 43 % 133

= 2666 % 133

= 6

And that matches the plaintext we put in at the beginning, so the algorithm worked!

Example 3:

A very simple example of RSA encryption

This is an extremely simple example using numbers you can work out on a pocket calculator (those of you over the age of 35 45 can probably even do it by hand).

Select primes p=11, q=3.

n = pq = 11.3 = 33

phi = (p-1)(q-1) = 10.2 = 20

Choose e=3

Check gcd(e, p-1) = gcd(3, 10) = 1 (i.e. 3 and 10 have no common factors except 1),

and check gcd(e, q-1) = gcd(3, 2) = 1

therefore gcd(e, phi) = gcd(e, (p-1)(q-1)) = gcd(3, 20) = 1

Compute d such that ed ≡ 1 (mod phi)

i.e. compute d = e-1 mod phi = 3-1 mod 20 i.e. find a value for d such that phi divides (ed-1)

i.e. find d such that 20 divides 3d-1.

Simple testing (d = 1, 2, ...) gives d = 7

Check: ed-1 = 3.7 - 1 = 20, which is divisible by phi.

Public key = (n, e) = (33, 3)

Private key = (n, d) = (33, 7).

This is actually the smallest possible value for the modulus n for which the RSA algorithm works.

Now say we want to encrypt the message m = 7,

c = me mod n = 73 mod 33 = 343 mod 33 = 13.

Hence the ciphertext c = 13.

To check decryption we compute

m' = cd mod n = 137 mod 33 = 7.

Security Concern

- Because $\phi(n)$ cannot be determined without knowing the prime factors p and q, it is possible to keep d secret even if e and n are made public.
- Thus the RSA scheme can be used for public-key encryption, where the enciphering transformation is made public and the deciphering transformation is kept secret.

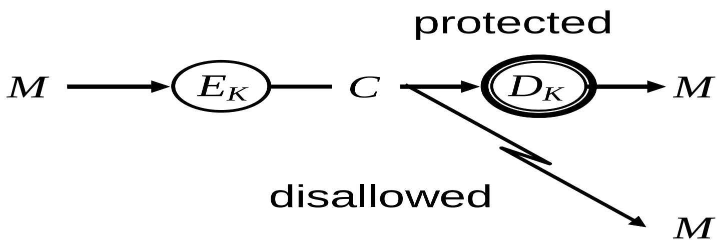
- The security of the system depends on the difficulty of factoring n into p and q. The fastest known factoring algorithm takes about the same number of steps required for solving the discrete logarithm problem.

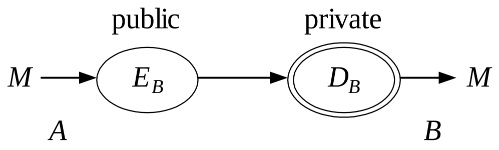
More About Euler's Theorem

- Recall that for Pohlig-Hellman and RSA schemes to work, we must have $M < n$ and $gcd(M, n) = 1$.
- For Pohlig-Hallman scheme, this is for sure since n is prime. But how about RSA? Since n equals $p \times q$, it is possible that M is a multiple of p or a multiple of q (but not both, of course).
- We want to show that even if M is a multiple of p or q, the RSA scheme still works.

Secrecy and Authenticity

- In a public-key system, secrecy and authenticity are both provided.
- Secrecy Suppose user A wishes to send a message M to another user B. If A knows B's public transformation E_B, A can transmit M to B in secrecy by sending the ciphertext $C = E_B(M)$.
- On receipt, B deciphers C using B's private transformation D_B, getting
- $D_B(C) = D_B(E_B(M)) = M$



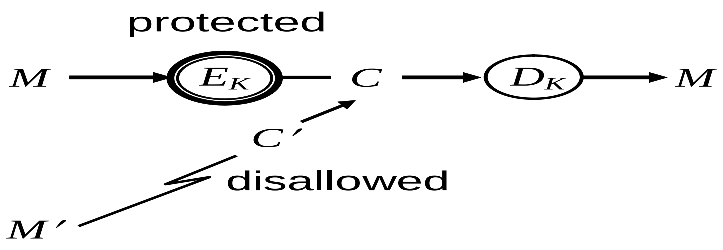


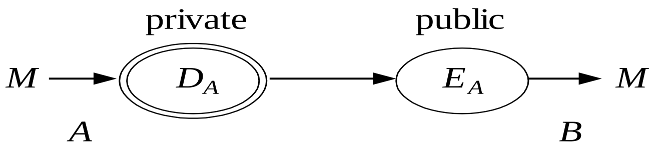
- The scheme does not provide authenticity because any user with access to *B*'s public transformation could substitute another message *M'* for *M* by replacing *C* with *C'* = *EB*(*M'*).

- AuthenticityFor authenticity, *M* must be transformed by *A*'s own private transformation D_A. *A* sends $C = D_A(M)$ to *B*.

- On receipt, *B* uses *A*'s public transformation *EA* to compute

- $E_A(C) = E_A(D_A(M)) = M$.





- Authenticity is provided because only A can apply the transformation D_A.
- Secrecy is not provided because any user with access to A's public transformation can recover M.

Both Secrecy and Authenticity

- To use a public-key system for both secrecy and authenticity:
  - the ciphertext space must be equivalent to the plaintext space so that E_A and D_A can operate on both plaintext and ciphertext messages.
  - Both E_A and D_A must be mutual inverses so that $E_A(D_A(M)) = D_A(E_A(M)) = M$.

