LAB MANUAL

INFORMATION AND NETWORK SECURITY

MIHIR SHUKLA
160470107054
VVPEC CE SEM-7

Contents

PRACTICAL-1

Aim: Introduction to Information And Network Security:

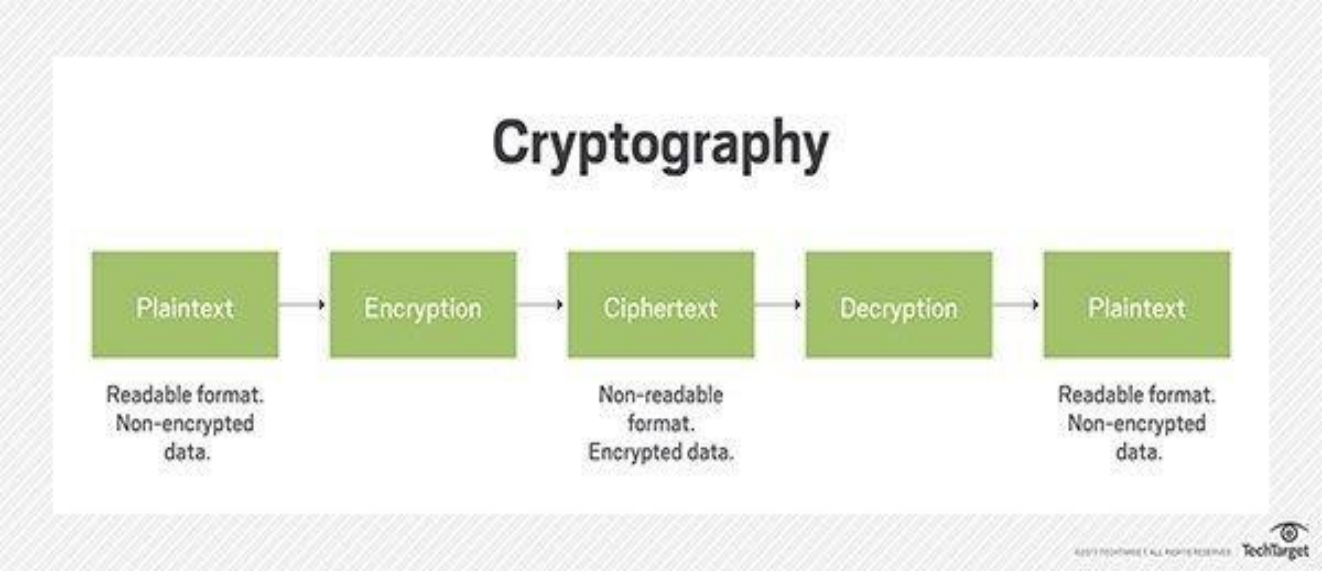
What is Cryptography?

Cryptography is a method of protecting information and communications through the use of secret key so that only those for whom the information is intended can read and process it. The pre-fix "crypt" means "hidden" or "vault" and the suffix "graphy" stands for "writing."

Cryptography is closely related to the disciplines of cryptology and cryptanalysis. It includes techniques such as microdots, merging words with images, and other ways to hide information in storage or transit. However, in today's computer-centric world, cryptography is most often associated with scrambling plaintext (ordinary text, sometimes referred to as cleartext) into ciphertext (a process called encryption), then back again (known as decryption). Individuals who practice this field are known as cryptographers.

Following objects of cryptography:

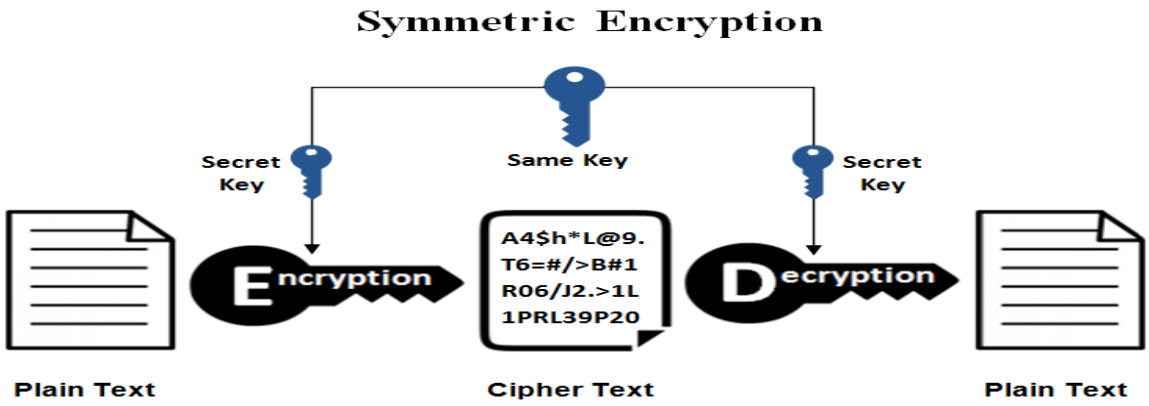
1. **Confidentiality**: the information cannot be understood by anyone for whom it was unintended.
2. **Integrity:** the information cannot be altered in storage or transit between sender and intended receiver without the alteration being detected.
3. **Authentication**: the sender and receiver can confirm each other's identity and the origin/destination of the information.



