Prof:Siddhesh Zele's

WE IT TUTORIALS

NETWORK SECURITY UNIT 1,2 AND 3

TYBSC(IT) SEM 5

COMPILED BY : SIDDHESH ZELE AND MADHURA VIDYA

302 PARANJPE UDYOG BHAVAN, NEAR KHANDELWAL SWEETS, NEAR THANE STATION , THANE (WEST)

PHONE NO: 8097071144 / 8097071155 / 8655081002

Prof: siddhesh zele's
We-IT Tutorials

UNIT 1

Computer Security

The meaning of the term computer security has evolved in recent years. Before the problem of data security became widely publicized in the media, most people's idea of computer security focused on the physical machine. Traditionally, computer facilities have been physically

protected for three reasons:

- To prevent theft of or damage to the hardware
- To prevent theft of or damage to the information
- To prevent disruption of service

Strict procedures for access to the machine room are used by most organizations, and these procedures are often an organization's only obvious computer security measures. Today, however, with pervasive remote terminal access, communications, and networking, physical measures rarely provide meaningful protection for either the information or the service; only the hardware is secure. Nonetheless, most computer facilities continue to protect their physical machine far better than they do their data, even when the value of the data is several times greater than the value of the hardware.

Why Do We Need Security?

In the ever changing world of global data communications, inexpensive Internet connections, and fast-paced software development, security is becoming more and more of an issue. Security is now a basic requirement because global computing is inherently insecure. As your data goes from point A to point B on the Internet, for example, it may pass through several other points along the way, giving other users the opportunity to intercept, and even alter it. It does nothing to protect your data center, other servers in your network, or a malicious user with physical access to your EnGarde system.

Security Models

1. **No Security**
   In this simplest case, the approach could be a decision to implement no security at all.
2. **Security through Obscurity**
   In this model, a system is secure simply because nobody knows about its existence and contents. This approach cannot work for too long, as there are many ways an attacker can come to know about it.
3. **Host Security**
   In this scheme, the security for each host is enforced individually. This is a very safe approach, but the trouble is that it cannot scale well. The complexity and diversity of modem sites/organizations makes the task even harder.
4. **Network Security**

Host security is tough to achieve as organizations grow and become more diverse. In this technique, the focus is to control network access to various hosts and their services, rather than individual host security. This is a very efficient and scalable model.

PRINCIPLES OF SECURITY

Let us assume that a person A wants to send a check worth $100 to another person B. Normally, what are the factors that A and B will think of, in such a case? A will write the check for $100, put it inside an envelope, and send it to B.

- A will like to ensure that no one except B gets the envelope, and even if someone else gets it, he/she does not come to know about the details of the check. This is the principle of **confidentiality.**
- A and B will further like to make sure that no one can tamper with the contents of the check (such as its amount, date, signature, name of the payee, etc.). This is the principle of **integrity.**
- B would like to be assured that the check has indeed come from A, and not from someone else posing as A (as it could be a fake check in that case). This is the principle of **authentication.**
- What will happen tomorrow if B deposits the check in his/her account, the money is transferred from A's account to B's account, and then A refuses having written/sent the check? The court of law will use A's signature to disallow A to refute this claim, and settle the dispute. This is the principle of **non-repudiation.**

These are the four chief principles of security. There are two more: **access control** and **availability,** which are not related to a particular message, but are linked to the overall system as a whole.

Confidentiality

The principle of *confidentiality* specifies that only the sender and the intended recipient(s) should be able to access the contents of a message. Confidentiality gets compromised if an unauthorized person is able to access a message.

Interception causes loss of message confidentiality.

Authentication

Authentication mechanisms help establish **proof of identities.** The authentication process ensures that the origin of an electronic message or document is correctly identified.

Fabrication is possible in absence of proper authentication mechanisms.

Integrity

When the contents of a message are changed after the sender sends it, but before it reaches the intended recipient, we say that the *integrity* of the message is lost.

Modification causes loss o f message integrity.

Non-repudiation

There are situations where a user sends a message, and later on refuses that she had sent that message.

Access Control

The principle of *access control* determines *who* should be able to access *what.*

Availability

The principle of *availability* states that resources (i.e. information) should be available to authorized parties at all times.

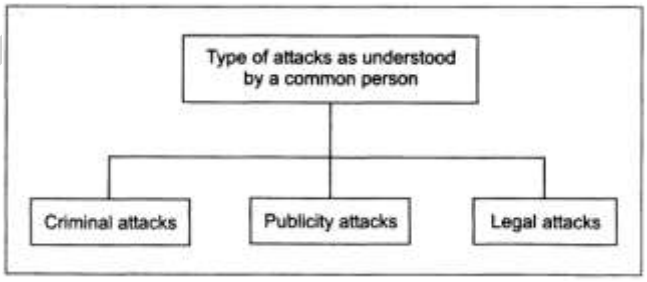
OSI standard for Security Model

- Authentication
- Access control
- Non-repudiation
- Data integrity
- Confidentiality
- Assurance or availability
- Notarization or signature

TYPES OF ATTACKS

attacks with respect to two views: the common person's view and a technologist's view.

Attacks: A General View



Criminal Attacks

Criminal attacks are the simplest to understand. Here, the 'sole aim of the attackers is to maximize financial gain by attacking computer systems.

Publicity Attacks

Publicity attacks occur because the attackers want to see their names appear on television news channels and newspapers.

Legal Attacks

This form of attack is quite novel and unique. Here, the attacker tries to make the judge or the jury doubtful about the security of a computer system. This works as follows. The attacker attacks the computer system, and the attacked party (say a bank or an organization) manages to take the attacker to the court. While the case is being fought, the attacker tries to convince the judge and the jury that there is inherent weakness in the computer system and that she has done nothing wrongful. The aim of the attacker is to exploit the weakness of the judge and the jury in technological matters.

Attacks: A Technical View

From a technical point of view, we can classify the types of attacks on computers and network systems into two categories for better understanding: **(a) Theoretical concepts** behind these attacks, and **(b) Practical approaches** used by the attackers. Let us discuss these one by one.

Theoretical Concepts

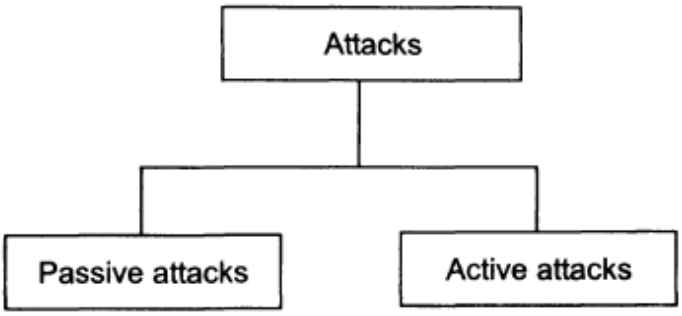
Interception It has been discussed in the context of *confidentiality* earlier. It means that an unauthorized party has gained access to a resource. The party can be a person, program, or computer-based system. Examples of interception are copying of data or programs, and listening to network traffic.

Fabrication It has been discussed in the context of *authentication* earlier. This involves the creation of illegal objects on a computer system. For example, the attacker may add fake records to a database.

Modification It has been discussed in the context of *integrity* earlier. Here, the attacker may modify the values in a database.

Interruption It has been discussed in the context of *availability* earlier. Here, the resource becomes unavailable, lost, or unusable. Examples of interruption are causing problems to a hardware device, erasing program, data, or operating-system components.

These attacks are further grouped into two types: passive attacks and active attacks



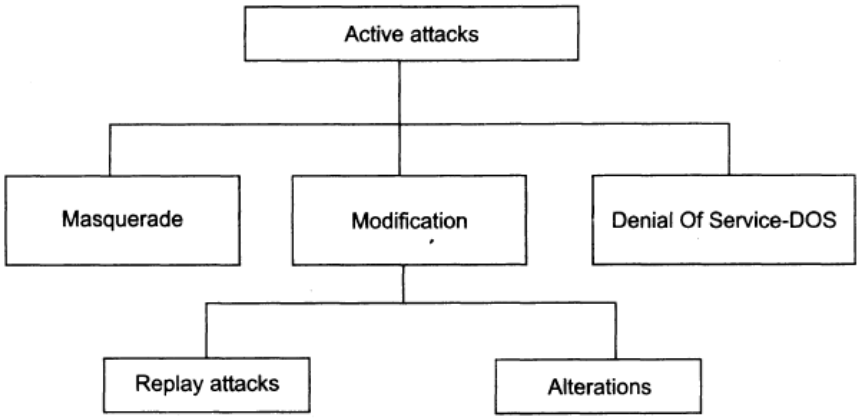
Passive Attacks *Passive attacks* are those wherein the attacker indulges in eavesdropping or monitoring of data transmission.

Passive attacks do not involve any modifications to the contents of an original message

types: **release of message contents** and **traffic analysis**

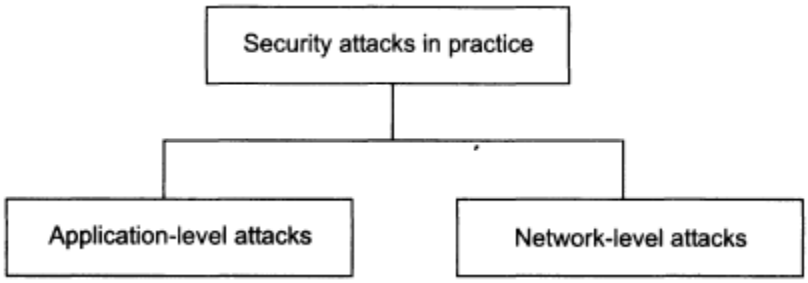
Active Attacks Unlike *passive attacks,* the *active attacks* are based on the modification of the original message in some manner, or in the creation of a false message.

In active attacks**,** the contents o f the original message are modified in some way.



Denial Of Service (DOS) attacks make an attempt to prevent legitimate users from accessing some services, which they are eligible for. For instance, an unauthorized user might send too many login requests to a server using random user ids in quick succession, so as to flood the network and deny other legitimate users to use the network facilities.

The Practical Side of Attacks



1. Application-level Attacks

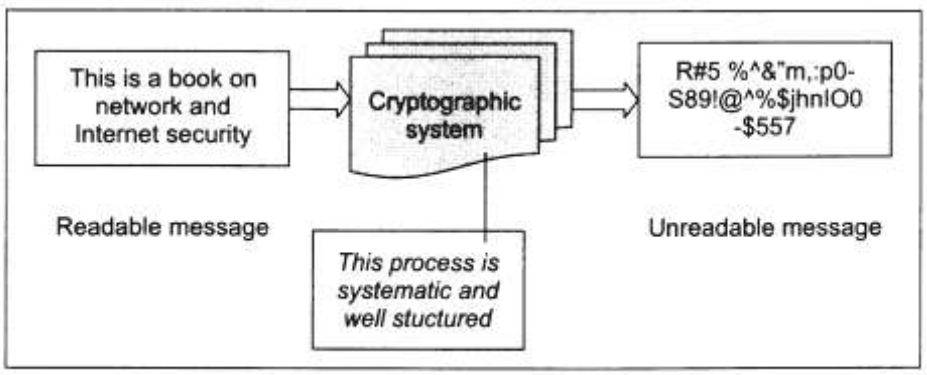
These attacks happen at an application level in the sense that the attacker attempts to access, modify, or prevent access to information of a particular application, or the application itself. Examples of this are trying to obtain someone's credit-card information on the Internet, or changing the contents of a message to change the amount in a transaction, etc.

2. Network-level Attacks

These attacks generally aim at reducing the capabilities of a network by a number of possible means. These attacks generally make an attempt to either slow down, or completely bring to halt, a computer network. Note that this automatically can lead to application-level attacks, because once someone is able to gain access to a network, usually he/she is able to access/modify at least some sensitive information, causing havoc.

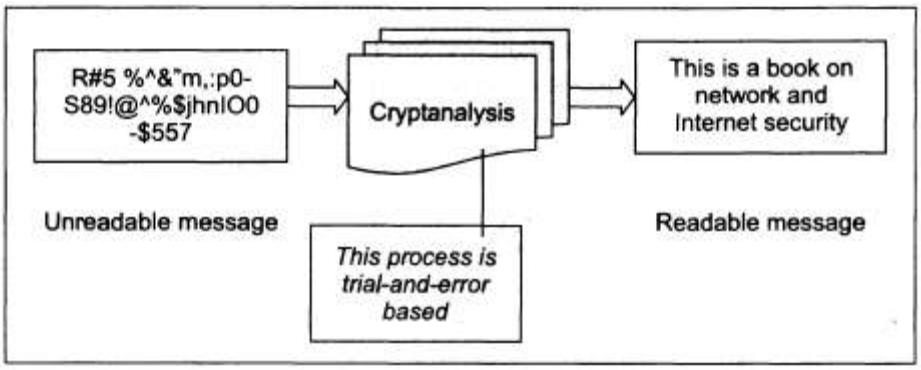
CRYPTOGRAPHY TECHNIQUES

Cryptography is the art o f achieving security by encoding messages to make them non-readable.