- Suppose A wishes to send a message M to B. A sends to B the ciphertext

$$C = E_B(D_A(M)) .$$

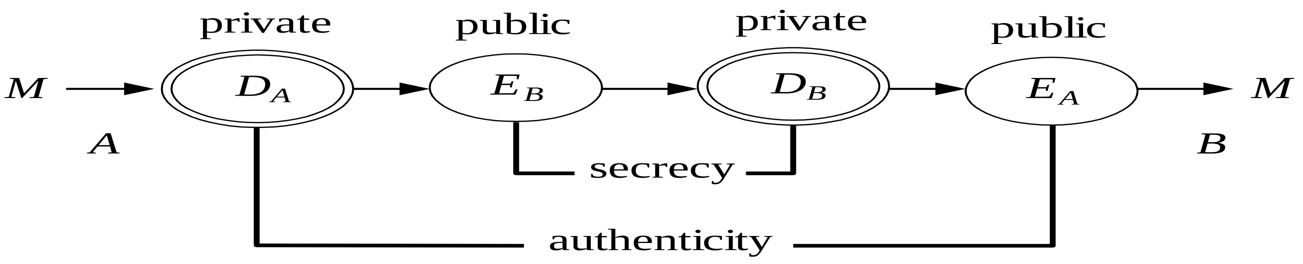
- On receipt, B deciphers C by

$E_A(D_B(C))$

$= E_A(D_B(E_B(D_A(M))))$

$= E_A(D_A(M))$

$= M$.



Merkle-Hellman Knapsacks

Knapsack problem: How to find the optimal way to pack a knapsack enclosing the maximum number of objects.

Numerically:

target sum: 17

set S {4, 7, 1,12,10}

one solution set: {4, 1,12 }=17

V {1, 0, 1, 1, 0 }

N-P Complete

- basically: an NP complete problem has a deterministic exponential time solution. For example, 2^n
- This allows us to control the brute force attack. Ie, make time to break very large!

Merkle-Hellman Knapsacks

Plaintext	1	0	1	0	0	1	
Knapsack	1	2	5	9	20	43	
Ciphertext	1		5			43	
Target Sum							49
Plaintext	0	1	1	0	1	0	
Knapsack	1	2	5	9	20	43	
Ciphertext		2	5		20		
Target Sum							27

Figure 3-5 Knapsack for Encryption

Example

Plain text	10011	11010	01011	00000
Knapsack	1 6 8 15 24	1 6 8 15 24	1 6 8 15 24	1 6 8 15 24
Cipher text	1 + 15 + 24 = 40	1 + 6 + 15 = 22	6 + 15 + 24 = 45	0 = 0

MH Knapsack

- Each element is larger than the previous
- Example $a_1, a_2, a_3, a_4, a_5, \dots a_{k-1}, a_k$
- sums between a_k and a_{k+1} must contain a_k
    - superincreasing knapsack-each integer is > $\sum a_k$
    - also called simple knapsack
- S=[1,4,11,17,38,73] is a superincreasing knapsack

Diffie-Hellman

Diffie-Hellman found a way to break the superincreasing sequence of integers. *$w * x \bmod n$.* If w and n are relatively prime, w will have a multiplicitive inverse. *$w * w^{-1} = 1 \bmod n$. $(w*q)\, w^{-1} = q$*

Table 3-1 $3 * x \bmod 5$

x	$3 * x$	$3 * x \bmod 5$
1	3	3
2	6	1
3	9	4
4	12	2
5	15	0
6	18	3
7	21	1

Table 3-2 $3 * x \bmod 6$

x	$3 * x$	$3 * x \bmod 6$
1	3	3
2	6	0
3	9	3
4	12	0
5	15	3
6	18	0
7	21	3

Why so important?

- This allows us to create a public knapsack (Hard) which can be based on a secret simple knapsack and a secret *w*, and *n*.

Example

Create a Superincreasing (or simple) knapsack

Sequence	Sum so far	Next term
[1,		
[1,	1	2
[1,2,	$1 + 2 = 3$	4
[1, 2, 4,	$1 + 2 + 4 = 7$	9
[1, 2, 4, 9,	$1 + 2 + 4 + 9 = 16$	19

S=[1,2,4,9] *m*=5

Example

S=[1,2,4,9] *m*=5

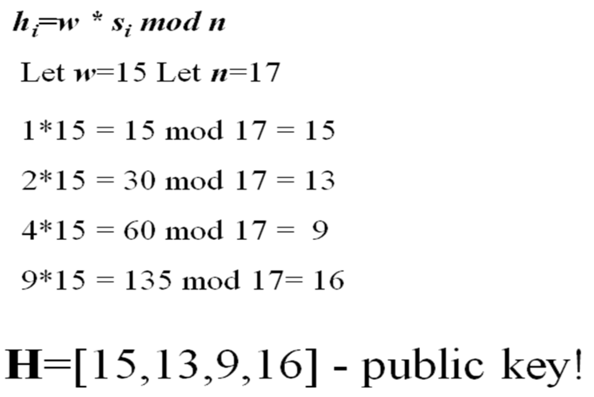
Choose a multiplier *w*, and modulus *n* *n* should be larger than the largest integer in your knapsack

Hint: Choose modulus (*n*) to be a prime number.

Generate the Hard knapsack by $h_i = w * s_i \bmod n$

H=[h_1, h_2, h_3, .. H_m]

S=[1,2,4,9]



$h_i = w * s_i \bmod n$

Let *w*=15 Let *n*=17

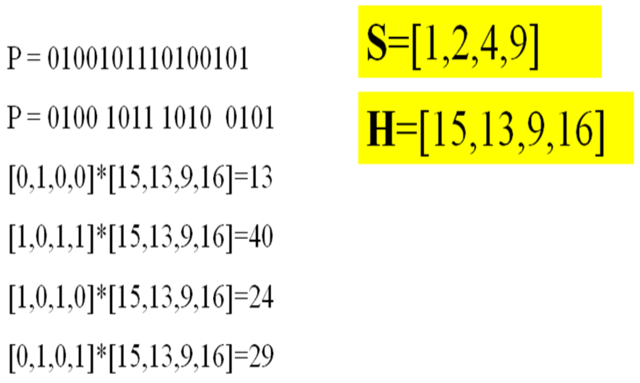
1*15 = 15 mod 17 = 15

2*15 = 30 mod 17 = 13

4*15 = 60 mod 17 = 9

9*15 = 135 mod 17= 16

H=[15,13,9,16] - public key!