What is Symmetric & Asymmetric Cryptography?

Symmetric key cryptography –

It involves usage of one secret key along with encryption and decryption algorithms which help in securing the contents of the message. The strength of symmetric key cryptography depends upon the number of key bits. It is relatively faster than asymmetric key cryptography. There arises a key distribution problem as the key has to be transferred from the sender to receiver through a secure channel.



Pros of Symmetric key cryptography:

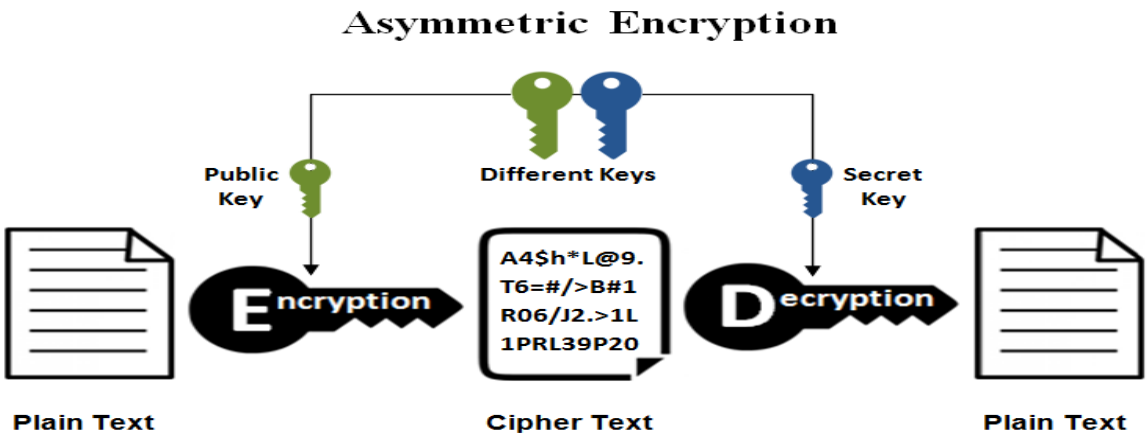
- A symmetric cryptography is faster.
- In Symmetric Cryptography, encrypted data can be transferred on the link even if there is a possibility that the data will be intercepted. Since there is no key transmitted with the data, the chances of data being decrypted are null.
- A symmetric cryptosystem uses password authentication to prove the receiver's identity.
- A system only which possesses the secret key can decrypt a message.

Cons of Symmetric key cryptography:

- Sharing the Key
- More Damage if Compromised

Assymetric key cryptography –

It is also known as public key cryptography because it involves usage of a public key along with secret key. It solves the problem of key distribution as both parties uses different keys for encryption/decryption. It is not feasible to use for decrypting bulk messages as it is very slow compared to symmetric key cryptography.



Pros of Asymmetric key cryptography:

- In asymmetric or public key, cryptography there is no need for exchanging keys, thus eliminating the key distribution problem.
- The primary advantage of public-key cryptography is increased security: the private keys do not ever need to be transmitted or revealed to anyone.
- Can provide digital signatures that can be repudiated

Cons of Asymmetric key cryptography:

- It is a slow process.
- It risks loss of private key, which may be irreparable.
-

References:

https://searchsecurity.techtarget.com/definition/cryptography

https://www.geeksforgeeks.org/cryptography-introduction-to-crypto-terminologies/

http://www.uobabylon.edu.iq/eprints/paper_1_2264_649.pdf

https://greengarageblog.org/8-pros-and-cons-of-asymmetric-encryption

PRACTICAL-2

Aim: Caesar Cipher

```
text = input("Enter plain text:")
key = input("Enter key:")
encry = []
for ch in range(len(text)):
    e = ord(text[ch])
    if(e>64 and e<91):
        e = ord(text[ch])+int(key)
        if(e>90):
            set = e-90
            e = set + 64
        encry.append(chr(e))
    if(e>96 and e<123):
        e = ord(text[ch])+int(key)
        if(e>122):
            set = e-122
            e = set + 96
        encry.append(chr(e))
#for enc in encry:

print("Encrypted Message:")
print(encry)

decry = []
for ch in range(len(encry)):
    d = ord(encry[ch])
    if(d>64 and d<91):
        d = ord(encry[ch])-int(key)
        if(d<65):
            set = 65 - d
            d = 91-set
        decry.append(chr(d))
    if(d>96 and d<123):
        d = ord(encry[ch])-int(key)
        if(d<97):
            set = 97 - d
            d = 123-set
        decry.append(chr(d))
print("Decrypted Message:")
print(decry)
```

OUTPUT:

```
Enter plain text:hello
Enter key:3
Encrypted Message:
khoor
Decrypted Message:
hello
```