Cryptographic system

Cryptanalysis is the technique o f decoding messages from a non-readable format back to a readable format without knowing how they were initially converted from readable format to nonreadable format.



Cryptanalysis

Cryptology *is a combination of cryptography and cryptanalysis.*

Cryptography

\+ = Cryptology

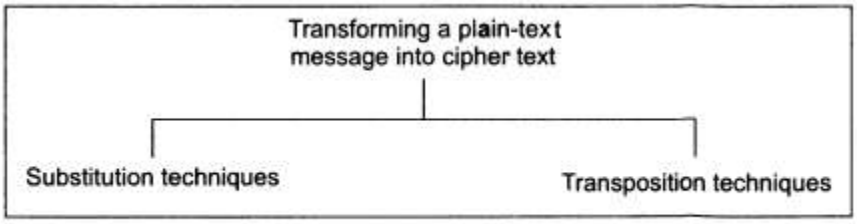
Cryptanalysis

PLAIN TEXT AND CIPHER TEXT

Clear text**,** or **plain text**, signifies a message that can be understood by the sender, the recipient, and also by anyone else who gets access to that message.

When a plain-text message is codified using any suitable scheme**,** the resulting message is called **cipher text**.

There are two primary ways in which a plain-text message can be codified to obtain the corresponding cipher text: substitution and transposition.



Techniques for transforming plain text to cipher text

SUBSTITUTION TECHNIQUES

Caesar Cipher

The scheme explained earlier (of replacing an alphabet with the one three places down the order) was first proposed by Julius Caesar, and is termed **Caesar cipher.** It was the first example of substitution cipher. In the substitution-cipher technique, the characters of a plain-text message are replaced by other characters, numbers or symbols. The Caesar cipher is a special case of substitution technique wherein each alphabet in a **message** is replaced by an alphabet three places down the line. For instance, using the Caesar cipher, the plain-text ATUL will become cipher-text DWXO.

In the substitution-cipher technique, the characters of a plain-text message are replaced by other characters, numbers or symbols.

Clearly, the Caesar cipher is a very weak scheme of hiding plain-text messages. All that is required to break the Caesar cipher is to do the reverse of the Caesar cipher process i.e. replace each alphabet in a cipher-text message produced by Caesar cipher with the alphabet that is three places up the line. Thus, to work backwards, take a cipher text produced by Caesar cipher, and replace each **A** with **X**, **B** with **Y**, **C** with **Z**, **D** with **A**, **E** with **B** and so on.

Modified Version of Caesar Cipher

The Caesar cipher is good in theory, but not so good in practice. Let us now try and complicate the Caesar cipher to make an attacker's task difficult. How can we generalize Caesar cipher a bit more? Let us assume that the cipher-text alphabets corresponding to the original plain-text alphabets may not necessarily be three places down the order, but instead, can be *any* places down the order. This can complicate matters a bit.

Thus, we are now saying that an alphabet A in plain text would not necessarily be replaced by D. It can be replaced by any valid alphabet, i.e. by E or by F or by G, and so on. Once the replacement scheme is decided, it would be constant and will be used for all other alphabets in that message. As we know, the English language contains 26 alphabets. Thus, an alphabet A can be replaced by any *other* alphabet in the English alphabet set, (i.e. B through Z). Of course, it does not make sense to replace an alphabet by itself (i.e. replacing A with A). Thus, for each alphabet, we have 25 possibilities of replacement. Hence, to break a message in the modified version of Caesar cipher, our earlier algorithm would not work.

Mono-alphabetic Cipher

The major weakness of the Caesar cipher is its predictability. Once we decide to replace an alphabet in a plain-text message with an alphabet that is *k* positions up or down the order, we replace all other alphabets in the plain-text message with the same technique. Thus, the cryptanalyst has to try out a maximum of 25 possible attacks, and he/she is assured of success.

Now imagine that rather than using a uniform scheme for all the alphabets in a given plain-text message, we decide to use random substitution. This means that in a given plain-text message, each A can be replaced by any other alphabet (B through Z), each B can also be replaced by any other random alphabet (A or C through Z), and so on. The crucial difference being, there is no relation between the replacement of B and replacement of A. That is, if we have decided to replace each A with D, we need not necessarily replace each B with E—we can replace each B with any other character!

To put it mathematically, we can now have any permutation or combination of the 26 alphabets, which means (26 x 25 x 24 x 23 x ... 2) or 4 x 1026 possibilities! This is extremely hard to crack. It might actually take years to try out these many combinations even with the most modem computers.

Mono-alphabetic ciphers pose a difficult problem for a cryptanalyst because it can be very difficult to crack, thanks to the high number of possible permutations and combinations.

Homophonic Substitution Cipher

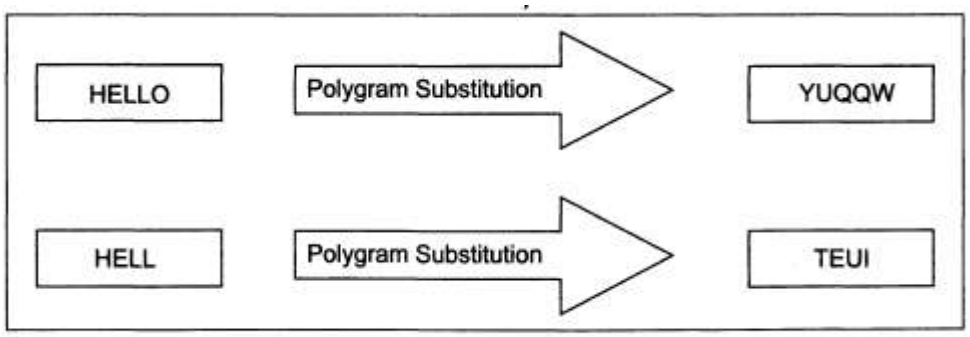
The **homophonic substitution cipher** is very similar to mono-alphabetic cipher. Like a plain substitution cipher technique, we replace one alphabet with another in this scheme. However, the difference between the two techniques is that whereas the replacement alphabet set in case of the simple substitution techniques is fixed (e.g. replace A with D, B with E, etc.), in the case of homophonic substitution cipher, one plain-text alphabet can map to more than one cipher-text alphabet. For instance, A can be replaced by D, H, P, R; B can be replaced by E, I, Q, S, etc.

Homophonic substitution cipher also involves substitution o f one plain-text character with a cipher-text character at a time, however the cipher-text character can be any one o f the chosen set.

Polygram Substitution Cipher

In the **polygram substitution cipher** technique, rather than replacing one plain-text alphabet with one cipher text alphabet at a time, a block of alphabets is replaced with another block. For instance, HELLO could be replaced by YUQQW, but HELL could be replaced by a totally different cipher text block

TEUI, as shown in Fig. This is true in spite of the first four characters of the two blocks of text (HELL) being the same. This shows that in the polygram substitution cipher, the replacement of plain text happens block by block, rather than character by character.



Polygram substitution

Polygram substitution cipher technique replaces one block of plain text with another block of cipher text—it does not work on a character-by-character basis.

Polyalphabetic Substitution Cipher

Leon Battista invented the **polyalphabetic substitution cipher** in 1568. This cipher has been broken many times, and yet it has been used extensively. The **Vigenere cipher** and the **Beaufort cipher** are examples of polyalphabetic substitution cipher.

This cipher uses multiple one-character keys. Each of the keys encrypts one plain-text character. The first key encrypts the first plain-text character; the second key encrypts the second plain-text character, and so on. After all the keys are used, they are recycled. Thus, if we have 30 one-letter keys, every 30th character in the plain text would be replaced with the same key. This number (in this case, 30) is called the **period** of the cipher.

The main features of polyalphabetic substitution cipher are the following:
(a) It uses a set of related monoalphabetic substitution rules.
(b) It uses a key that determines which rule is used for which transformation.

For example, let us discuss the Vigenere cipher, which is an example of this cipher. In this algorithm, 26 Caesar ciphers make up the mono-alphabetic substitution rules. There is a shifting mechanism, from a count of 0 to 25. For each plain-text letter, we have a corresponding substitution, which we call the *key letter.* For instance, the key value is *e* for a letter with shift as 3.

KEY

PLAIN TEXT

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

The logic for encryption is quite simple. For key letter *p* and plain-text letter *q*, the corresponding cipher-text letter is at the intersection of row titled *p* and column titled *q.* For this very particular case, the cipher text, therefore, would be *F*, based on the above table.

By now, it should be clear that for encrypting a plain-text message, we need a key whose length is equal to that of the plain-text message. Usually, a key that repeats itself is used.

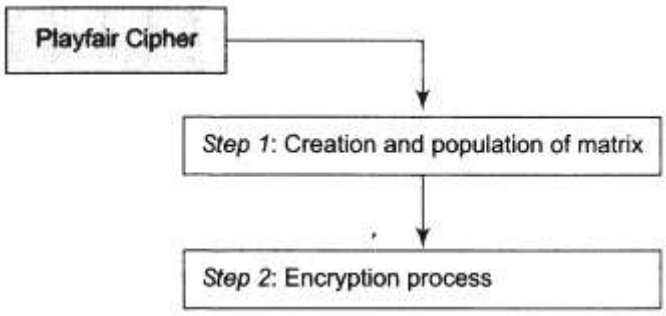
Playfair Cipher

The Playfair cipher, also called Playfair square, is a cryptographic technique used for manual encryption of data. This scheme was invented by Charles Wheatstone in 1854. However, eventually the scheme came to be known by the name of Lord Playfair, who was Wheatstone's friend. Playfair made this scheme popular, and hence his name was used.

The Playfair cipher was used by the British army in World War I and by the Australians in World War II. This was possible because the Playfair cipher is quite fast to use and does not demand any special

equipment to be used. It was used to protect important but not very critical information, so that by the time the cryptanalysts could break it, the value of the information was nullified anyway! In today's world, of course, Playfair cipher would be deemed as an outdated cryptographic algorithm, and rightly so. Playfair cipher now has only academic purpose, except in its usage in some crosswords that appear in several newspapers.

The Playfair encryption scheme uses two main processes.



Step 1: Creation and Population of Matrix

The Playfair cipher makes use of a 5 x 5 matrix (table), which is used to store a *keyword* or *phrase* that becomes the *key* for encryption and decryption. The way this is entered into the 5 x 5 matrix is based on some simple rules

1. Enter the keyword in the matrix row-wise: left-to-right, and then top-to-bottom.
2. Drop duplicate letters.
3. Fill the remaining spaces in the matrix with the rest of the English alphabets (A-Z) that were not a part of our keyword. While doing so, combine I and J in the same cell of the table. In other words, if I or J is a part of the keyword, disregard both I and J while filling the remaining slots.

For example, suppose that our keyword is PLAYFAIR EXAMPLE Then, the 5 x 5 matrix containing our keyword will look as shown

P	L	A	Y	F
I	R	E	X	M
B	C	D	G	H
K	N	O	Q	S
T	U	V	W	Z

EXAMPLE OF ENCRYPTION AND DECRYPTION IN PLAYFAIR

Step:1 Creation and population of matrix (using key)

Let us say, the keyword is "**NUCLEAR ATTACK**"

N	U	C	L	E
A	R	T	K	B
D	F	G	H	I
J	M	O	P	Q
S	V	W	X	Y

Let us say, our original message is "BOMB PLACE IN EUROPE"

Step:2 The encryption process

break the message into pair of 2 letters as:

BO MB PL AC EI NE UR OP EX
TQ QR XK TN BQ UN RF PQ LY

- Before initiating the encryption, break the plain text in pair of 2 letters.
- If both the alphabets are same or 1 letter is remaining, add X after the first alphabet.
- After the initial process, take the pairs for encryption.
- If the letters of the pair appear in same row of the matrix then substitute them with their immediate right letter. If the letter of the plain text is itself the rightmost, then wrap it up with the left letter
- If the letters of the pair appear in same column of the matrix then substitute them with their immediate below letter. If the letter of the plain text is itself below, then wrap it up with the top letter
- If the letter of the pair are not in same row or column then define a rectangle with the original pair and substitute them with other corners of the rectangle.

Hill Cipher

The **Hill cipher** works on multiple letters at the same time. Hence, it is a type of polygraphic substitution cipher. Lester Hill invented this in 1929. The Hill cipher has its roots in the matrix theory of mathematics. More specifically, we need to know how to compute the inverse of a matrix.