P = 0100101110100101

P = 0100 1011 1010 0101

[0,1,0,0]*[15,13,9,16]=13

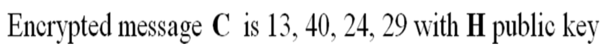
[1,0,1,1]*[15,13,9,16]=40

[1,0,1,0]*[15,13,9,16]=24

[0,1,0,1]*[15,13,9,16]=29

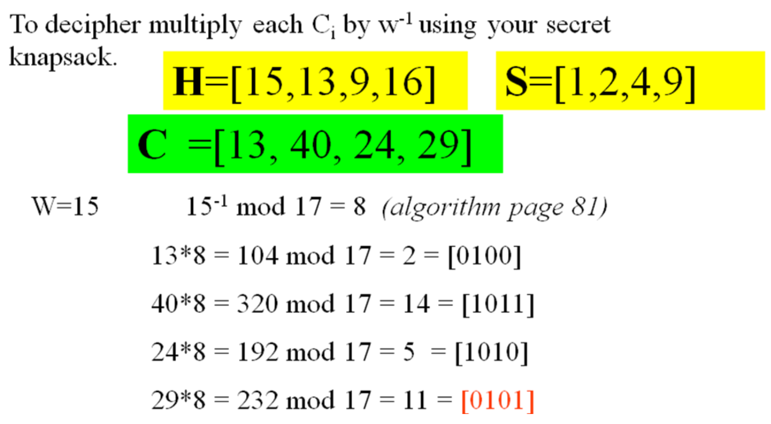
S=[1,2,4,9]

H=[15,13,9,16]



Encrypted message **C** is 13, 40, 24, 29 with **H** public key

Example (decipher)


To decipher multiply each C_i by w^{-1} using your secret knapsack.

H=[15,13,9,16] **S**=[1,2,4,9]

C =[13, 40, 24, 29]

W=15 15^{-1} mod 17 = 8 *(algorithm page 81)*

13*8 = 104 mod 17 = 2 = [0100]

40*8 = 320 mod 17 = 14 = [1011]

24*8 = 192 mod 17 = 5 = [1010]

29*8 = 232 mod 17 = 11 = [0101]

Example2:

First, a superincreasing sequence w is created

w = {2, 7, 11, 21, 42, 89, 180, 354}

This is the basis for a private key. From this, calculate the sum.

Then, choose a number q that is greater than the sum.

q = 881

Also, choose a number r that is in the range [1,q) and is coprime to q.

r = 588

The private key consists of q, w and r.

To calculate a public key, generate the sequence β by multiplying each element in w by r mod q

β = {295, 592, 301, 14, 28, 353, 120, 236}

because

2 * 588 mod 881 = 295

7 * 588 mod 881 = 592

11 * 588 mod 881 = 301

21 * 588 mod 881 = 14

42 * 588 mod 881 = 28

89 * 588 mod 881 = 353

180 * 588 mod 881 = 120

354 * 588 mod 881 = 236

The sequence β makes up the public key.

Say Alice wishes to encrypt "a". First, she must translate "a" to binary (in this case, using ASCII or UTF-8)

01100001

She multiplies each respective bit by the corresponding number in β

a = 01100001

0 * 295

+ 1 * 592

+ 1 * 301

+ 0 * 14

+ 0 * 28

+ 0 * 353

+ 0 * 120

+ 1 * 236

= 1129

She sends this to the recipient.

To decrypt, Bob multiplies 1129 by r -1 mod q (See Modular inverse)

1129 * 442 mod 881 = 372

Now Bob decomposes 372 by selecting the largest element in w which is less than or equal to 372. Then selecting the next largest element less than or equal to the difference, until the difference is 0 :

372 - 354 = 18

18 - 11 = 7

7 - 7 = 0

The elements we selected from our private key correspond to the 1 bits in the message

01100001

When translated back from binary, this "a" is the final decrypted message.