PRACTICAL-3

Aim: Columnar Cipher

```
key = input("Enter Key:")
pt = input("Enter Text:")

lenkey = len(key)
lenpt = len(pt)

#----------Encryption-----------

a=0
while (lenpt % lenkey) != 0:
    a=a+1
    lenpt = lenpt + 1

for i in range(a):
    pt = pt + "*"

dict = {}
encry = ""
for i in range(lenkey):
    for j in range(i,lenpt,lenkey):
        encry = encry + pt[j]
    dict[str(i+1)] = encry
    encry = ""

dic = {}
j=0
for i in range(lenkey):
    dic[str(j+1)] = key[j]
    j = j+1

dicValue = list(dic.values())
dicValue.sort()

keyy = ""
k=0

for i in dicValue:
    for j,l in dic.items():
        if i==l:
            keyy = keyy + j
            k+=1



encryption = ""
for k in keyy:
    encryption = encryption + dict[k]
```

```
#-----------Decryption-------------

lenenc = len(encryption)

dictd = {}

decry = ""
k=0
d = int(lenenc/lenkey)
for i in range(lenkey):
    for j in range(d):
        decry = decry + encryption[k]
        k = k+1
    dictd[str(i+1)] = decry
    decry = ""

dc = ""
keyyy = ""
i=0
k=0
dicc = {}
j=0
for i in range(lenkey):
    dicc[str(j+1)] = key[j]
    j = j+1

dicValue = list(dic.values())
dicValue.sort()

for i in dicValue:
    for j,l in dicc.items():
        if i==l:
            keyyy = keyyy + j
            k+=1

for k in keyyy:
    dc =  dc + dictd[k]

decryption = ""
de = ""

l = len(dictd['1'])
ll = int(lenenc / lenkey)


for i in range(ll):
    for j in range(i,lenenc,l):
        de = de + dc[j]
    decryption = decryption + de
    de = ""
```

```
encryption=encryption.replace("*","")
decryption=decryption.replace("*","")
print("Encrypted Text: " + encryption)
print("Decrypted Text: " + decryption)
```

OUTPUT:

```
Enter Key:1324
Enter Text:cryptography
Encrypted Text: ctayghroppry
Decrypted Text: cryptography
```

PRACTICAL-4

Aim: Playfair Cipher

```
key = input("Enter Key:")
pt = input("Enter Plain Text:")
key = key.replace("j","i")
lenpt = len(pt)
i=0
opt = pt

if (lenpt % 2) != 0:
    pt = pt + "x"
ptGroup = []
j=0
k=2
l = int(len(pt)/2)
for i in range(l):
    ptGroup.append('')
for i in range(l):
    ptGroup[i] = pt[j:k]
    j=j+2
    k=k+2

optGroup = ptGroup
print(opt)
print(optGroup)
for i in range(len(ptGroup)):
    ptgrp = ptGroup[i]
    if(ptgrp[0]==ptgrp[1]):
        one = ptgrp[0]
        two = ptgrp[1]
        ptgrp = ptgrp.replace(ptgrp[0],"")
        ptgrp = one + "x"
        #print(ptgrp)
    ptGroup[i] = ptgrp
matrix = []

for i in key:
    if i not in matrix:
        matrix.append(i)

alphabet = "abcdefghiklmnopqrstuvwxyz"

for i in alphabet:
    if i not in matrix:
        matrix.append(i)
matrix = "".join(matrix)
matrixGroup = []

for e in range(5):
            matrixGroup.append('')
```

```
matrixGroup[0] = matrix[0:5]
matrixGroup[1] = matrix[5:10]
matrixGroup[2] = matrix[10:15]
matrixGroup[3] = matrix[15:20]
matrixGroup[4] = matrix[20:25]

print(ptGroup)
print(matrixGroup)

def findPosition(char):
    k=0
    for i in range(len(matrixGroup)):
        if char in matrixGroup[i]:
            s = matrixGroup[i]
            for j in s:
                if char in j:
                    return i,k
                k = k + 1

def findCharE(x,xi,y,yi):
    if x != y and xi != yi:
        group1 = matrixGroup[x]
        group2 = matrixGroup[y]
        k1 = group1[yi]
        k2 = group2[xi]
        return k1,k2

    if xi == yi:
        if x == 4:
            x = -1
        if y == 4:
            y = -1
        group1 = matrixGroup[x+1]
        group2 = matrixGroup[y+1]

        k1 = group1[xi]  # or k1 = group1[yi]
        k2 = group2[xi]  # or k2 = group2[yi]
        return k1,k2

    if x == y:
        if xi == 4:
            xi = -1
        if yi == 4:
            yi = -1
        group = matrixGroup[x]  # or group = matrixGroup[y]
        k1 = group[xi+1]
        k2 = group[yi+1]
        return k1,k2
```

```
def findCharD(x,xi,y,yi):
    if x != y and xi != yi:
        group1 = matrixGroup[x]
        group2 = matrixGroup[y]
        k1 = group1[yi]
        k2 = group2[xi]
        return k1,k2

    if xi == yi:
        if x == 0:
            x = 5
        if y == 0:
            y = 5
        group1 = matrixGroup[x-1]
        group2 = matrixGroup[y-1]

        k1 = group1[xi]  # or k1 = group1[yi]
        k2 = group2[xi]  # or k2 = group2[yi]
        return k1,k2

    if x == y:
        if xi == 0:
            xi = 5
        if yi == 0:
            yi = 5
        group = matrixGroup[x]  # or group = matrixGroup[y]
        k1 = group[xi-1]
        k2 = group[yi-1]
        return k1,k2

encry = ""
for i in range(len(ptGroup)):
    z = ptGroup[i]
    x,xi = findPosition(z[0])
    y,yi = findPosition(z[1])
    k1,k2 = findCharE(x,xi,y,yi)
    encry = encry + k1 + k2


enGroup = []
j=0
k=2
l = int(len(encry)/2)
for i in range(l):
    enGroup.append('')
for i in range(l):
    enGroup[i] = encry[j:k]
    j=j+2
    k=k+2



decry = ""
```

```
for i in range(len(enGroup)):
    z = enGroup[i]
    x,xi = findPosition(z[0])
    y,yi = findPosition(z[1])
    k1,k2 = findCharD(x,xi,y,yi)
    decry = decry + k1 + k2



if len(decry)!=len(opt):
    decry = decry[:-1]
print("Encrypted Text:" + encry)
print("Decrypted Text:" + decry)
```