WORKING

- Treat each letter with a number like A=0, B=1, C=2……
- Let us say, our original message is "TAJ"
- As per the rule, T=19 A=0 J=9
- Convert into matrix form as

$$\begin{bmatrix} 19 \\ 0 \\ 9 \end{bmatrix}$$

- Now ***multiply the plain text matrix with any number as keys***. The multiplying matrix should be of *n* x *n* where n is the number of rows of original matrix
- $$\begin{bmatrix} 6 & 24 & 1 \\ 13 & 16 & 10 \\ 20 & 17 & 15 \end{bmatrix} \times \begin{bmatrix} 19 \\ 0 \\ 9 \end{bmatrix} = \begin{bmatrix} 123 \\ 337 \\ 515 \end{bmatrix}$$
- Now compute ***mod 26*** on resultant matrix i.e. take the remainder after dividing by 26.
- $$\begin{bmatrix} 123 \\ 337 \\ 515 \end{bmatrix} \bmod 26 = \begin{bmatrix} 19 \\ 25 \\ 21 \end{bmatrix}$$
- Now translating numbers into alphabets, we get:
  19=T 25= Z 21=V
- Therefore our cipher text is ***TZV***

To decrypt hill cipher, follow the steps:

1.) take cipher text matrix and multiply it by inverse of original key matrix

2.) Again perform mod by 26.

Thus we get our original text.

Transposition techniques

Rail – fence technique

- It involves writing plain text as a sequence of diagonals and then reading it row by row to produce cipher text.

Original Message: ATTACK TAJ

Cipher Text : ATCTJTAKA

Simple Columnar Transposition Technique:

- Write the plain text message row by row in a rectangle of pre-defined size
- Read the message column by column but the sequence of columns can be any order.

Original Text : ATTACK ON EUROPE

Col 1	Col 2	Col 3	Col 4	Col 5	Col 6
A	T	T	A	C	K
O	N	E	U	R	O
P	E				

Columns are read in 2,4,6,1,5,3 order
TNEAUKOAOPCRTE

Vernam Cipher (one time pad):

- It uses a one-time pad, which is discarded after a single use and therefore is suitable only for short messages.
- Treat each plain text alphabet with numbers as A=0, B=1, C=2……

Original Message: ATTACKTAJ

Plain text	A	T	T	A	C	K	T	A	J
	0	19	19	0	2	10	19	0	9
+									
One Time Pad (substitute with any letters which are used only ones)	N	B	D	E	P	S	F	Z	L
	13	1	3	4	15	18	5	25	11
Initial Total	13	20	22	4	17	28	24	25	20
Substrcat 26, If >25	13	20	22	4	17	2	24	25	20
Substitute	N	U	W	E	R	C	Y	Z	U

Book Cipher/Running-Key Cipher

The idea used in **book cipher,** also incorrectly called **running-key cipher,** is quite simple, and is similar in principle to the Vernam cipher. For producing cipher text, some portion of text from a book is used, which serves the purpose of a one-time pad. Thus, the characters from a book are used as onetime pad, and they are *added* to the input plain-text message similar to the way a one-time pad works.

Encryption & Decryption

- Encryption or Encoding or Encode
  - The process of converting plain text into cipher text is called as encoding.
- Decryption or Decoding or Decode
  - The process of converting cipher text into plain text is called as decoding.

The important aspects of Encryption & Decryption process are:

- Algorithm
  - The technique/ method used to encrypt or decrypt. Algorithm is generally not kept secret.
- Key
  - A key is a character or a group of characters used to encrypt or decrypt the plain text. A key is generally kept secret.

Depending on what keys are used, there are two types of cryptography mechanisms:-

- **Symmetric Key Cryptography**
  Symmetric key cryptography involves the usage o f the same key for encryption and decryption.
- **Asymmetric Key Cryptography**
  Asymmetric key cryptography involves the usage of one key for encryption, and another, different key for decryption.

Symmetric Key Cryptography and the Problem of Key Distribution

why we need two different types of cryptographic algorithms in the first place. To understand this, let us consider a simple problem statement.

Person A wants to send a highly confidential letter to another person B. A and B both reside in the same city, but are separated by a few miles, and for some reason, cannot meet each other.

Let us now see how we can tackle this problem. The simplest solution would appear to be that A puts the confidential letter in an envelope, seals it, and sends it by post. A hopes that no one opens it before it reaches B.

Clearly, this solution does not seem to be acceptable. What is the guarantee that an unscrupulous person does not obtain and open the envelope before it reaches B? Sending the envelope by registered post or courier might slightly improve the situation, but will not guarantee that the envelope does not get opened before it reaches B. After all, someone can open the envelope, read the confidential letter and re-seal the envelope!

Another option is to send the envelope via a hand-delivery mechanism. Here, A hands the envelope over to another person P, who personally hand-delivers the envelope to B. This seems to be a slightly better solution. However, it is still not foolproof.

Consequently, A comes up with another idea. A now puts the envelope inside a box, seals that box with a highly secure lock, and sends the box to B (through the mechanism of post/courier/hand-delivery). Since the lock is highly secure, nobody can open the box while in transit, and therefore, open the envelope. Consequently, nobody will be able to read/access the highly confidential letter! The problem is resolved! If we think about it, we will realize that the problem indeed seems to be resolved. However, this solution has given birth to a new problem. How on earth can the intended recipient (B) now open the box, and therefore, the envelope? This solution has not only prevented unauthorized access to the letter, but also the authorized access. That is, even B would not be able to open the lock. This defeats the purpose of sending the letter in this manner, in the first place.

What if A also sends the key of the lock along with the box, so that B can open the lock, and get access to the envelope inside the box, and hence the letter? This seems absurd. If the key travels with the box, anybody who has access to the box in transit (e.g. P) can unlock and open the box.

Therefore, A now comes up with an improved plan. A decides that the locked box should travel to B as discussed (by post/courier/hand-delivery). However, she will not send the key used to lock the box along with the box. Instead, she will decide a place and a time to meet B in person, meet B at that time, and hand over the key personally to B. This will ensure that the key does not land up in the wrong hands, and that only B can access the confidential letter! This now seems to be a foolproof solution! Is it, really? If A can meet B in person to hand over the key, she can as well hand the confidential letter to B in person!

Why have all these additional worries and overheads? Remember that the whole problem started because A and B cannot, for some reason, meet in person!

As a result, we will observe that no solution is completely acceptable. Either it is not foolproof, or is not practically possible. This is the problem of key distribution or key exchange. Since the sender and the receiver will use the same key to lock and unlock, this is called *symmetric key operation* (when used in the context of cryptography, this operation is called symmetric key cryptography). Thus, we observe that the key distribution problem is inherently linked with the symmetric key operation.

Let us now imagine that not only A and B but also thousands of people want to send such confidential letters securely to each other. What would happen if they decide to go for symmetric key operation? If we examine this approach more closely, we can see that it has one big drawback if the number of people that want to avail of its services is very large.

We will start with small numbers and then inspect this scheme for a larger number of participants. For instance, let us assume that A now wants to communicate with two persons, B and C, securely. Can A use the same kind of lock (i.e. a lock with the same properties, which can be opened with the same key) and key for sealing the box to be sent to B and C? Of course, this is not advisable at all! After all, if A uses the same kind of lock and key to seal the boxes addressed for B and C, what is the guarantee that B does not open the box intended for C, or vice versa (because B and C would also possess the same key as A)? Even if B and C live in the two extreme comers of the city, A cannot simply take such a chance! Therefore, no matter how secure the lock and key is, **A must use a *different* lock-and-key pair for B and C. This means that A must buy two *different* locks and the corresponding two keys (i.e. one key per lock).**

Parties involved	Number of lock-and-key pairs required
2(A,B)	1 (A-B)
3 (A, B, C)	3 (A-B, A-C, B-C)
4 (A, B, C, D)	6 (A-B, A-C, A-D, B-C, B-D, C-D)
5 (A, B, C, D, E)	10 (A-B, A-C, A-D, A-E, B-C, B-D, B-E, C-D, C-E, D-E)

Therefore, can we see that, in general, for n persons, the number of lock-and-key pairs is $n * (n - l)/2$? Now, if we have about 1,000 persons in this scheme, we will have 1000 * (1000 - l)/2 = 1000 * (999)/2 = 99,9000/2 = 499,500 lock-and-key pairs!

In symmetric key cryptography key sharing is a big issue due to sending key directly or indirectly will lead to leak of key to an hacker.

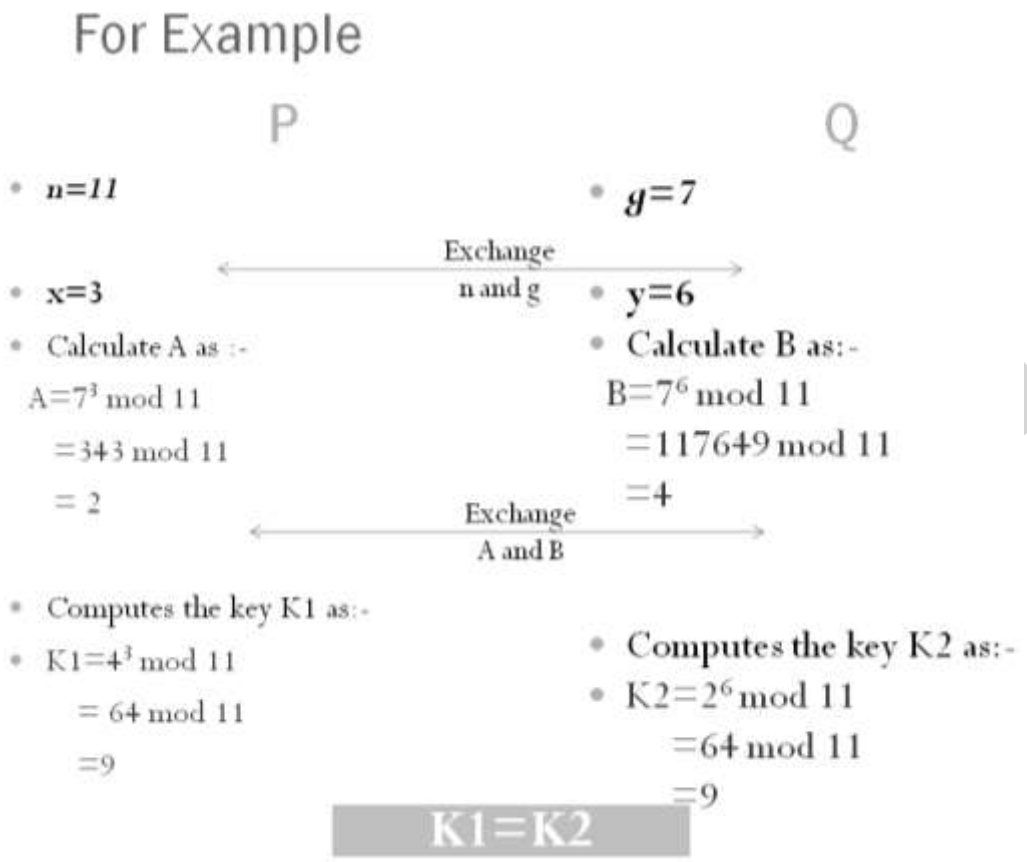
Diffie-Hellman Key-Exchange/Agreement Algorithm

Whitefield Diffie and Martin Hellman devised an amazing solution to the problem of key agreement, or key exchange, in 1976. This solution is called the Diffie-Hellman key-exchange/agreement algorithm. The beauty of this scheme is that the two parties, who want to communicate securely, can agree on a symmetric key using this technique. This key can then be used for encryption/decryption. However, we must note that the Diffie-Hellman key exchange algorithm can be used only for key agreement, but not for encryption or decryption of messages. Once both the parties agree on the key to be used, they need to use other symmetric key-encryption algorithms for actual encryption or decryption of messages.

Diffie-Hellman Key Exchange Algorithm

P	Q
• Prime number '***n***'	• Prime number '***g***'
← Exchange n and g →	
• Random number '**x**'	• Random number '**y**'
• Calculate A as :- $A=g^x \bmod n$	• Calculate B as:- $B=g^y \bmod n$
← Exchange A and B →	
• Computes the key K1 as:- $K1=B^x \bmod n$	• Computes the key K2 as:- $K2=A^y \bmod n$

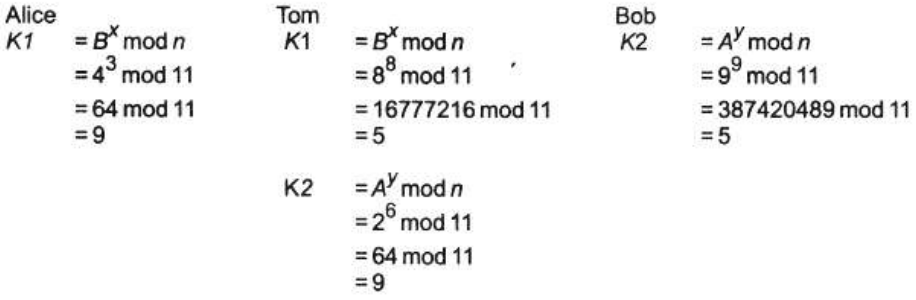
K1=K2



Problems with the Algorithm

Can we now consider that the Diffie-Hellman key-exchange algorithm solves all our problems associated with key exchange? Unfortunately, not quite!

The Diffie-Hellman key-exchange algorithm can fall pray to the **man-in-the-middle attack** (or to be politically correct, **woman-in-the-middle attack**), also called **bucket-brigade attack.** The name *bucketbrigade attack* comes from the way firefighters of yesteryears formed a line between the fire and water source, and passed full buckets towards the fire and the empty buckets back.