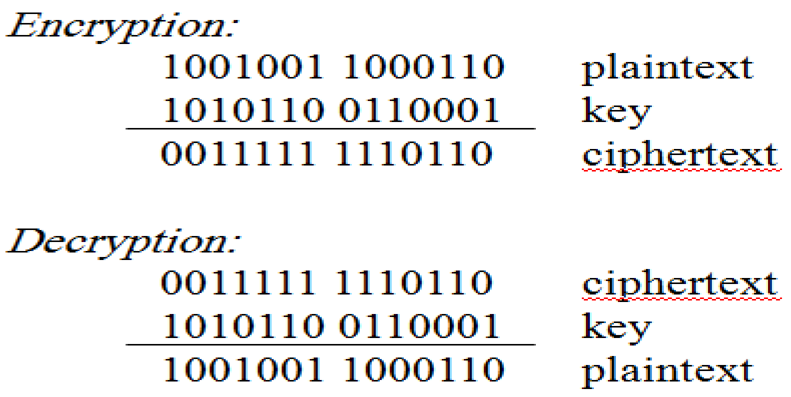
Chapter Six

Stream Cipher

One-Time Pad or Vernam Cipher

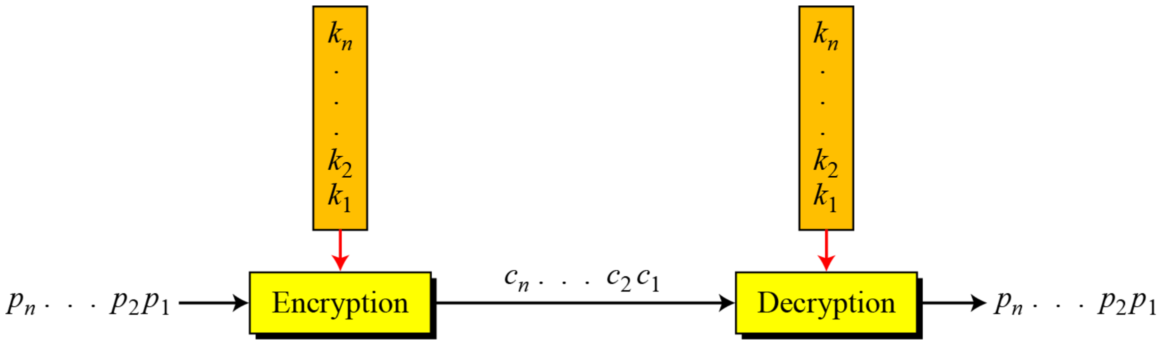
- The one-time pad, which is a provably secure cryptosystem, was developed by Gilbert Vernam in 1918.
- The message is represented as a binary string (a sequence of 0's and 1's using a coding mechanism such as ASCII coding.
- The key is a truly random sequence of 0's and 1's of the same length as the message.
- The encryption is done by adding the key to the message modulo 2, bit by bit. This process is often called *exclusive or*, and is denoted by *XOR*. The symbol ⊕ is used.

Example: Let the message be IF then its ASCII code be (1001001 1000110) and the key be (1010110 0110001). The ciphertext can be found exoring message and key bits



```
Encryption:
        1001001 1000110     plaintext
        1010110 0110001     key
      -------------------
        0011111 1110110     ciphertext

Decryption:
        0011111 1110110     ciphertext
        1010110 0110001     key
      -------------------
        1001001 1000110     plaintext
```



Basic Idea comes from One-Time-Pad cipher,

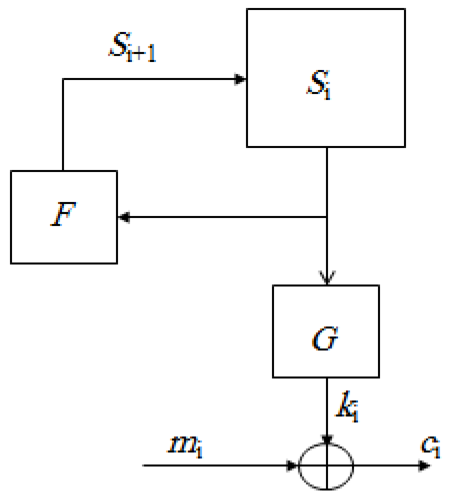
$$
\begin{aligned}
\text{Encryption} \quad &: \; c_i = m_i \oplus k_i \qquad i = 1,2,3,\ldots \\
m_i \quad &: \; \text{plain-text bits.} \\
k_i \quad &: \; \text{key (key-stream) bits} \\
c_i \quad &: \; \text{cipher-text bits.} \\
\text{Decryption} \quad &: \; m_i = c_i \oplus k_i \qquad i = 1,2,3,\ldots
\end{aligned}
$$

Drawback : Key-stream should be as long as plain-text. Key distribution & Management difficult.

Solution : Stream Ciphers (in which key-stream is generated in pseudo-random fashion from relatively short *secret key*.

Randomness : Closely related to *unpredictability*.

Pseudo-randomness :PR sequences appears random to a computationally bounded adversary. Stream Ciphers can be modeled as Finite-state machine.



S_i : state of the cipher at time $t = i$.
F : state function.
G : output function.

Initial state, output and state functions are controlled by the secret key.

1.**Synchronous Stream Ciphers**

- Key-stream is independent of plain and cipher-text.
- Both sender &receiver must be synchronized.
- Resynchronization can be needed.
- No error propagation.
- Active attacks can easily be detected.

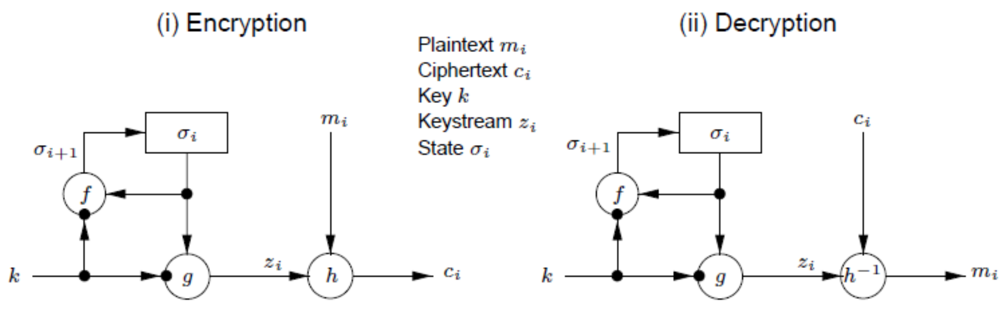
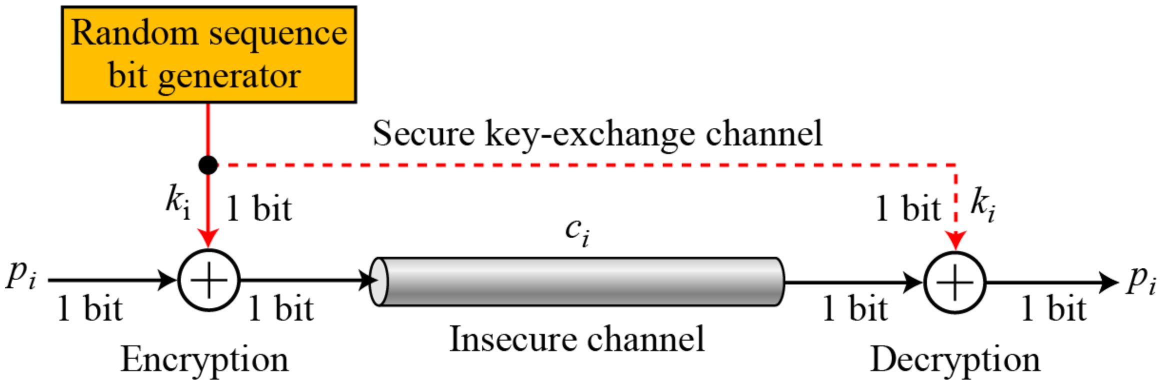