OUTPUT:

```
Enter Key:hello
Enter Plain Text:cryptography
['cr', 'yp', 'to', 'gr', 'ap', 'hy']
['heloa', 'bcdfg', 'ikmnp', 'qrstu', 'vwxyz']
Encrypted Text:kwznyfcuguov
Decrypted Text:cryptography
```

PRACTICAL-5

Aim: Hill Cipher

```
import numpy as np
def encrypt(msg):
    msg = msg.replace(" ", "")
    C = make_key()
    len_check = len(msg) % 2 == 0
    if not len_check:
        msg += "0"
    P = create_matrix_of_integers_from_string(msg)
    msg_len = int(len(msg) / 2)
    encrypted_msg = ""
    for i in range(msg_len):
        row_0 = P[0][i] * C[0][0] + P[1][i] * C[0][1]
        integer = int(row_0 % 26 + 65)
        encrypted_msg += chr(integer)
        row_1 = P[0][i] * C[1][0] + P[1][i] * C[1][1]
        integer = int(row_1 % 26 + 65)
        encrypted_msg += chr(integer)
    return encrypted_msg

def decrypt(encrypted_msg):
    C = make_key()
    determinant = C[0][0] * C[1][1] - C[0][1] * C[1][0]
    determinant = determinant % 26
    multiplicative_inverse = find_multiplicative_inverse(determinant)
    C_inverse = C
    C_inverse[0][0], C_inverse[1][1] = C_inverse[1, 1], C_inverse[0, 0]
    C[0][1] *= -1
    C[1][0] *= -1
    for row in range(2):
        for column in range(2):
            C_inverse[row][column] *= multiplicative_inverse
            C_inverse[row][column] = C_inverse[row][column] % 26

    P = create_matrix_of_integers_from_string(encrypted_msg)
    msg_len = int(len(encrypted_msg) / 2)
    decrypted_msg = ""
    for i in range(msg_len):
        column_0 = P[0][i] * C_inverse[0][0] + P[1][i] * C_inverse[0][1]
        integer = int(column_0 % 26 + 65)
        decrypted_msg += chr(integer)
        column_1 = P[0][i] * C_inverse[1][0] + P[1][i] * C_inverse[1][1]
        integer = int(column_1 % 26 + 65)
        decrypted_msg += chr(integer)
    if decrypted_msg[-1] == "0":
        decrypted_msg = decrypted_msg[:-1]
    return decrypted_msg

def find_multiplicative_inverse(determinant):
    multiplicative_inverse = -1
```

```
    for i in range(26):
        inverse = determinant * i
        if inverse % 26 == 1:
            multiplicative_inverse = i
            break
    return multiplicative_inverse

def make_key():
    determinant = 0
    C = None
    while True:
        cipher = input("Input 4 letter cipher: ")
        C = create_matrix_of_integers_from_string(cipher)
        determinant = C[0][0] * C[1][1] - C[0][1] * C[1][0]
        determinant = determinant % 26
        inverse_element = find_multiplicative_inverse(determinant)
        if inverse_element == -1:
            print("Determinant is not relatively prime to 26, uninvertible key")
        elif np.amax(C) > 26 and np.amin(C) < 0:
            print("Only a-z characters are accepted")
            print(np.amax(C), np.amin(C))
        else:
            break
    return C

def create_matrix_of_integers_from_string(string):
    integers = [chr_to_int(c) for c in string]
    length = len(integers)
    M = np.zeros((2, int(length / 2)), dtype=np.int32)
    iterator = 0
    for column in range(int(length / 2)):
        for row in range(2):
            M[row][column] = integers[iterator]
            iterator += 1
    return M

def chr_to_int(char):
    char = char.upper()
    integer = ord(char) - 65
    return integer

if __name__ == "__main__":
    msg = input("Message: ")
    encrypted_msg = encrypt(msg)
    print(encrypted_msg)
    decrypted_msg = decrypt(encrypted_msg)
    print(decrypted_msg)
```

OUTPUT:

```
Enter plain text: hell
Enter Key:5462
Encrypted message: CFHE
Decrypted message: HELL
```