In this example Tom intercept in between and acts as a middle man, and sharing if keys is actually done between Alice and Tom and Bob and Tom.

Asymmetric Key Operation

Public-key cryptography, also known as asymmetric cryptography, is a class of cryptographic algorithms which require two separate keys, one of which is secret (or private) and one of which is public. Although different, the two parts of this key pair are mathematically linked. The public key is used to encrypt plaintext or to verify a digital signature; whereas the private key is used to decrypt ciphertext or to create a digital signature.

STEGANOGRAPHY

Steganography is a technique that facilitates hiding of a message that is to be kept secret inside other messages. This results in the concealment of the secret message itself! Historically, the sender used methods such as invisible ink, tiny pin punctures on specific characters, minute variations between handwritten characters, pencil marks on handwritten characters, etc.

Of late, people hide secret messages within graphic images. For instance, suppose that we have a secret message to send. We can take another image file and we can replace the last two rightmost bits of each byte of that image with (the next) two bits of our secret message. The resulting image would not look too different, and yet carry a secret message inside! The receiver would perform the opposite trick: it would read the last two bits of each byte of the image file, and reconstruct the secret message.

KEY RANGE AND KEY SIZE

The encrypted messages can be attacked, too! Here, the cryptanalyst is armed with the following information:

- The encryption/decryption algorithm
- The encrypted message
- Knowledge about the key size (e.g. the value of the key is a number between 0 and 100 billion)

the encryption/decryption algorithm is usually not a secret—everybody knows about it. Also, one can access an encrypted message by various means (such as by listening to the flow of information over a

network). Thus, only the actual value of the key remains a challenge for the attacker. If the key is found, the attacker can resolve the mystery by working backwards to the original plain-text message

brute-force attack here, which works on the principle of trying every possible key in the key range, until you get the right key.

It usually takes a very small amount of time to try a key. The attacker can write a computer program that tries many such keys in one second. In the best case, the attacker finds the right key in the first attempt itself, and in the worst case, it is the 100 billionth attempt. However, the usual observation is that the key is found somewhere in between the possible range. Mathematics tells us that on an average, the key can be found after about half of the possible values in the key range are checked. Of course, this is just a guideline, and may or may not work in real practice for a given situation.

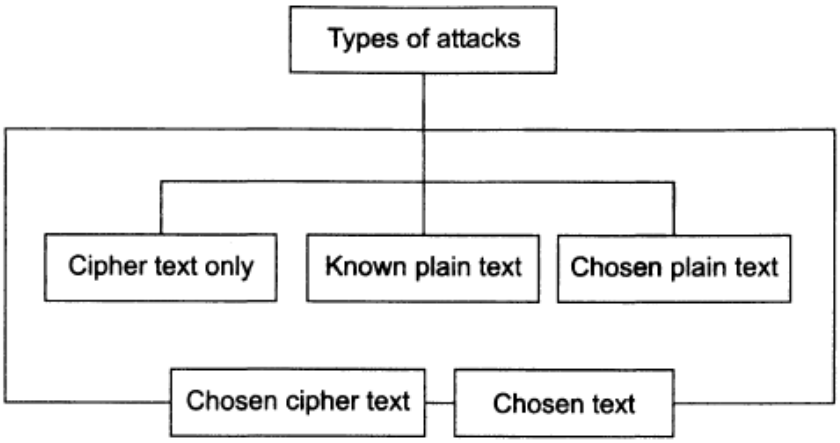
In order to protect ourselves against a brute-force attack, the key size should be such that the attacker cannot crack it within a specified amount of time.

Efforts required to break a key

Key size on bits	Time required to search 1 percent of the key space	Time required to search 50 percent of the key space
66	1 second	1 minute
57	2 seconds	2 minutes
58	4 seconds	4 minutes
64	4.2 minutes	4.2 hours
72	17.9 hours	44.8 days
80	190.9 days	31.4 years
90	535 years	321 centuries
128	146 billion millennia	8 trillion millennia

POSSIBLE TYPES OF ATTACKS

Based on the discussion so far, when the sender of a message encrypts a plain-text message into its corresponding cipher text, there are five possibilities for an attack on this message.



Cipher-Text Only Attack

The attacker analyzes the cipher text at leisure to try and figure out the original plain text. Based on the frequency of letters (e.g. the alphabets e, i, a are very common in English, etc.) the attacker makes an attempt to guess the plain text. Obviously, the more cipher text available to the attacker, more are the chances of a successful attack.

Known Plain-Text Attack

attacker knows about s()me pairs of plain text and corresponding cipher text for those pairs. Using this information, the attacker tries to find other pairs, and therefore, know more and more of the plain text. Examples of such *known plain texts* are company banners, file headers, etc., which are found commonly in all the documents of a particular company. How can the attacker obtain the plain text, in the first place? This can happen because plain-text information may become outdated over time, and hence, become public knowledge.

Chosen Plain-Text Attack

the attacker selects a plain-text block, and tries to look for the encryption of the same in the cipher text. Here, the attacker is able to choose the messages to encrypt. Based on this, the attacker intentionally picks patterns of cipher text that result in obtaining more information about the key.

Chosen Cipher-Text Attack

In the **chosen cipher-text attack,** the attacker knows the cipher text to be decrypted, the encryption algorithm that was used to produce this cipher text, and the corresponding plain-text block. The attacker's job is to discover the key used for encryption. However, this type of attack is not very commonly used.

Chosen-Text Attack

The **chosen-text attack** is essentially a combination of *chosen plain-text attack* and *chosen cipher-text attack.*

Summary of types of attacks

Attack	**Things known to the attacker**	**Things the attacker wants to find out**
Cipher-text only	• Cipher text of several messages, all of which are encrypted with the same encryption key. • Algorithm used	• Plain text messages corresponding to these cipher text messages • Key used for encryption
Known cipher text	• Cipher text of several messages, all of which are encrypted with the same encryption key. • Plain text messages corresponding to above cipher text messages • Algorithm used	• Key used for encryption • Algorithm to decrypt cipher text with the same key
Chosen plain text	• Cipher text and associated plain text messages • Chooses the plain text to be encrypted	• Key used for encryption • Algorithm to decrypt cipher text with the same key
Chosen cipher text	• Cipher text of several messages to be decrypted • Corresponding plain text messages	• Key used for encryption
Chosen text	• Some of the above	Some of the above

UNIT 2

COMPUTER BASED SYMMETRIC KEY CRYPTOGRAPHY ALGORITHM

ALGORITHM TYPES AND MODES:-

ALGORITHM TYPES: -

There are two basic algorithm types:- (1) stream ciphers and (2) block cipher text

STREAM CIPHERS: -

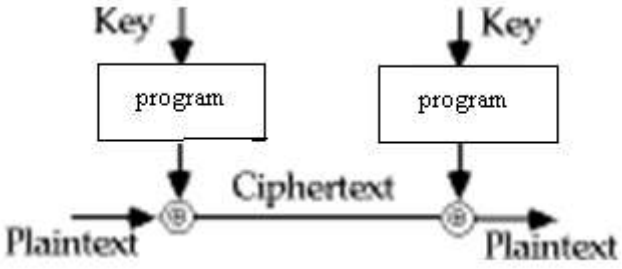
- In stream ciphers, the plain text is encrypted one bit at a time.
- Suppose the original message (plain text) is *Pay 100* in ASCII (i.e. text format).
- When we convert these ASCII characters to their binary values, let us assume that it translates to 01011100 Let us also Assume that we apply the *XOR* logic as the encryption algorithm:-

Input 1	Input 2	Outputs
0	0	0
0	1	1
1	0	1
1	1	0

- We can see the effect of XOR:-

In text format	In binary format	
Pay 100	010111001	Plaintext
	100101011	XOR operation with the key
ZTU91 A%D	11001001	Cipher text

- As a result of applying one bit of key for every respective bit of the original message, suppose the cipher text is generated as 11001001 in binary (ZTU91 A% in text).



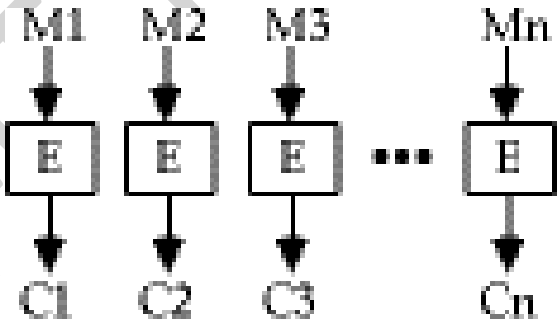
- Stream-cipher technique involves the encryption of one plain-text bit at a time. The decryption also happens one bit at a time.
- Another interesting property of XOR is that when used twice, it produces the original data.
- For example, suppose we have two binary numbers A = 101 and B = 110. We now want to perform an XOR operation on A and B to produce a third number C, i.e.

C = A XOR B
Thus, we will have
C = 101 XOR 110 = 011
Now, if we perform C XOR A, we will get B.
That is,
C = 011 XOR 101 = 110
Similarly, if we perform C XOR B, we will get A.
That is,
A = 011 XOR 110 = 101

- This reversibility of XOR operations has many implications in cryptographic algorithms, as we shall notice. XOR is reversible—when used twice, it produces the original values. This is useful in cryptography.

BLOCK CIPHERS: -

- In block ciphers, rather than encrypting one bit at a time, a block of bits is encrypted at one go.
- Suppose we have a plain text FOUR AND FOUR that needs to be encrypted. Using block cipher, FOUR could be encrypted first, followed by _AND_ and finally FOUR.
- Thus, one block of characters gets encrypted at a time.



- During decryption, each block would be translated back to the original form.
- In actual practice, the communication takes place only in bits. Therefore, FOUR actually means the binary equivalent of the ASCII characters FOUR.
- After any algorithm encrypts these, the resultant bits are converted back into their ASCII equivalents. Therefore, we get funny symbols such as Vfa%9 etc.
- In actual practice, their binary equivalents are received, which are decrypted back into binary equivalent of ASCII FOUR.

- The block-cipher technique involves encryption o f one block of text at a time. Decryption also takes one block of encrypted text at a time.
- Practically, the blocks used in a block cipher generally contain 64 bits or more.
- As we have seen, stream ciphers encrypt one bit at a time. This can be very time-consuming and actually unnecessary in real life.
- That is why block ciphers are used more often in computer-based cryptographic algorithms as compared to stream ciphers.

GROUP STRUCTURES :-

- When discussing an algorithm, many times a question arises, as to whether it is a group.
- The elements of the group are the cipher-text blocks with each possible key.
- Grouping, thus, means how many times the plain text is scrambled in various ways to generate the cipher text.

CONCEPTS OF CONFUSION AND DIFFUSION:-

- Claude Shannon introduced the concepts of confusion and diffusion, which are significant from the perspective of the computer-based cryptographic techniques.
- Confusion is a technique of ensuring that a cipher text gives no clue about the original plain text.
- This is to try and thwart the attempts of a cryptanalyst to look for patterns in the cipher text, so as to deduce the corresponding plain text.
- We already know how to achieve confusion: it is achieved by means of the substitution techniques discussed earlier.
- Diffusion increases the redundancy of the plain text by spreading it across rows and columns.
- We have already seen that this can be achieved by using the transposition techniques (also called permutation techniques).
- Stream cipher relies only on confusion. Block cipher uses both confusion and diffusion.

ALGORITHM MODES:-

There are four important algorithm modes, namely Electronic Code Book (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), and Output Feedback (OFB).

Electronic Code Book (ECB) Mode: -

- Electronic Code Book (ECB) is the simplest mode of operation.
- Here, the incoming plain-text message is divided into blocks of 64 bits each.
- Each such block is then encrypted independently of the other blocks.
- For all blocks in a message, the same key is used for encryption.
- This process is shown:-

Plain-text block 1	**Plain-text block 2**	**Plain-text block *n***

I	I	I
Key -→Encrypt	Key —>Encrypt	Key —► Encrypt
Cipher-text block 1	Cipher-text block 2	Cipher-text block *n*

- At the receiver's end, the incoming data is divided into 64-bit blocks, and by using the same key as was used for encryption, each block is decrypted to produce the corresponding plain-text block.
- This process is shown:-

Cipher-text block 1	**Cipher-text block 2**	**Cipher-text block *n***
Key Decrypt	Key —► Decrypt	Key —>- Decrypt
Plain-text block 1	Plain-text block 2	Plain-text block *n*

- In ECB, since a single key is used for encrypting all the blocks of a message, if a plain-text block repeats in the original message, the corresponding cipher-text block will also repeat in the encrypted message.
- Therefore, ECB is suitable only for encrypting small messages, where the scope for repeating the same plain-text blocks is quite less.