Figure 6.1: *General model of a synchronous stream cipher.*



2. Self-Synchronizing Stream Ciphers

- Key-stream is a function of fixed number *t* of cipher-text bits.
- Limited error propagation (up to *t* bits).
- Active attacks cannot be detected.
- At most *t* bits later, it resynchronizes itself when synchronization is lost.
- It helps to diffuse plain-text statistics.

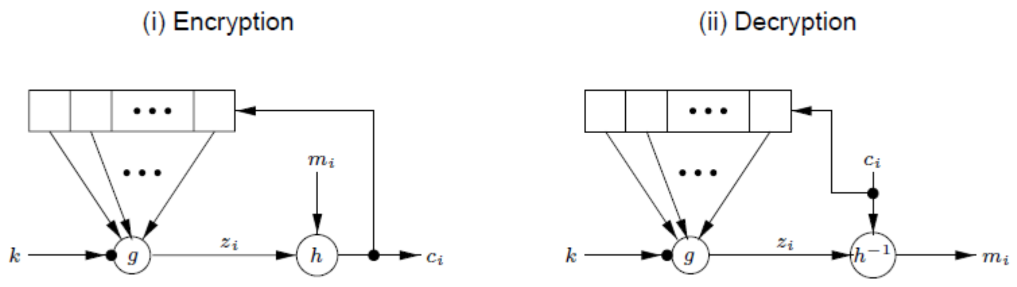


Figure 6.3: *General model of a self-synchronizing stream cipher.*

Linear feedback shift registers

Linear feedback shift registers (LFSRs) are used in many of the keystream generators that have been proposed in the literature. There are several reasons for this:

1. LFSRs are well-suited to hardware implementation;
2. they can produce sequences of large period;
3. they can produce sequences with good statistical properties; and
4. because of their structure, they can be readily analyzed using algebraic techniques.

Definition A *linear feedback shift register* (LFSR) of length L consists of L *stages* (or *delay elements*) numbered 0, 1,….., L − 1, each capable of storing one bit and having one input and one output; and a clock which controls the movement of data. During each unit of time the following operations are performed:

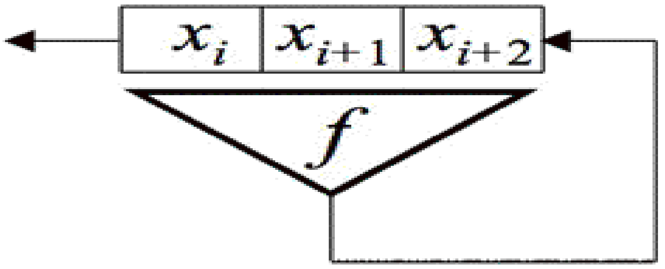
(i) the content of stage 0 is output and forms part of the *output sequence*;

(ii) the content of stage i is moved to stage i − 1 for each i, $1 \leq i \leq L - 1$; and

(iii) the new content of stage L − 1 is the *feedback bit* sj which is calculated by adding together modulo 2 the previous contents of a fixed subset of stages 0, 1,….., L – 1.

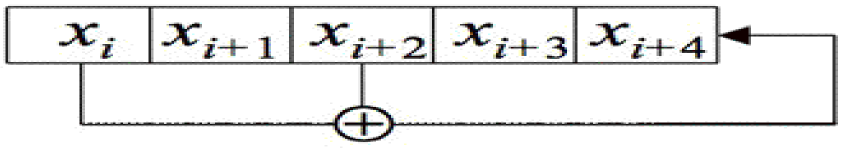
- Traditionally, stream ciphers were based on shift registers
  - Today, a wider variety of designs
- Shift register includes
  - A series of stages each holding one bit
  - A feedback function
- A linear feedback shift register (**LFSR**) has a linear feedback function

- Example (nonlinear) feedback function $f(x_i, x_{i+1}, x_{i+2}) = 1 \oplus x_i \oplus x_{i+2} \oplus x_{i+1}x_{i+2}$
- Example (nonlinear) shift register



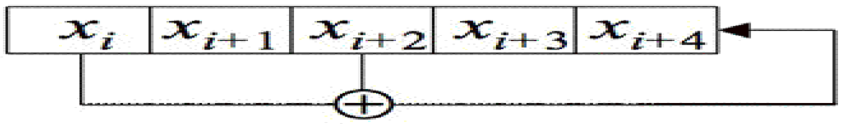
- First 3 bits are **initial fill**: (x_0, x_1, x_2)