PRACTICAL-6

Aim: Vigenere Cipher

```
def generateKey(string, key):
	key = list(key)
	if len(string) == len(key):
		return(key)
	else:
		for i in range(len(string) - len(key)):
			key.append(key[i % len(key)])
	return("" . join(key))

def cipherText(string, key):
	cipher_text = []
	for i in range(len(string)):
		x = (ord(string[i]) + ord(key[i])) % 26
		x += ord('A')
		cipher_text.append(chr(x))
	return("" . join(cipher_text))

def originalText(cipher_text, key):
	orig_text = []
	for i in range(len(cipher_text)):
		x = (ord(cipher_text[i]) - ord(key[i]) + 26) % 26
		x += ord('A')
		orig_text.append(chr(x))
	return("" . join(orig_text))

if __name__ == "__main__":
    string = input("Enter Plain text\n")
    keyword = input("Enter Key\n")
    key = generateKey(string, keyword)
    cipher_text = cipherText(string,key)
    print("Ciphertext :", cipher_text)
    print("Original/Decrypted Text :",originalText(cipher_text, key))
```

OUTPUT:

```
Enter Plain text
HELLO
Enter Key
ABC
Ciphertext : HFNLP
Original/Decrypted Text : HELLO
```

PRACTICAL-7

Aim: Rail Fence Cipher

```
def encryptRailFence(text, key):

    rail = [['\n' for i in range(len(text))]
                  for j in range(key)]

    dir_down = False
    row, col = 0, 0

    for i in range(len(text)):
        if (row == 0) or (row == key - 1):
            dir_down = not dir_down

        rail[row][col] = text[i]
        col += 1

        if dir_down:
            row += 1
        else:
            row -= 1

    result = []
    for i in range(key):
        for j in range(len(text)):
            if rail[i][j] != '\n':
                result.append(rail[i][j])
    return("" . join(result))

def decryptRailFence(cipher, key):

    rail = [['\n' for i in range(len(cipher))]
                  for j in range(key)]

    dir_down = None
    row, col = 0, 0

    for i in range(len(cipher)):
        if row == 0:
            dir_down = True
        if row == key - 1:
            dir_down = False

        rail[row][col] = '*'
        col += 1

        if dir_down:
            row += 1
        else:
            row -= 1
```

```
    index = 0
    for i in range(key):
        for j in range(len(cipher)):
            if ((rail[i][j] == '*') and
               (index < len(cipher))):
                rail[i][j] = cipher[index]
                index += 1

    result = []
    row, col = 0, 0
    for i in range(len(cipher)):

        if row == 0:
            dir_down = True
        if row == key-1:
            dir_down = False

        if (rail[row][col] != '*'):
            result.append(rail[row][col])
            col += 1

        if dir_down:
            row += 1
        else:
            row -= 1
    return("".join(result))

if __name__ == "__main__":
    print(encryptRailFence("attack at once", 2))
    print(decryptRailFence("atc toctaka ne", 2))
```

OUTPUT:

```
atc toctaka ne
attack at once
```

PRACTICAL-8

Aim: Vernum Cipher

```
def makeVernamCypher( text, key ):
    answer = "" # the Cypher text
    p = 0 # pointer for the key
    for char in text:
        answer += chr(ord(char) ^ ord(key[p]))
        p += 1
        if p==len(key):
            p = 0
    return answer


MY_KEY = "cvwopslweinedvq9fnasdlkfn2"
while True:
    print("\n\n---Vernam Cypher---")
    PlainText = input("Enter text to encrypt: ")
    # Encrypt
    Cypher = makeVernamCypher(PlainText, MY_KEY)
    print("Cypher text: "+Cypher)
    # Decrypt
    decrypt = makeVernamCypher(Cypher, MY_KEY)
    print("Decrypt: "+decrypt)
```

OUTPUT:

```
---Vernam Cypher---
Enter text to encrypt: HELLO
Cypher text: +3;#?
Decrypt: HELLO
```

PRACTICAL-9

Aim: Diffie Hellman Key Exchange

```
import math
import random
global prime, root

def secretnumber ():
    secret = int(random.randint(0,100))
    return secret

prime = 17
print("The prime is ",prime, "\n")
root = 3
print("The root is",root, "\n")

alicesecret = secretnumber()
print("Alice chooses a secret number",alicesecret)

bobsecret = secretnumber()
print("Bob chooses a secret number", bobsecret, "\n")

alicepublic = (root ** alicesecret) % prime
print("Alice's public key is",alicepublic, "\n")
bobpublic = (root ** bobsecret) % prime
print("Bob's public key is", bobpublic, "\n")

alicekey = (bobpublic ** alicesecret) % prime
bobkey = (alicepublic ** bobsecret) % prime
print("Alice calculates the shared key and gets", alicekey)
print("Bob calculates the shared key and gets", bobkey, "\n")
```

OUTPUT:

```
The prime is  17

The root is 3

Alice chooses a secret number 31
Bob chooses a secret number 15

Alice's public key is 6

Bob's public key is 6

Alice calculates the shared key and gets 3
Bob calculates the shared key and gets 3
```

Practical-10

Aim: Case study of DES Algorithm

DES Algorithm :

- have a high security level related to a small key used for encryption and decryption
- be easily understood
- not depend on the algorithm's confidentiality
- be adaptable and economical
- be efficient and exportable