Cipher Block Chaining (CBC) Mode: -

- We saw that in the case of ECB, within a given message (i.e. for a given key), a plain-text block always produces the same cipher-text block.
- Thus, if a block of plain text occurs more than once in the input, the corresponding cipher text block will also occur more than once in the output, thus providing some clues to a cryptanalyst.
- To overcome this problem, the Cipher Block Chaining (CBC) mode ensures that even if a block of plain text repeats in the input, these two (or more) identical plain-text blocks yield totally different cipher-text blocks in the output.
- For this, a feedback mechanism is used.
- Chaining adds a feedback mechanism to a block cipher.
- In Cipher Block Chaining (CBC), the results of the encryption of the previous block are fed back into the encryption of the current block.
- That is, each block is used to modify the encryption of the next block.
- Thus, each block of cipher text is dependent on the corresponding current input plain-text block, as well as all the previous plain-text blocks.

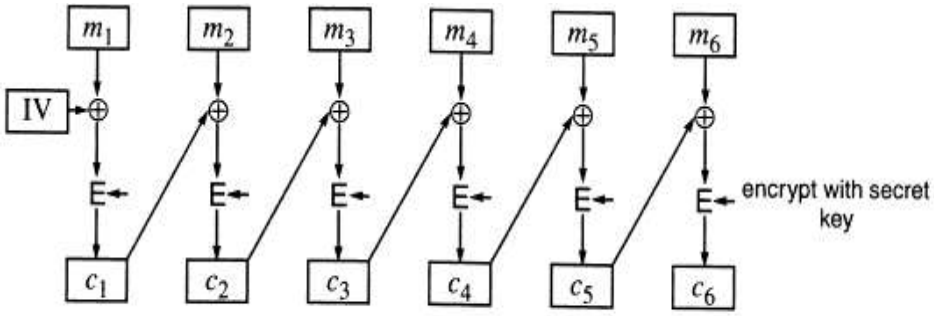


Figure 4-5. Cipher Block Chaining Encryption

Decryption is simple because ⊕ is its own inverse.

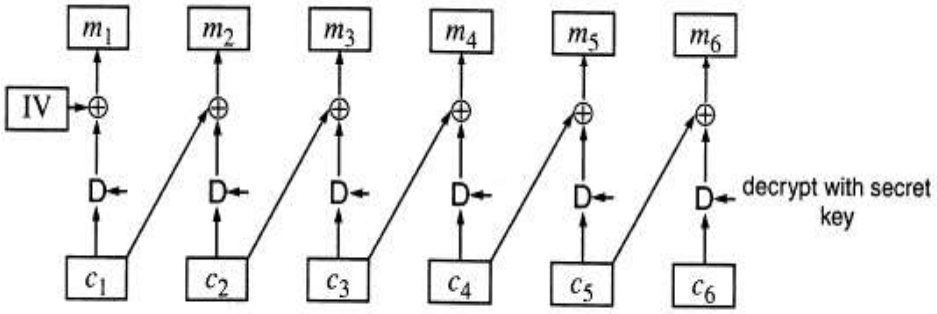
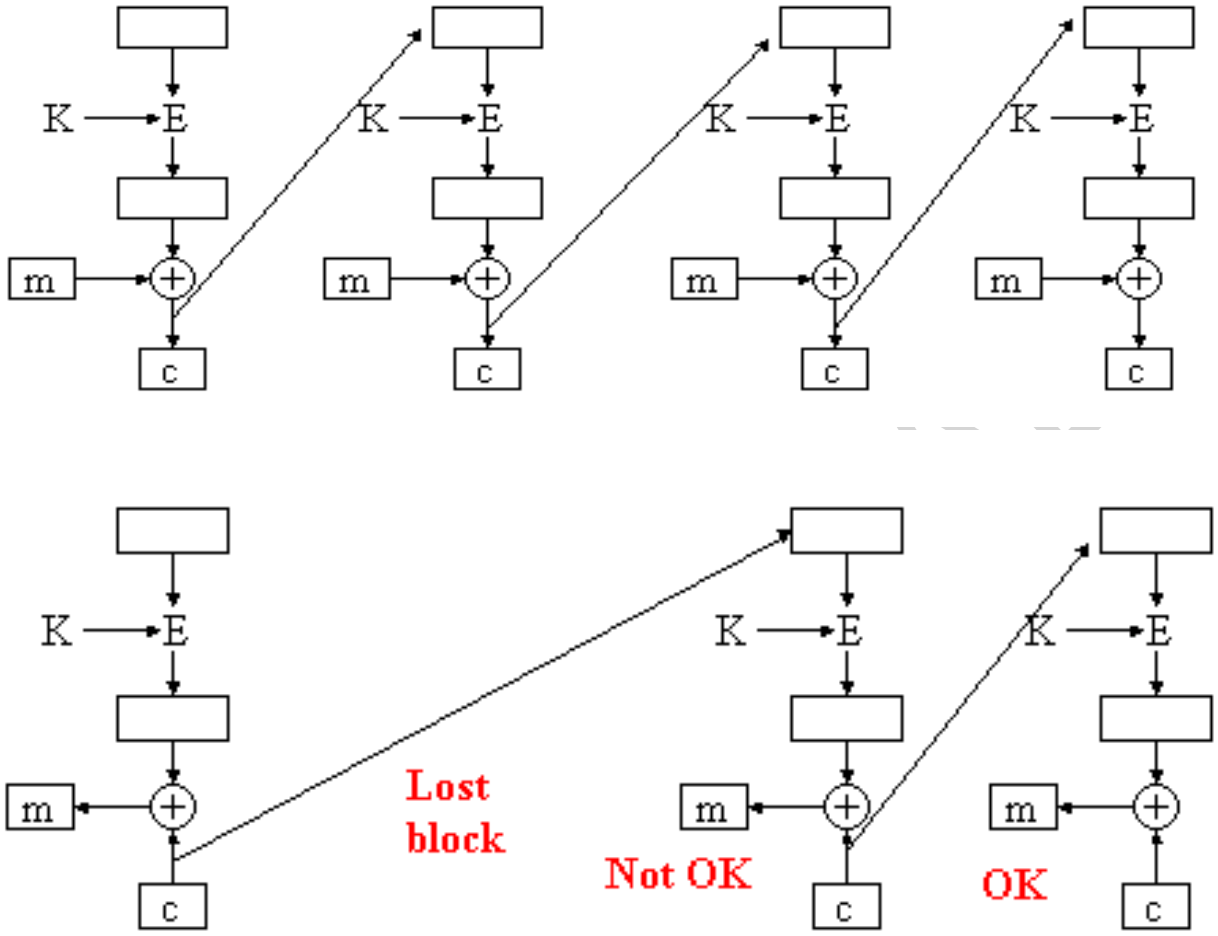


Figure 4-6. Cipher Block Chaining Decryption

Cipher Feedback (CFB) Mode: -

- Not all applications can work with blocks of data. Security is also required in applications that are character-oriented.
- For instance, an operator can be typing keystrokes at a terminal, which needs to be immediately transmitted across the communications link in a secure manner, i.e. by using encryption.
- In such situations, stream cipher must be used. The Cipher Feedback (CFB) mode is useful in such cases.
- In this mode, data is encrypted in units that are smaller (e.g. they could be of size 8 bits, i.e. the size of a character typed by an operator) than a defined block size (which is usually 64 bits).



- Let us understand how CFB mode works, assuming that we are dealing with *j* bits at a time (as we have seen usually, but not always, *j* = 8). Since CFB is slightly more complicated as compared to the first two cryptography modes, we shall study CFB in a step-by-step fashion.

  Step 1 Like CBC, a 64-bit Initialization Vector (IV) is used in the case of CFB mode. The IV is kept in a shift register. It is encrypted in the first step to produce a corresponding 64 bit cipher text.

  Step 2 Now, the leftmost (i.e. the most significant) *j* bits of the encrypted IV are XORed with the first *j* bits of the plain text.

  Step 3 Now, the bits of IV (i.e. the contents of the shift register containing IV) are shifted left by j positions. Thus, the rightmost *j* positions of the shift register now contain unpredictable data. These rightmost *j* positions are now filled with C.

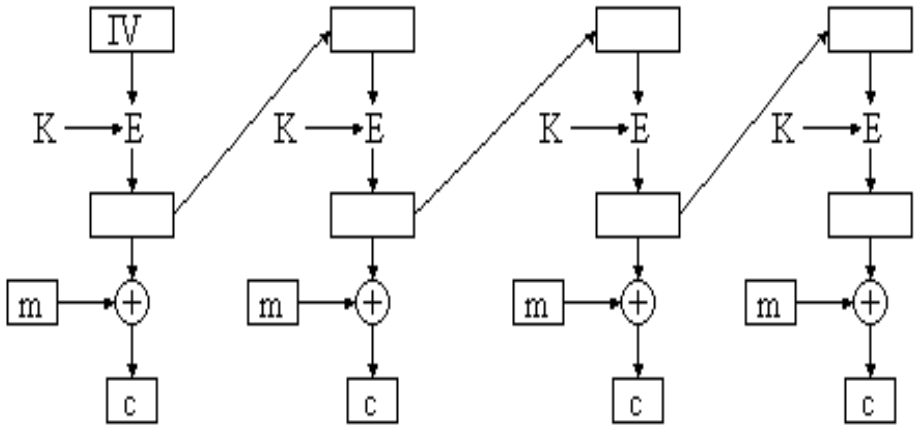
  Step 4 Now, steps 1 through 3 continue until all the plain-text units are encrypted. That is, the following steps are repeated:

- IV is encrypted.
- The leftmost *j* bits resulting from this encryption process are XORed with the next *j* bits of the plain text.
- The resulting cipher-text portion (i.e. the next *j* bits of cipher text) is sent to the receiver.

- The shift register containing the IV is left-shifted by *j* bits.
- The *j* bits of the cipher text are inserted from right into the shift register containing the IV.

Output Feedback (OFB) Mode: -

- The Output Feedback (OFB) mode is extremely similar to the CFB.
- The only difference is that in the case of CFB, the cipher text is fed into the next stage of encryption process.
- But in the case of OFB, the output of the IV encryption process is fed into the next stage of encryption process.
- In simple terms, we can state that in this mode, if there are errors in individual bits, they remain errors in individual bits and do not corrupt the whole message.
- That is, bit errors do not get propagated.
- If a cipher-text bit C is in error, only the decrypted value corresponding to this bit, i.e. P. is wrong.
- Other bits are not affected.
- The possible drawback with OFB is that an attacker can make necessary changes to the cipher text and the checksum of the message in a controlled fashion.
- This causes changes in the cipher text without it getting detected.
- In other words, the attacker changes both the cipher text and the checksum at the same time, hence there is no way to detect this change.



- Major advantages of OFB:

  The pad can be *generated in advance* and used when the message arrive. If some bits of cipher text get garbled, only the corresponding bits in the plain text get garbled.

- Major disadvantages of OFB:

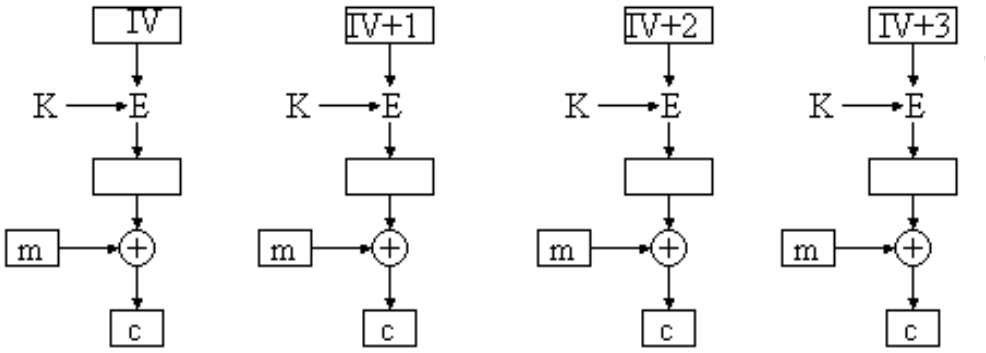
  - ✓ if the plaintext P, ciphertext (P®K) > are known by Trudy, she can modify the plain text into anything she wants (P') since: ((P ® K) ® P) ® P') ® K = P'
  - ✓ If one block is lost, the rest of the blocks will be garbled.

- ✓ If data is stored on disk, you can not randomly read any block unless you decrypt all the preceding blocks.

- To solve the last two problems, we use CFB below, where if one block is lost, only the next block is garbled and the rest of the blocks will decrypt properly.

Counter (CTR) Mode: -

- The Counter (CTR) mode is quite similar to the OFB mode, with one variation.
- It uses *sequence numbers* called *counters* as the inputs to the algorithm.
- After each block is encrypted, to fill the register, the next counter value is used.
- Usually, a constant is used as the initial counter value, and is incremented (usually by 1) for every iteration.
- The size of the counter block is the same as that of the plain-text block.