Example of LFSR

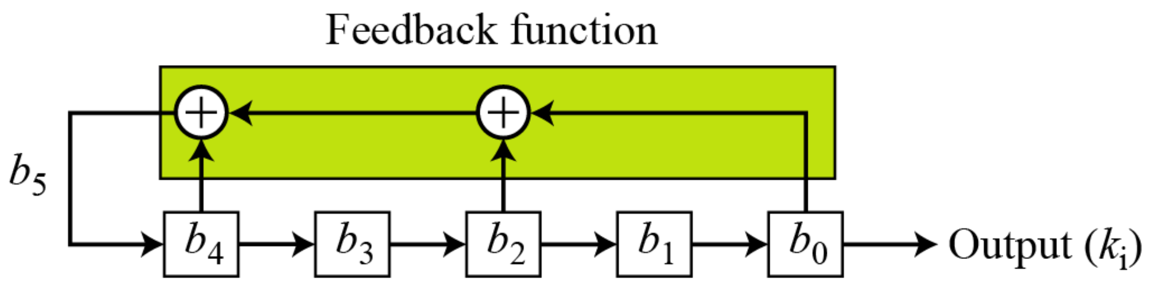


- Then $x_{i+5} = x_i \oplus x_{i+2}$ for all i
- If initial fill is $(x_0,x_1,x_2,x_3,x_4) = 01110$

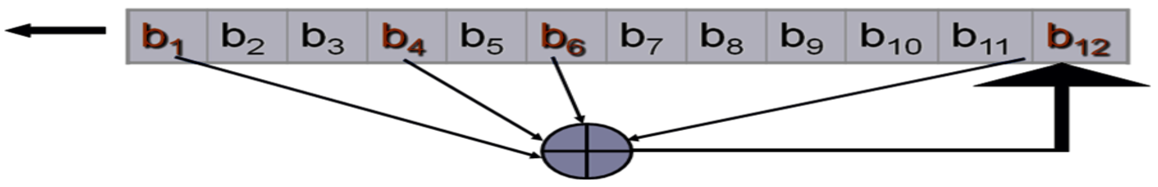
  then $(x_0,x_1,\ldots,x_{15},\ldots) = 0111010100001001\ldots$
- For LFSR

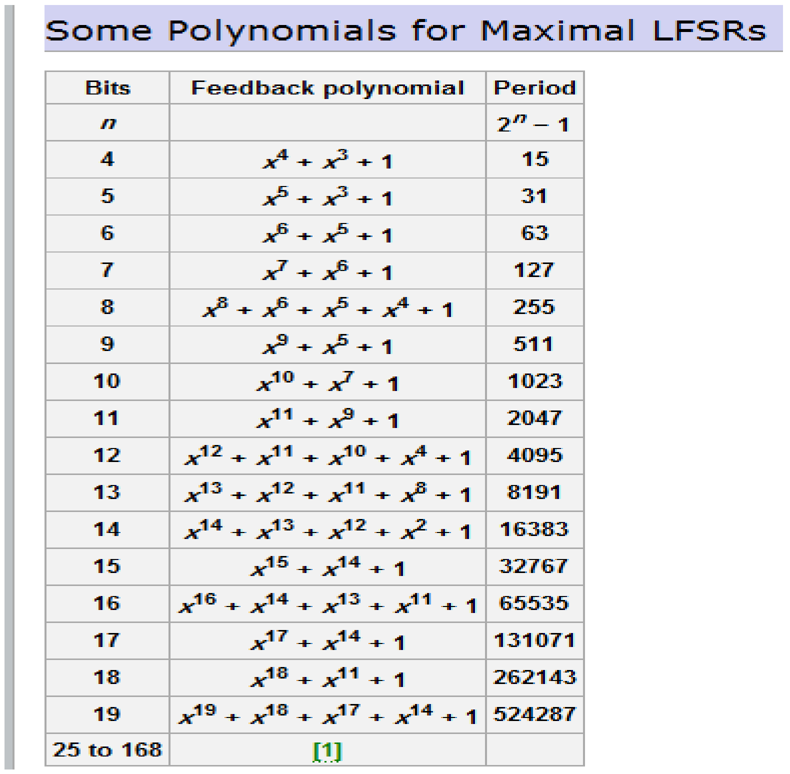


- We have $x_{i+5} = x_i \oplus x_{i+2}$ for all i
- Linear feedback functions often written in polynomial form: $x^5 + x^2 + 1$
- **Connection polynomial** of the LFSR



- Example :
  $x^{12}+x^6+x^4+x+1$ corresponds to LFSR of length 12

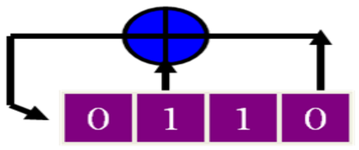




Some Polynomials for Maximal LFSRs

Bits	Feedback polynomial	Period
n		$2^n - 1$
4	$x^4 + x^3 + 1$	15
5	$x^5 + x^3 + 1$	31
6	$x^6 + x^5 + 1$	63
7	$x^7 + x^6 + 1$	127
8	$x^8 + x^6 + x^5 + x^4 + 1$	255
9	$x^9 + x^5 + 1$	511
10	$x^{10} + x^7 + 1$	1023
11	$x^{11} + x^9 + 1$	2047
12	$x^{12} + x^{11} + x^{10} + x^4 + 1$	4095
13	$x^{13} + x^{12} + x^{11} + x^8 + 1$	8191
14	$x^{14} + x^{13} + x^{12} + x^2 + 1$	16383
15	$x^{15} + x^{14} + 1$	32767
16	$x^{16} + x^{14} + x^{13} + x^{11} + 1$	65535
17	$x^{17} + x^{14} + 1$	131071
18	$x^{18} + x^{11} + 1$	262143
19	$x^{19} + x^{18} + x^{17} + x^{14} + 1$	524287
25 to 168	[1]	

Some Polynomials Don't



t	D_3	D_2	D_1	D_0
0	0	1	1	0
1	1	0	1	1
2	1	1	0	1
3	0	1	1	0
4	1	0	1	1
5	1	1	0	1
6	0	1	1	0
7	1	0	1	1

t	D_3	D_2	D_1	D_0
8	1	1	0	1
9	0	1	1	0
10	1	0	1	1
11	1	1	0	1
12	0	1	1	0
13	1	0	1	1
14	1	1	0	1
15	0	1	1	0

The polynomial $C(D)=1+D+D^3$ does not give a maximal period sequence.