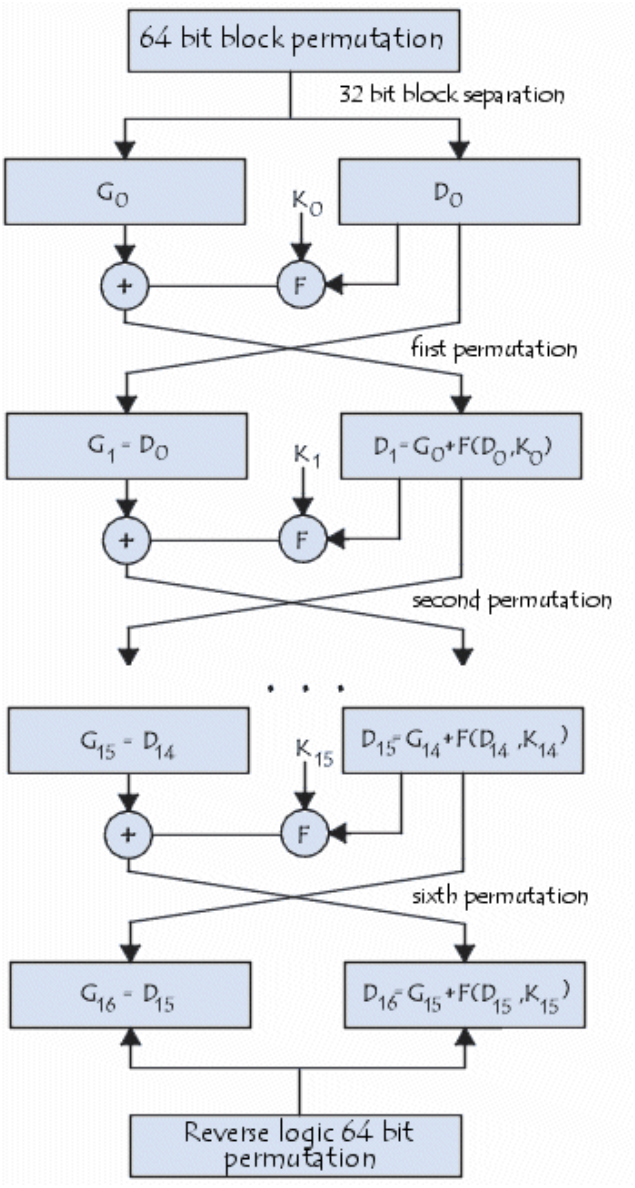
In late 1974, IBM proposed "Lucifer", which, thanks to the NSA (National Security Agency), was modified on 23 November 1976 to become the **DES** (*Data Encryption Standard*). The DES was approved by the NBS in 1978. The DES was standardized by the *ANSI* (*American National Standard Institute*) under the name of *ANSI X3.92*, better known as *DEA* (*Data Encryption Algorithm*).

Principle of the DES

It is a symmetric encryption system that uses 64-bit blocks, 8 bits (one octet) of which are used for parity checks (to verify the key's integrity). Each of the key's parity bits (1 every 8 bits) is used to check one of the key's octets by odd parity, that is, each of the parity bits is adjusted to have an odd number of '1's in the octet it belongs to. The key therefore has a "useful" length of 56 bits, which means that only 56 bits are actually used in the algorithm.

The algorithm involves carrying out combinations, substitutions and permutations between the text to be encrypted and the key, while making sure the operations can be performed in both directions (for decryption). The combination of substitutions and permutations is called a **product cipher**.

The key is ciphered on 64 bits and made of 16 blocks of 4 bits, generally denoted k_1 to k_{16}. Given that "only" 56 bits are actually used for encrypting, there can be 2^{56} (or $7.2*10^{16}$) different keys!



The DES algorithm

The main parts of the algorithm are as follows:

- Fractioning of the text into 64-bit (8 octet) blocks;
- Initial permutation of blocks;
- Breakdown of the blocks into two parts: left and right, named *L* and *R*;
- Permutation and substitution steps repeated 16 times (called **rounds**);
- Re-joining of the left and right parts then inverse initial permutation.

Fractioning of the text

Initial permutation

Firstly, each bit of a block is subject to initial permutation, which can be represented by the following initial permutation (*IP*) table:

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

This permutation table shows, when reading the table from left to right then from top to bottom, that the 58th bit of the 64-bit block is in first position, the 50th in second position and so forth.
Division into 32-bit blocks

Once the initial permutation is completed, the 64-bit block is divided into two 32-bit blocks, respectively denoted **L** and **R** (for left and right). The initial status of these two blocks is denoted $\mathbf{L}_0$ and $\mathbf{R}_0$:

$\mathbf{L}_0$

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

$\mathbf{R}_0$

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

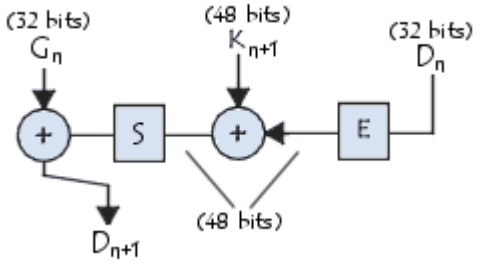
It is interesting to note that $\mathbf{L}_0$ contains all bits having an even position in the initial message, whereas $\mathbf{R}_0$ contains bits with an odd position.