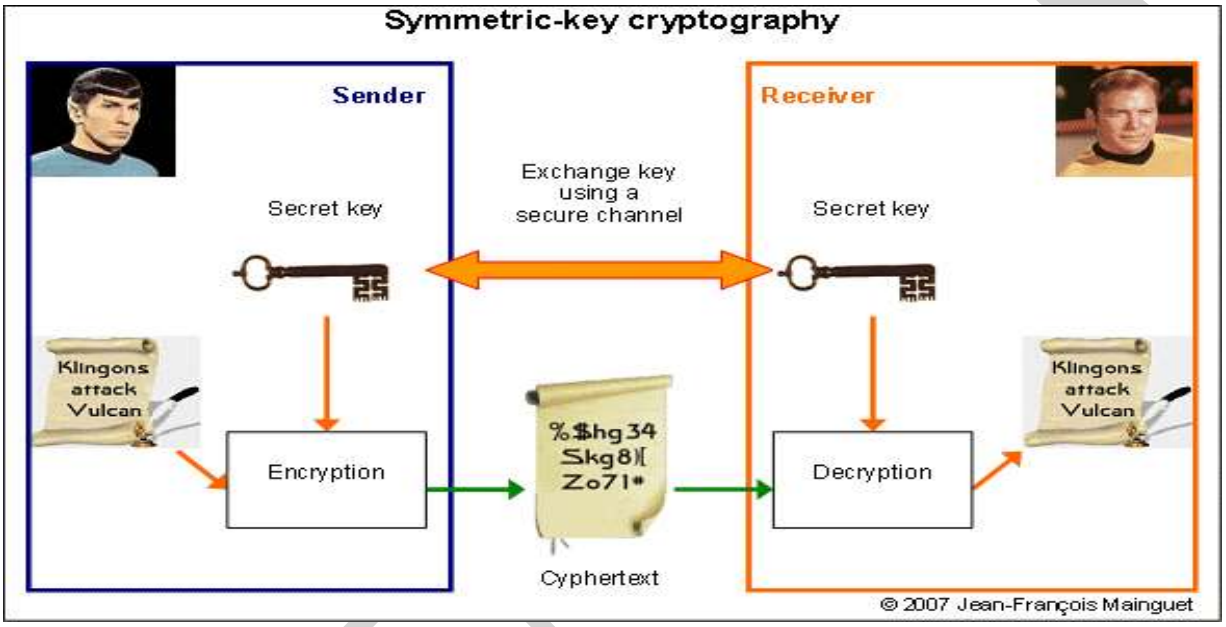


- For encryption, the counter is encrypted and then XORed with the plain text block to get the cipher text.
- No chaining process is used. On the other hand, for decryption, the same sequence of counters is used.
- Here, each encrypted counter is XORed with the corresponding cipher-text block to obtain the original plain-text block.
- The encryption or decryption process in the counter can be done in parallel on multiple text blocks, since no chaining is involved.
- This can make the execution speed of the counter faster.
- Multiprocessing systems can take advantage of this feature to introduce features that help reduce the overall processing time.
- Pre-processing can be achieved to prepare the output of the encryption boxes that input to the XOR operations.
- The counter mode mandates implementation of the encryption process only, and not of the decryption process.

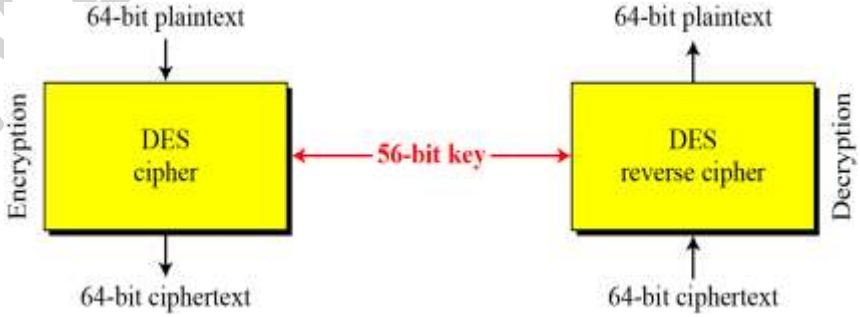
SYMMETRIC KEY CRYPTOGRAPHY:-

- An encryption system in which the sender and receiver of a message share a single, common key that is used to encrypt and decrypt the message.

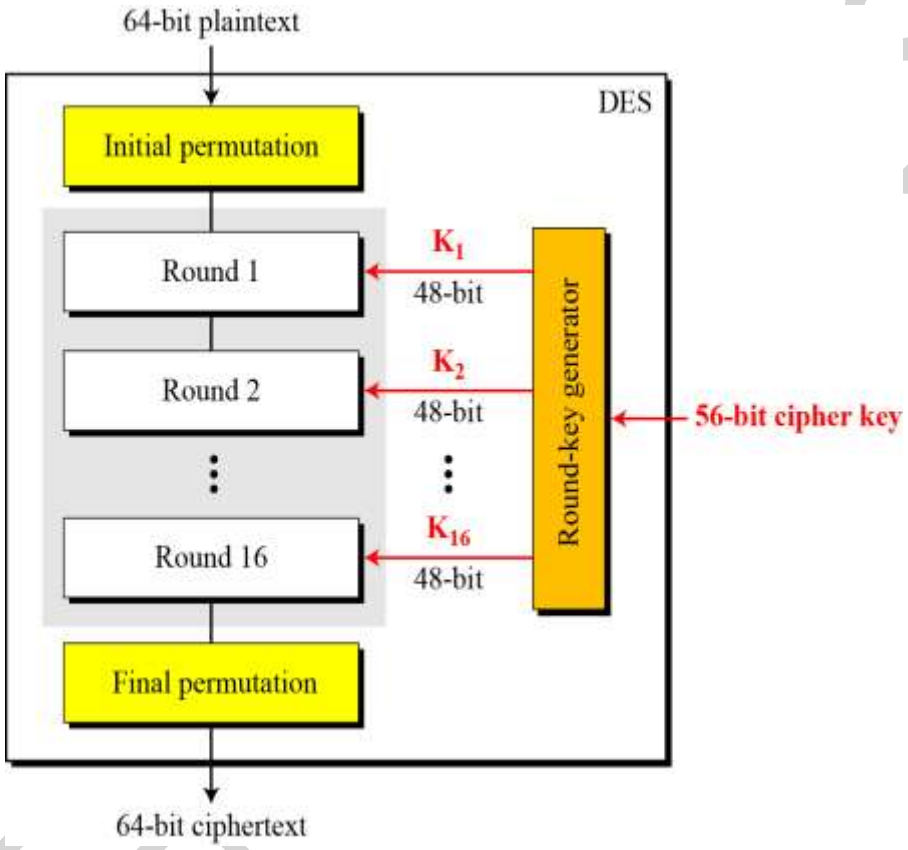
- Contrast this with public-key cryptology, which utilizes two keys - a public key to encrypt messages and a private key to decrypt them.
- Symmetric-key systems are simpler and faster, but their main drawback is that the two parties must somehow exchange the key in a secure way.
- Public-key encryption avoids this problem because the public key can be distributed in a non-secure way, and the private key is never transmitted.
- Symmetric-key cryptography is sometimes called *secret-key cryptography*. The most popular symmetric-key system is the *Data Encryption Standard (DES)*.



DATA ENCRYPTION STANDARD: -



General structure of DES: -

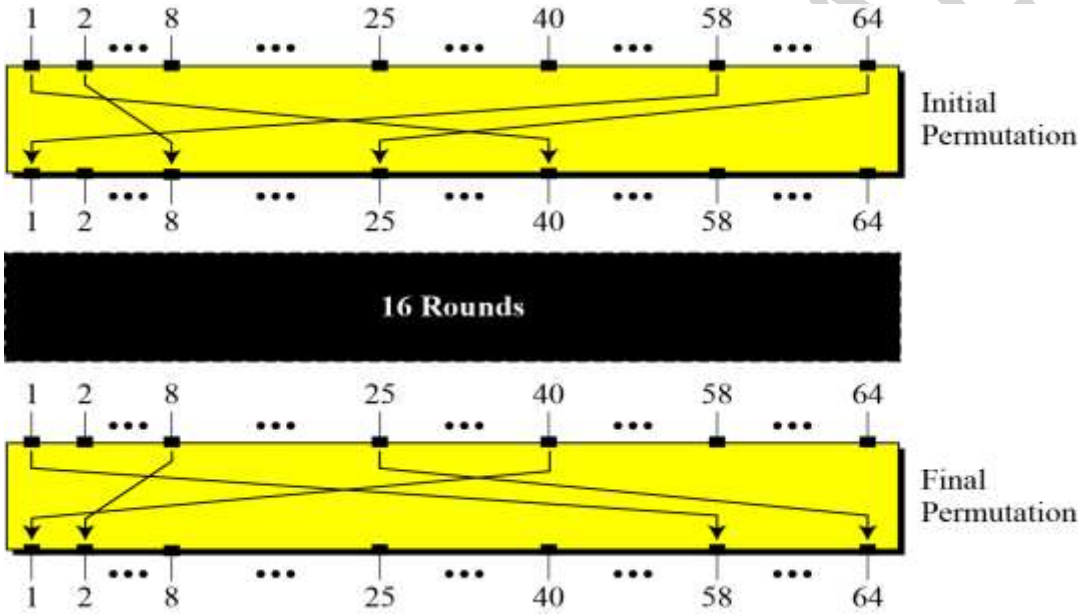


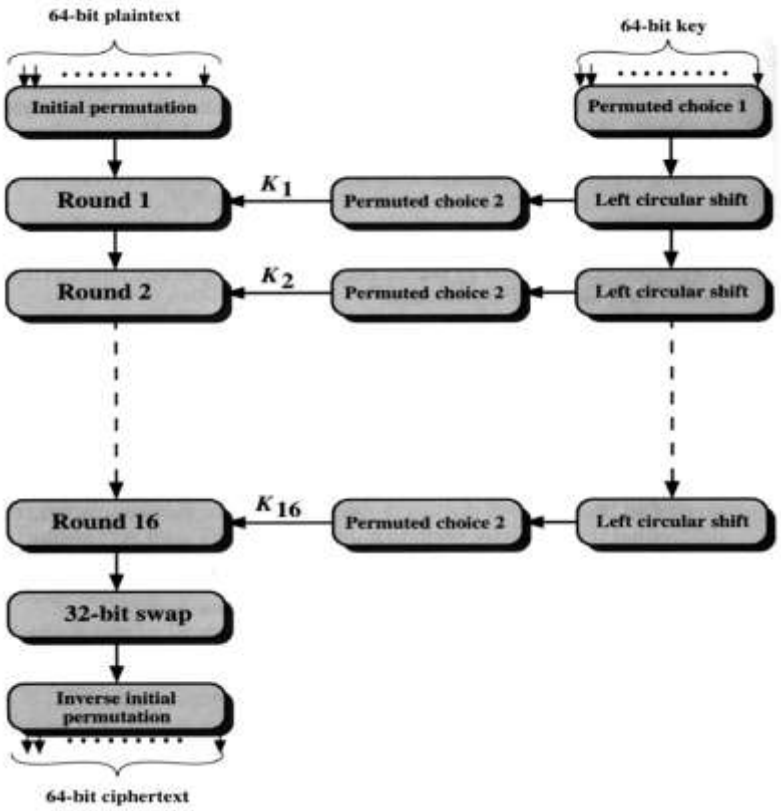
- DES is a block cipher.
- It encrypts data in blocks of 64 bits each.
- That is, 64 bits of plain text goes as the input to DES, which produces 64 bits of cipher text.
- The same algorithm and key are used for encryption and decryption, with minor differences.
- The key length is 56 bits.
- Actually, the initial key consists of 64 bits.
- However, before the DES process even starts, every eighth bit of the key is discarded to produce a 56-bit key.
- That is, bit positions 8, 16, 24, 32, 40, 48, 56 and 64 are discarded.
- Thus, the discarding of every 8th bit of the key produces a 56-bit key from the original 64-bit key.

Broad-level steps in DES: -

1. In the first step, the 64-bit plain-text block is handed over to an Initial Permutation (IP) function.
2. The initial permutation is performed on plain text.
3. Next, the Initial Permutation (IP) produces two halves of the permuted block; say Left Plain Text (LPT) and Right Plain Text (RPT).
4. Now, each of LPT and RPT go through 16 *rounds* of encryption process, each with its own key.
5. In the end, LPT and RPT are rejoined, and a Final Permutation (FP) is performed on the combined block.
6. The result of this process produces 64-bit cipher text.

Initial and final permutation steps in DES





Initial and final permutation table:-

Initial Permutation	*Final Permutation*
58 50 42 34 26 18 10 02	40 08 48 16 56 24 64 32
60 52 44 36 28 20 12 04	39 07 47 15 55 23 63 31
62 54 46 38 30 22 14 06	38 06 46 14 54 22 62 30
64 56 48 40 32 24 16 08	37 05 45 13 53 21 61 29
57 49 41 33 25 17 09 01	36 04 44 12 52 20 60 28
59 51 43 35 27 19 11 03	35 03 43 11 51 19 59 27
61 53 45 37 29 21 13 05	34 02 42 10 50 18 58 26
63 55 47 39 31 23 15 07	33 01 41 09 49 17 57 25

Initial Permutation (IP)

- It happens only once, and it happens before the first round.
- It suggests how the transposition in IP should proceed.
- For example, it says that the IP replaces the first bit of the original plain-text block with the 58th bit of the original plain-text block, the second bit with the 50th bit of the original plain text block, and so on.