Example

Create a linear feedback shift register with 4 cells in which $b_4 = b_1$ xor b_0. Show the value of output for 20 transitions (shifts) if the seed is $(0001)_2$.

Solution

LFSR for Example

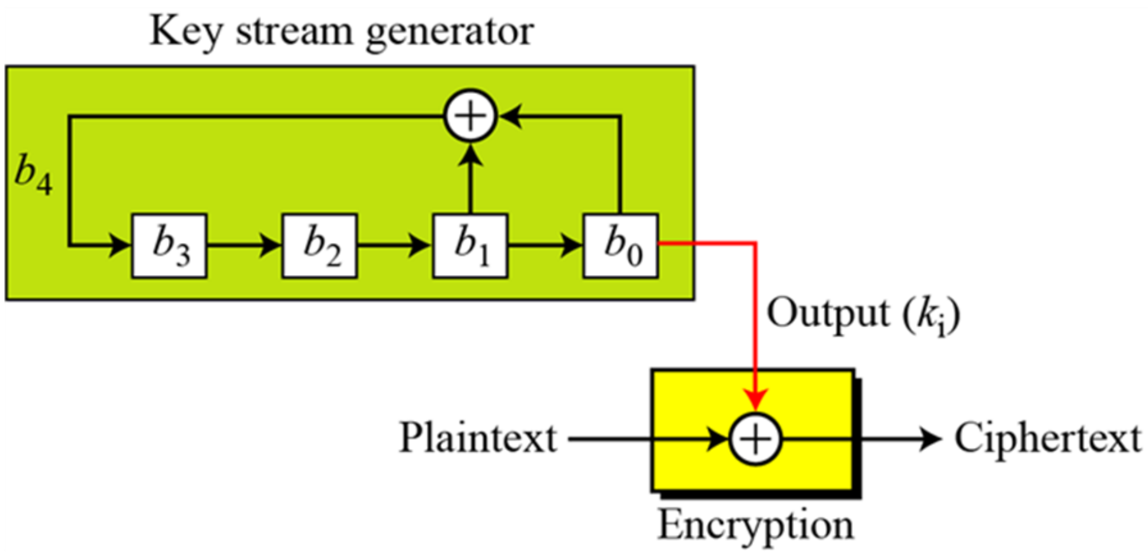


Table ***Cell values and key sequence for Example 5.19***

States	b_4	b_3	b_2	b_1	b_0	k_i
Initial	1	0	0	0	1	
1	0	1	0	0	0	1
2	0	0	1	0	0	0
3	1	0	0	1	0	0
4	1	1	0	0	1	0
5	0	1	1	0	0	1
6	1	0	1	1	0	0
7	0	1	0	1	1	0
8	1	0	1	0	1	1
9	1	1	0	1	0	1
10	1	1	1	0	1	0

Table Continued

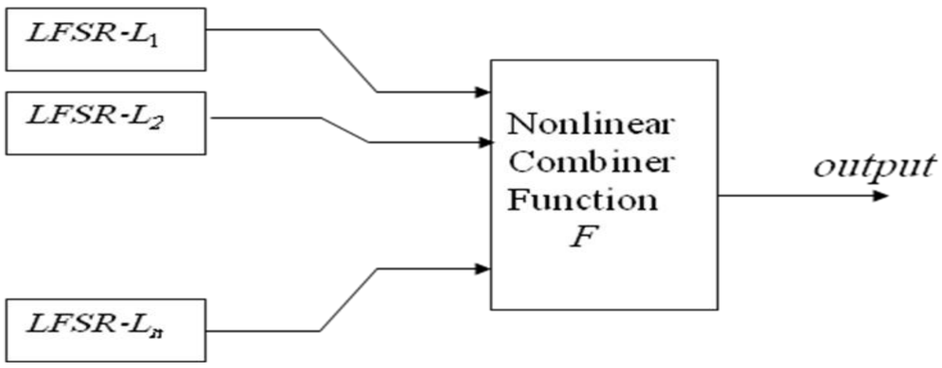
11	1	1	1	1	0	1
12	0	1	1	1	1	0
13	0	0	1	1	1	1
14	0	0	0	1	1	1
15	1	0	0	0	1	1
16	0	1	0	0	0	1
17	0	0	1	0	0	0
18	1	0	0	1	0	0
19	1	1	0	0	1	0
20	1	1	1	0	0	1

Note that the key stream is 100010011010111 10001…. This looks like a random sequence at first glance, but if we go through more transitions, we see that the sequence is periodic. It is a repetition of 15 bits as shown below:

100010011010111 **100010011010111** 100010011010111 **100010011010111** …

The key stream generated from a LFSR is a pseudorandom sequence in which the the sequence is repeated after *N* bits. The maximum period of an LFSR is to $2^m - 1$.

Nonlinear combination Generators



The Combiner Function should be, Balanced, Highly nonlinear, and Correlation Immune. Utilizing the *algebraic normal form* of the combiner function we can compute the linear complexity of the output sequence.

Example (Geffe Generator) :

$$F(x_1, x_2, x_3) = x_1x_2 \oplus x_2x_3 \oplus x_3$$

If the lengths of the LFSRs are relatively prime and all connection polynomials are primitive, then

$$L = L_1L_2 + L_2L_3 + L_3$$
$$T = (2^{L_1} - 1) \cdot (2^{L_2} - 1) \cdot (2^{L_3} - 1)$$

When we inspect the truth table of the combiner function we gain more insight about the security of Geffe generator.