Rounds

The $\mathbf{L}_n$ and $\mathbf{R}_n$ blocks are subject to a set of repeated transformations called *rounds*, shown in this diagram, and the details of which are given below:

Expansion function

The 32 bits of the $\mathbf{R}_0$ block are expanded to 48 bits thanks to a table called an *expansion table* (denoted **E**), in which the 48 bits are mixed together and 16 of them are duplicated:



E	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

As such, the last bit of $\mathbf{R}_0$ (that is, the 7th bit of the original block) becomes the first, the first becomes the second, etc. In addition, the bits 1,4,5,8,9,12,13,16,17,20,21,24,25,28 and 29 of $\mathbf{R}_0$ (respectively 57, 33, 25, l, 59, 35, 27, 3, 6l, 37, 29, 5, 63, 39, 31 and 7 of the original block) are duplicated and scattered in the table.
exclusive OR with the key

The resulting 48-bit table is called $\mathbf{R'}_0$ or $\mathbf{E}[\mathbf{R}_0]$. The DES algorithm then *exclusive ORs* the first key $\mathbf{K}_1$ with $\mathbf{E}[\mathbf{R}_0]$. The result of this *exclusive OR* is a 48-bit table we will call $\mathbf{R}_0$ out of convenience (it is not the starting $\mathbf{R}_0$!).
Substitution function

$\mathbf{R}_0$ is then divided into 8 6-bit blocks, denoted $\mathbf{R}_{0i}$. Each of these blocks is processed by **selection functions** (sometimes called *substitution boxes* or *compression functions*), generally denoted $\mathbf{S}_i$. The first and last bits of each $\mathbf{R}_{0i}$ determine (in binary value) the line of the selection function; the other bits (respectively 2, 3, 4 and 5) determine the column. As the selection of the line is based on two bits, there are 4 possibilities (0,1,2,3). As the selection of the column is based on 4 bits, there are 16 possibilities (0 to 15). Thanks to this information, the selection function "selects" a ciphered value of 4 bits.
Here is the first substitution function, represented by a 4-by-16 table:

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
	0	14	4	13	1	2	15	11	8	3	10	6	12	5	9	0	7
$\mathbf{S}_1$	**1**	0	15	7	4	14	2	13	1	10	6	12	11	9	5	3	8
	2	4	1	14	8	13	6	2	11	15	12	9	7	3	10	5	0
	3	15	12	8	2	4	9	1	7	5	11	3	14	10	0	6	13

Let $\mathbf{R}_{01}$ equal *101110*. The first and last bits give *10*, that is, 2 in binary value. The bits 2,3,4 and 5 give *0111*, or 7 in binary value. The result of the selection function is therefore the value located on line no. 2, in column no. 7. It is the value *11*, or *111* binary.
Each of the 8 6-bit blocks is passed through the corresponding selection function, which gives an output of 8 values with 4 bits each. Here are the other selection functions:

	0	15	1	8	14	6	11	3	4	9	7	2	13	12	0	5	10
S_2	**1**	3	13	4	7	15	2	8	14	12	0	1	10	6	9	11	5
	2	0	14	7	11	10	4	13	1	5	8	12	6	9	3	2	15
	3	13	8	10	1	3	15	4	2	11	6	7	12	0	5	14	9

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
	0	10	0	9	14	6	3	15	5	1	13	12	7	11	4	2	8
S_3	**1**	13	7	0	9	3	4	6	10	2	8	5	14	12	11	15	1
	2	13	6	4	9	8	15	3	0	11	1	2	12	5	10	14	7
	3	1	10	13	0	6	9	8	7	4	15	14	3	11	5	2	12

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
	0	7	13	14	3	0	6	9	10	1	2	8	5	11	12	4	15
S_4	**1**	13	8	11	5	6	15	0	3	4	7	2	12	1	10	14	9
	2	10	6	9	0	12	11	7	13	15	1	3	14	5	2	8	4
	3	3	15	0	6	10	1	13	8	9	4	5	11	12	7	2	14

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
	0	2	12	4	1	7	10	11	6	8	5	3	15	13	0	14	9
S_5	**1**	14	11	2	12	4	7	13	1	5	0	15	10	3	9	8	6
	2	4	2	1	11	10	13	7	8	15	9	12	5	6	3	0	14
	3	11	8	12	7	1	14	2	13	6	15	0	9	10	4	5	3

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
	0	12	1	10	15	9	2	6	8	0	13	3	4	14	7	5	11
S_6	**1**	10	15	4	2	7	12	9	5	6	1	13	14	0	11	3	8
	2	9	14	15	5	2	8	12	3	7	0	4	10	1	13	11	6
	3	4	3	2	12	9	5	15	10	11	14	1	7	6	0	8	13

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
	0	4	11	2	14	15	0	8	13	3	12	9	7	5	10	6	1
S_7	**1**	13	0	11	7	4	9	1	10	14	3	5	12	2	15	8	6
	2	1	4	11	13	12	3	7	14	10	15	6	8	0	5	9	2
	3	6	11	13	8	1	4	10	7	9	5	0	15	14	2	3	12

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
	0	13	2	8	4	6	15	11	1	10	9	3	14	5	0	12	7
S_8	**1**	1	15	13	8	10	3	7	4	12	5	6	11	0	14	9	2
	1	7	11	4	1	9	12	14	2	0	6	10	13	15	3	5	8
	1	2	1	14	7	4	10	8	13	15	12	9	0	3	5	6	11