Rounds : -

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 of key bits shifted	1 1 2 2 2 2 2 2 1 2 2 2 2 2 2 2 1

- The initial 64-bit key is transformed into a 56-bit key by discarding every 8th bit of the initial key.
- Thus, for each round, a 56-bit key is available.
- From this 56-bit key, a different 48-bit subkey is generated during each *round* using a process called key transformation.
- For this, the 56-bit key is divided into two halves, each of 28 bits.
- These halves are circularly shifted left by one or two positions, depending on the round.
- For example, if the *round* number is 1,2, 9 or 16, the shift is done by only one position.
- For other rounds, the circular shift is done by two positions.
- After an appropriate shift, 48 of the 56 bits are selected.
- For selecting 48 of the 56 bits, the table is shown below.
- For instance, after the shift, bit number 14 moves into the first position, bit number 17 moves into the second position, and so on.
- If we observe the table carefully, we will realize that it contains only 48 bit positions. Bit number 18 is discarded (we will not find it in the table), like 7 others, to reduce the 56-bit key to a 48-bit key.
- Since the key-transformation process involves permutation as well as selection of a 48-bit subset of the original 56-bit key, it is called compression permutation.

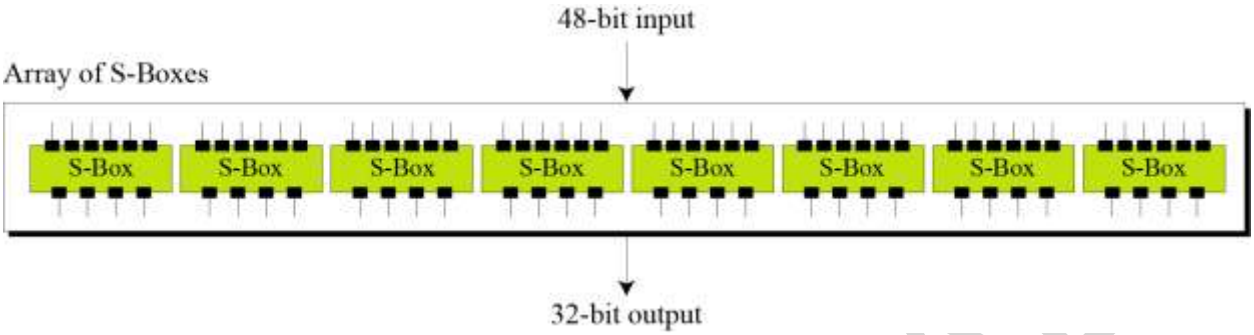
14	17	11	24	1	5	3	28	15	6	21	10
23	19	12	4	26	8	16	7	27	20	13	2
41	52	31	37	47	55	30	40	51	45	33	48
44	49	39	56	34	53	46	42	50	36	29	32

- Because of this compression permutation technique, a different subset of key bits is used in each round. That makes DES more difficult to crack.
- During expansion permutation, the RPT is expanded from 32 bits to 48 bits.
- Besides increasing the bit size from 32 to 48, the bits are permuted as well, hence the name *expansion permutation.* This happens as follows:

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

S-box substitution

- The S-boxes do the real mixing (confusion). DES uses 8 S-boxes, each with a 6-bit input and a 4-bit output



- It is a process that accepts the 48-bit input from theXOR operation involving the compressed key and expanded RPT, and produces a 32-bit output using the substitution technique.
- The substitution is performed by eight substitution boxes (also called as Sboxes).
- Each of the eight S-boxes has a 6-bit input and a 4-bit output.
- The 48-bit input block is divided into 8 sub-blocks (each containing 6 bits), and each such sub-block is given to an S-box.
- The S-box transforms the 6-bit input into a 4-bit output.

S_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_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_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_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_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
	4	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_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_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_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

- The 6-bit input indicates which row and column, and therefore, which intersection is to be selected, and thus, determines the 4-bit output.
- How is it done? Let us assume that the six bits of an S-box are indicated by bl, b2, b3, b4, b5, and b6. Now, bits bl and b6 are combined to form a two-bit number. Two bits can store any decimal number between 0 (binary 00) and 3 (binary 11). This specifies the row number. The remaining four bits b2, b3, b4, b5 make up a four-bit number, which specifies the column number between decimal 0 (binary 0000) and 15 (binary 1111). Thus, the 6-bit input automatically selects the row number and column number for the selection of the output.
- Let us take an example now. Suppose the bits 5 to 8 of the 48-bit input (i.e. the input to the second S-box) contain a value 101101 in binary. Therefore, using our earlier diagram, we have (bl, b6) = 11 in binary (i.e. 3 in decimal), and (b2, b3, b4, b5) = 0110 in binary (i.e. 6 in decimal).
- Thus, the output of S-box 2 at the intersection of row number 3 and column number 6 will be selected, which is 4. (Remember to count rows and columns from 0, not 1).
- The output of each S-box is then combined to form a 32-bit block, which is given to the last stage of a *round*, the P-box permutation.

P-box Permutation

- The output of S-box consists of 32 bits.
- These 32 bits are permuted using a P-box.
- This straightforward permutation mechanism involves simple permutation (i.e. replacement of each bit with another bit, as specified in the P-box table, without any expansion or compression).
- This is called P-box permutation.

- For example, a 16 in the first block indicates that the bit at position 16 of the original input moves to the bit at position 1 in the output, and a 10 in the block number 16 indicates that the bit at the position 10 of the original input moves to bit at the position 16 in the output.

16	7	20	21	29	12	28	17	1	15	23	26
2	8	24	14	32	27	3	9	19	13	30	6

XOR and SWAP:-

- The left half portion of the initial 64-bit plain text block (i.e. LPT) is XORed with the output produced by P-box permutation.
- The result of this XOR operation becomes the new right half (i.e. RPT).
- The old right half (i.e. RPT) becomes the new left half, in a process of swapping.

Final Permutation: -

- At the end of the 16 rounds, the final permutation is performed (only once).
- For instance, the 40th input bit takes the position of the 1st output bit, and so on.
- The output of the final permutation is the 64-bit encrypted block.

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

DES Decryption:-

- The same algorithm used for encryption in DES also works for decryption!
- The values of the various tables and the operations as well as their sequence are so carefully chosen that the algorithm is reversible.
- The only difference between the encryption and the decryption process is the reversal of key portions.
- If the original key *K* was divided into *Kl, K2, K3, ..., K16* for the 16 encryption *rounds*, then for decryption, the key should be used as *K16, K l5, K l4, ..., K l.*

VARIATIONS OF DES: -

Two main variations of DES are double DES and triple DES.

Double DES: -

- Essentially, it does twice what DES normally does only once.
- Double DES uses two keys, say *K l* and *K2.* It first performs DES on the original plain text using *K l* to get the encrypted text.
- It again performs DES on the encrypted text, but this time with the other key, i.e. *K2.*
- The final output is the encryption of encrypted text (i.e. the original plain text encrypted twice with two different keys).

- The doubly encrypted cipher-text block is first decrypted using the key *K2* to produce the singly encrypted cipher text.
- This cipher-text block is then decrypted using the key *K l* to obtain the original plain-text block.
- If we use a key of just 1 bit, there are two possible keys (0 and 1).
- If we use a 2-bit key, there are four possible key values (00, 01,10 and 11). In general, if we use an *n*-bit key, the cryptanalyst has to perform 2" operations to try out all the possible keys.
- If we use two different keys, each consisting of *n* bits, the cryptanalyst would need *l 2n* attempts to crack the key.
- Therefore, on the face of it, we may think that since the cryptanalysis for the basic version of DES requires a search of 256 keys, Double DES would require a key search of (22*56), i.e. 2112 keys.
- However, it is not quite true. Merkle and Heilman introduced the concept of the meet-in-the-middle attack.
- This attack involves encryption from one end, decryption from the other, and matching the results in the middle, hence the name *meet-in-the-middle* attack.

- **Let us understand how it works.**

Suppose that the cryptanalyst knows two basic pieces of information: *P* (a plain-text block), and C (the corresponding final cipher-text block) for a message. The result of the first encryption is called 71, and is denoted as *T= EKl* (*P*) [i.e. encrypt the block *P* with key *Kl]*. After this encrypted block is encrypted with another key *K l,* we denote the result as C = *EK2(EKl(P))* [i.e. encrypt the already encrypted block *T,* with a different key *K l,* and call the final cipher text as C].

It works as follows: -

1. The cryptanalyst would encrypt the plain-text block *P* by performing the first encryption operation, i.e. *EK](P).* That is, it will calculate *T.*

2. The cryptanalyst would store the output of the operation *EKX(P*), i.e. the temporary cipher text (7), in the next available row of the table in the memory.

3. In the first step, the cryptanalyst was calculating the value of *T* from the left-hand side (i.e. encrypt *P* with *K* to find 7). Thus, here *T = EKX(P).*

4. In the second step, the cryptanalyst was finding the value of *T* from the right-hand side (i.e. decrypt C with *K l* to find 7). Thus, here *T= DK2(Q).*

Clearly, this attack is possible, but requires a lot of memory which is too high for the next few generations of computers.

Triple DES: -

It. comes in two kinds, one that uses three keys, and the other that uses two keys.

(a) Triple DES with Three Keys: -

- The plain-text block *P* is first encrypted with a key *K l* , then encrypted with a second key *K l,* and finally with a third key, *K3,* where *K l* , *K l* and *K3* are all different from each other.
- Triple DES with three keys is used quite extensively in many products, including PGP and S/MIME.
- To decrypt the cipher text *C* and obtain the plain text *P,* we need to perform the operation
  P = DKl(D J.D J.Q)).

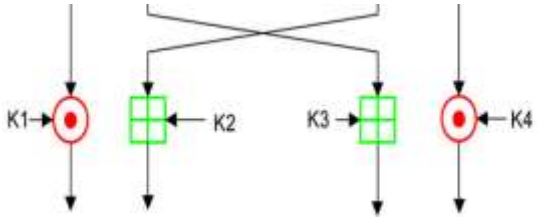
(b) Triple DES with Two Keys: -

- Triple DES with three keys is highly secure.
- It can be denoted in the form of an equation as C = *EK2(EK2(EKX(P)))*.
- However, triple DES with three keys also has the drawback of requiring 56 x 3 = 168 bits for the key, which can be slightly difficult to have in practical situations.
- A workaround suggested by Tuchman uses just two keys for triple DES. Here, the algorithm works as follows:
    1. Encrypt the plain text with key *K l* . Thus, we have *EK](P).*
    2. Decrypt the output of step 1 above with key *K l.* Thus, we have *DK1(EKX{P)).*
    3. Finally, encrypt the output of step 2 again with key *K l* . Thus, we have *EKX{DK2(EKX(P))).*
- To decrypt the cipher text C and obtain the original plain text P, we need to perform the operation *P = ((D ^ E A D ^ Q))*
- There is no special meaning attached to the second step of decryption.
- Its only significance is that it allows triple DES to work with two, rather than three keys. This is also called Encrypt-Decrypt-Encrypt (EDE) mode.
- Triple DES with two keys is not susceptible to the *meet-in-the-middle* attack, unlike double DES as *K l* and *K l* alternate here.

INTERNATIONAL DATA ENCRYPTION ALGORITHM: -

Operation: -

- IDEA operates on 64-bit blocks using a 128-bit key, and consists of a series of eight identical transformations (a *round*, see the illustration) and an output transformation (the *half-round*). The processes for encryption and decryption are similar.
- IDEA derives much of its security by interleaving operations from different groups — modular addition and multiplication, and bitwise eXclusive OR (XOR) — which are algebraically "incompatible" in some sense.
- In more detail, these operators, which all deal with 16-bit quantities, are:
    Bitwise eXclusive OR (denoted with a blue circled plus ⊕).
    Addition modulo 2^{16} (denoted with a green boxed plus ⊞).
- Multiplication modulo $2^{16}+1$, where the all-zero word (0x0000) in inputs is interpreted as 2^{16} and 2^{16} in output is interpreted as the all-zero word (0x0000) (denoted by a red circled dot ⊙).
- After the eight rounds comes a final "half round", the output transformation illustrated below (the swap of the middle two values cancels out the swap at the end of the last round, so that there is no net swap):



Structure: -

- The overall structure of IDEA follows the Lai-Massey scheme.
- XOR is used for both subtraction and addition. IDEA uses a key-dependent half-round function.
- To work with 16 bit words (meaning four inputs instead of two for the 64 bit block size), IDEA uses the Lai-Massey scheme twice in parallel, with the two parallel round functions being interwoven with each other.
- To ensure sufficient diffusion, two of the sub-blocks are swapped after each round.