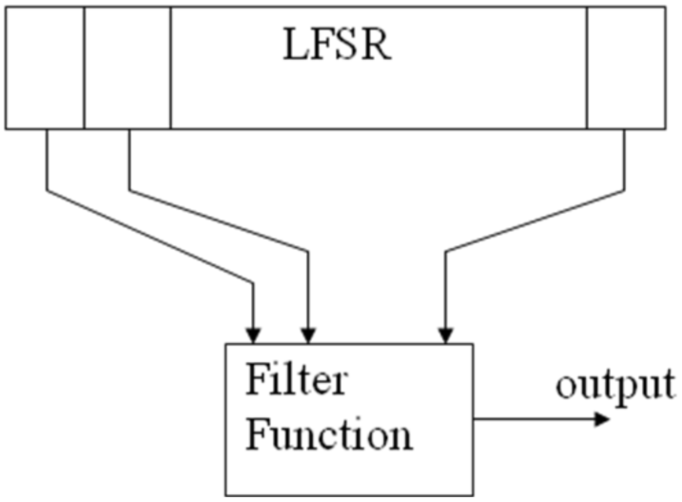
x_1	x_2	x_3	$z = F(x_1, x_2, x_3)$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

- The combiner function is balanced.
- However, the correlation probability,

$$P(z = x_1) = 3/4.$$

- Geffe generator is not secure.

Nonlinear Filter Generator



- Upper bound for linear complexity, m : nonlinear order of the filter function.

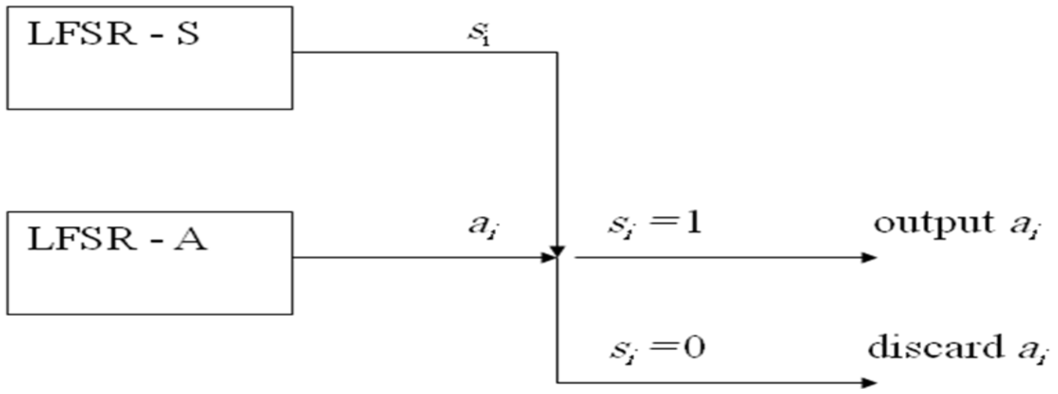
$$L_m = \sum_{i=1}^{m} \binom{L}{i}$$

- When L and m are big enough, the linear complexity will become large.

Clock-controlled Generators

- An LFSR can be clocked by the output of another LFSR.
- This introduces an irregularity in clocking of the first LFSR, hence increase the linear complexity of its output.

Example : Shrinking Generator



- Relatively new design.
- However, it is analyzed and it seems secure under certain circumstances.

$$\begin{aligned} &\textit{if } \gcd(L_s, L_A) = 1 \Rightarrow \\ &T = (2^{L_A} - 1) \cdot 2^{L_S - 1} \\ &L_A \cdot 2^{L_S - 2} < L < L_A \cdot 2^{L_S - 1} \end{aligned}$$

Randomness key Tests

The first tests for random numbers were published by M.G. Kendall and Bernard Babington Smith in the Journal of the Royal Statistical Society in 1938.They were built on statistical tools such as Pearson's chi-squared test that were developed to distinguish whether experimental phenomena matched their theoretical probabilities. Pearson developed his test originally by showing that a number of dice experiments by W.F.R. Weldon did not display "random" behavior.

Kendall and Smith's original four tests were hypothesis tests, which took as their null hypothesis the idea that each number in a given random sequence had an equal chance of occurring, and that various other patterns in the data should be also distributed equiprobably.

1. The frequency test, was very basic: checking to make sure that there were roughly the same number of 0s, 1s, 2s, 3s, etc.
2. The serial test, did the same thing but for sequences of two digits at a time (00, 01, 02, etc.), comparing their observed frequencies with their hypothetical predictions were they equally distributed.
3. The poker test, tested for certain sequences of five numbers at a time (aaaaa, aaaab, aaabb, etc.) based on hands in the game poker.
4. The gap test, looked at the distances between zeroes (00 would be a distance of 0, 030 would be a distance of 1, 02250 would be a distance of 3, etc.).

If a given sequence was able to pass all of these tests within a given degree of significance (generally 5%), then it was judged to be, in their words "locally random". Kendall and Smith differentiated "local randomness" from "true randomness" in that many sequences generated with truly random methods might not display "local randomness" to a given degree — very large sequences might contain many rows of a single digit. This might be "random" on the scale of the entire sequence, but in a smaller block it would not be "random" (it would not pass their tests), and would be useless for a number of statistical applications.

As random number sets became more and more common, more tests, of increasing sophistication were used. Some modern tests plot random digits as points on a three-dimensional plane, which can then be rotated to look for hidden patterns. In 1995, the statistician George Marsaglia created a set of tests known as the diehard tests, which he distributes with a CD-ROM of 5 billion pseudorandom numbers.

Pseudorandom number generators require tests as exclusive verifications for their "randomness," as they are decidedly not produced by "truly random" processes, but rather by deterministic algorithms. Over the history of random number generation, many sources of numbers thought to appear "random" under testing have later been discovered to be very non-random when subjected to certain types of tests. The notion of quasi-random numbers was developed to circumvent some of these problems, though pseudorandom number generators are still extensively used in many applications (even ones known to be extremely "non-random"), as they are "good enough" for most applications.