Each 6-bit block is therefore substituted in a 4-bit block. These bits are combined to form a 32-bit block.
Permutation

The obtained 32-bit block is then subject to a permutation **P** here is the table:

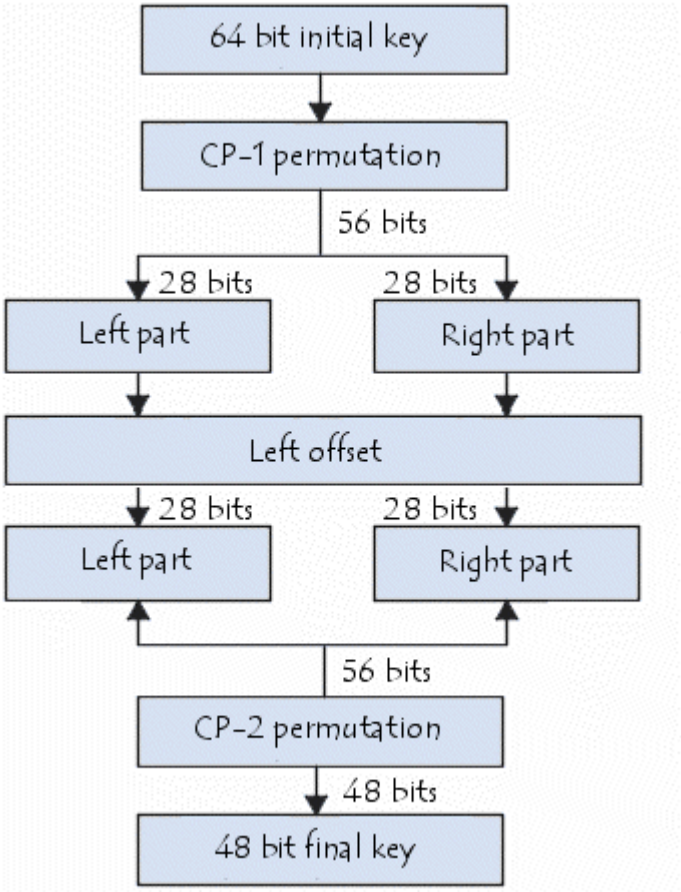
P	16	7	20	21	29	12	28	17
	1	15	23	26	5	18	31	10
	2	8	24	14	32	27	3	9
	19	13	30	6	22	11	4	25

Exclusive OR

All of these results output from **P** are subject to an *Exclusive OR* with the starting $\mathbf{L}_0$ (as shown on the first diagram) to give R1, whereas the initial $\mathbf{R}_0$ gives $\mathbf{L}_1$.
Iteration

All of the previous steps (*rounds*) are repeated 16 times.
Inverse initial permutation

At the end of the iterations, the two blocks $\mathbf{L}_{16}$ and $\mathbf{R}_{16}$ are re-joined, then subject to inverse initial permutation:



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

The output result is a 64-bit ciphertext!
Generation of keys

Given that the DES algorithm presented above is public, security is based on the complexity of encryption keys.

The algorithm below shows how to obtain, from a 64-bit key (made of any 64 alphanumeric characters), 8 different 48-bit keys each used in the DES algorithm:

Firstly, the key's parity bits are eliminated so as to obtain a key with a useful length of 56 bits.

PC-1

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

L_i

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

R_i

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

The result of this first permutation is denoted $\mathbf{L}_0$ and $\mathbf{R}_0$.
These two blocks are then rotated to the left, such that the bits in second position take the first position, those in third position take the second, etc.
The bits in first position move to last position.
The 2 28-bit blocks are then grouped into one 56-bit block. This passes through a permutation, denoted **PC-2**, giving a 48-bit block as output, representing the key $\mathbf{K}_i$.

PC-2

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		

Repeating the algorithm makes it possible to give the 16 keys $\mathbf{K}_1$ to $\mathbf{K}_{16}$ used in the DES algorithm.
LS 1 2 4 6 8 10 12 14 15 17 19 21 23 25 27 28

Practical-12

Aim: Case study of RSA algorithm

RSA : Key Generation Algorithm

This is the original 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.
2. Compute n = pq and (phi) ϕ = (p-1)(q-1).
3. Choose an integer *e*, 1 < e < phi, such that gcd(e, phi) = 1.
4. Compute the secret exponent *d*, 1 < d < phi, such that ed ≡ 1 (mod phi).
5. The public key is (n, e) and the private key (d, p, q). Keep all the values d, p, q and phi secret. [We prefer sometimes to write the private key as (n, d) because you need the value of n when using d.]

- 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*, 1 < m < n.
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*.

Digital signing

Sender A does the following:-

1. Creates a *message digest* of the information to be sent.
2. Represents this digest as an integer *m* between 1 and *n*-1.
3. Uses her *private* key (n, d) to compute the signature $s = m^d \bmod n$.
4. Sends this signature *s* to the recipient, B.

Signature verification

Recipient B does the following:-

1. Uses sender A's public key (n, e) to compute integer $v = s^e \bmod n$.
2. Extracts the message digest from this integer.
3. Independently computes the message digest of the information that has been signed.
4. If both message digests are identical, the signature is valid.