Key schedule: -

- Each round uses six 16-bit sub-keys, while the half-round uses four, a total of 52 for 8.5 rounds.
- The first eight sub-keys are extracted directly from the key, with K1 from the first round being the lower sixteen bits; further groups of eight keys are created by rotating the main key left 25 bits between each group of eight.
- This means that it is rotated less than once per round, on average, for a total of six rotations.

Key generation:-

- First, the 128-bit key is partitioned into eight 16-bit sub-blocks which are then directly used as the first eight key sub-blocks
- The 128-bit key is then cyclically shifted to the left by 25 positions, after which the resulting 128-bit block is again partitioned into eight 16-bit sub-blocks to be directly used as the next eight key sub-blocks
- The cyclic shift procedure described above is repeated until all of the required 52 16-bit key sub-blocks have been generated

Key generation

- The 64-bit plaintext block is partitioned into four 16-bit sub-blocks
- six 16-bit key are generated from the 128-bit key. Since a further four 16-bit key-sub-blocks are required for the subsequent output transformation, a total of 52 (= 8 x 6 + 4) different 16-bit sub-blocks have to be generated from the 128-bit key.

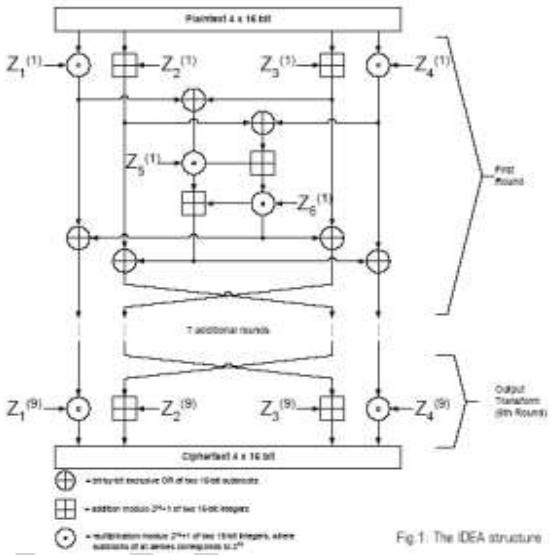


Fig.1: The IDEA structure

Encryption: -

Encryption

- the first four 16-bit key sub-blocks are combined with two of the 16-bit plaintext blocks using addition modulo 2^{16}, and with the other two plaintext blocks using multiplication modulo $2^{16} + 1$
- At the end of the first encryption round four 16-bit values are produced which are used as input to the second encryption round
- The process is repeated in each of the subsequent 7 encryption rounds
- The four 16-bit values produced at the end of the 8th encryption round are combined with the last four of the 52 key sub-blocks using addition modulo 2^{16} and multiplication modulo $2^{16} + 1$ to form the resulting four 16-bit ciphertext blocks

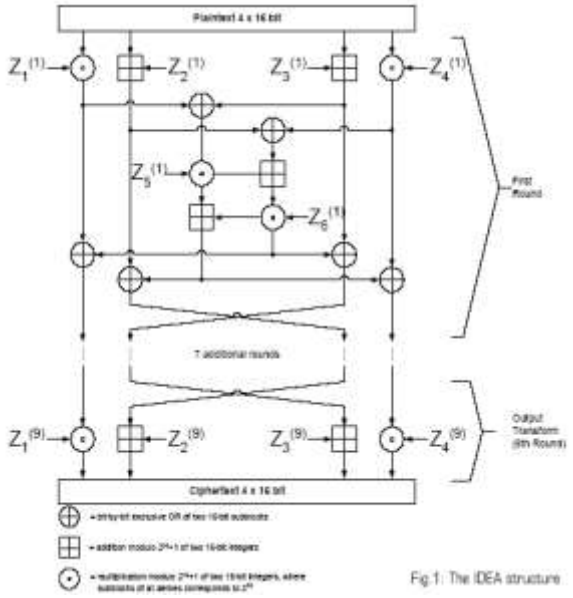


Fig.1: The IDEA structure

Decryption: -

- Decryption works like encryption, but the order of the round keys is inverted, and each value of subkeys K1 – K4 is replaced by its inverse for the respective group operation.
- The computational process used for decryption of the ciphertext is essentially the same as that used for encryption
- The only difference is that each of the 52 16-bit key sub-blocks used for decryption is the inverse of the key sub-block used during encryption
- In addition, the key sub-blocks must be used in the reverse order during decryption in order to reverse the encryption process

Security: -

- The designers analysed IDEA to measure its strength against differential cryptanalysis and concluded that it is immune under certain assumptions.
- No successful linear or algebraic weaknesses have been reported.
- As of 2007, the best attack which applied to all keys could break IDEA reduced to 6 rounds (the full IDEA cipher uses 8.5 rounds).
- Note that a "break" is any attack which requires less than 2^{128} operations; the 6-round attack requires 2^{64} known plaintexts and $2^{126.8}$ operations.

Weak keys: -

- The very simple key schedule makes IDEA subject to a class of weak keys; some keys containing a large number of 0 bits produce weak encryption.
- These are of little concern in practice, being sufficiently rare that they are unnecessary to avoid explicitly when generating keys randomly.
- A simple fix was proposed: exclusive-ORing each subkey with a 16-bit constant, such as 0x0DAE.

Applications: -

- Today, there are hundreds of IDEA-based security solutions available in many market areas, ranging from Financial Services, and Broadcasting to Government
- The IDEA algorithm can easily be embedded in any encryption software.
- Data encryption can be used to protect data transmission and storage.
- Typical fields are:
    1. Audio and video data for cable TV, pay TV, video conferencing, distance learning
    2. Sensitive financial and commercial data
    3. Email via public networks
    4. Smart cards

RC 4

- RC4 generates a pseudorandom stream of bits called keystream.
- This is combined with the plain text using XOR for encryption. Even decryption is performed in a similar manner.
- There is a variable length key consisting of 1 to 256 bytes (or 8 to 2048 bits).

- This key is used to initialize a 256-byte *state vector* with elements identified as *S* [0]. *S* [1]..... , *S* [255]. To perform an encryption or decryption operation, one of these 256 bytes of S is selected, and processed.
- We will call the resulting output as *k.* After this, the entries in S are permuted once again.
- Overall, there are two processes involved: (a) initialization of S, and (b) stream generation.

1. Initialization of S

- This process consists of the following steps.
    1. Choose a key *(K)* of length between 1 and 256 bytes.
    2. Set the values in the state vector *S* equal to the values from 0 to 255 in an ascending order. In other words, we should have *S* [0] = 0, *S* [1] = 1, ..., *S* [255] = 255.
    3. Create another temporary array *T.* If the length of the key *K* (termed as *keylen)* is 256 bytes, copy *K* into *T* as is. Otherwise, after copying *K* to 7, whatever are the remaining positions in *T* are filled with the values of *K* again. At the end, *T* should be completely filled.

- Thus, the following steps are executed:

```
for i = 0 to 255
// Copy the current value of i into the current position in the S array
s [i] = i;
// Now copy the contents of the current position of the K array into T. If K is exhausted, loop back
// to get the values of the K array from the unexhausted portion of K.
T [i] = K [i mod keylen];
```

- A small Java program shown implements this logic. Here, we have considered that the *K* array contains 10 elements, i.e. *keylen* is 10: -

```
public ciass InitRC4 {
public static void main (String [] args) {
int [ ] S, T, K;
S = new int [255] ;
T = new int [255] ;
K = new int [255] ;
int i;
int keylen = 10;
for (i=0; i<200; i++)
K [ i ] = i * 2;
for (i=0; i<255; i++) {
S [i] = i;
T [i] = K [i % k e y l e n ] ;
}
}
}
```

- After this, the *T* array is used to produce initial permutation of *S.*
- For this purpose, a loop executes, iterating i from 0 to 255. In each case, the byte at the position *S* [i] is swapped with another byte in the *S* array, as per an arrangement decided by *T* [i].
- For this purpose, the following logic is used:

```
y = 0;
. for i = 0 to 255
j = (j + S [i] + T [i]) mod 256;
swap (S [i], S [j]);
```

- Note that this is just a permutation. The values of *S* are simply being rearranged, not changed. The corresponding Java portion is shown: -

```
// Initial permutation of S
int j = 0, temp;
for (i=0; i<255; i++) {
j = (j + S [i] + T [i] ) % 256;
temp = S [i] ;
S [i] = T [i];
T [i] = temp;
}
for (i=0; i<255; i++) {
System.out.printIn("S [i] = " + S [i ] ) ;
}
```

2. <u>Stream Generation</u>

- Now that the *S* array is ready with the above initializations and permutations, the initial key array *K* is discarded.
- Now, we need to again loop for *i* = 0 to 255. In each step, we swap *S* [i] with another byte in S, as per the mechanism decided by the implementation of *S.*
- Once we exhaust the 255 positions, we need to restart at *S* [0].
- The logic is as follows: -

```
i = 0;
j = 0;
while (true)
i = (i + 1) mod 256;
j = (j + S [i]) mod 256;
swap (S [i], S [j]);
t = (S [i] + s[j]) mod 256;
k = S[t];
```

- After this, for encryption, *k* is XORed with the next byte of the plain text. For decryption, *k* is XORed with the next byte of the cipher text.

- Some vulnerability has been found in RC4, hence it is not recommended for new applications.

RC 5

RC5 block, round and key details: -

- In RC5, the word size (i.e. input plain-text block size), number of rounds and number of 8-bit bytes (octets) of the key, all can be of variable length.
- These values can consist of the sizes as shown in Table Of course, once decided, these values remain the same for a particular execution of the cryptographic algorithm.
- These are variable in the sense that before the execution of a particular instance of RC5, these values can be chosen from those allowed.
- This is unlike DES* for instance, where the block size must be of 64 bits and the key size must always be of 56 bits; or unlike IDEA, which uses 64-bit blocks and 128-bit keys.

Parameter	Allowed Values
Word size in bits (RC5 encrypts 2-word blocks at a time)	16, 32,64
Number o f rounds	0-255
Number o f 8-bit bytes (octets) in the key	0-255

- **The following conclusions emerge from the table:**
  - The plain-text block size can be of 32, 64 or 128 bits (since 2-word blocks are used).
  - The key length can be 0 to 2040 bits (since we have specified the allowed values for 8-bit keys).

- The output resulting from RC5 is the cipher text, which has the same size as the input plain text.
- Since RC5 allows for variable values in the three parameters, as specified, a particular instance of the RC5 algorithm is denoted as RC5-*w/r/b*, where w = word size in bits, *r* = number of rounds, *b* = number of 8-bit bytes in the key.
- Thus, if we have *RC5-32/16/16*, it means that we are using RC5 with a block size of *64* bits (remember that RC5 uses 2-word blocks), *16* rounds of encryption, and *16* bytes (i.e. 128 bits) in the key. Rivest has suggested RC5-32/12/16 as the *minimum safety version.*

Operation: -

- For simplicity, we shall assume that we are working on an input plain block with size 64 bits.
- The same principles operation will apply to other block sizes, in general.
- In the first two steps of the one-time initial operation, the input plain text is divided into two 32-bit blocks *A* and *B.*
- The first two subkeys *S[*0] and S[1] are added to *A* and *B*, respectively.
- This produces C and *D* respectively, and marks the end of the one-time operation.

- Then, the *rounds* begin. In each round, there are following operations:
  - Bitwise XOR
  - Left circular-shift
  - Addition with the next subkey, for both C and *D.*
- This is the addition operation first, and then the result of the addition mod *2W* (since *w* = 32 here, we have 232) is performed.

One-time Initial Operation

- This consists of two simple steps: first, the input plain text is divided into two equal-sized blocks, *A* and *B.*
- Then the first subkey, i.e. *S[*0] is added to *A,* and the second sub-key, i.e. *S[* 1] is added to *B.*
- These operations are mod 2^{32}, as we have noted, and produce C and Z), respectively.

Details of One Round

- Now, we observe the results for the first *round.* The process for the first *round* will apply for further *rounds.*

  Step 1: XOR C and D In the first step of each *round,* C and *D* are XORed together to form *E*.

  Step 2: Circular-left shift *E* Now, *E* is circular-left shifted by *D* positions.

  Step 3: Add *E* and Next Subkey In this step, *E* is added to the next subkey (which is S[2] for the first round, and *S[2i]* in general, for any round, where *i* starts with 1. The output of this process is *F.*

  Step 4: XOR D and *F* This step is similar to step 1. Here, *D* and *F* are XORed to produce *G.*

  Step 5: Circular-left Shift G This step is similar to step 2. Here, *G* is circular-left shifted by *F* Positions

  Step 6: Add G and Next Subkey In this step (which is identical to step 3), *G* is added to the next subkey (which is *S[*3] for the first round, and *S[2i* + 1] in general, for any round, where *i* starts with 1. The output of this process is *H.*

  Step 7: Miscellaneous Tasks In this step, we check to see if all the rounds are over or not. For this, perform the following steps:
  - Increment *i* by 1 # Check to see if *i* < *r* Assuming that *i* is still less than *r,* we rename *F* as *C* and *H* as D, and return back to step 1

Mathematical representation:-

- Interestingly, all these operations (one-time initial operation and all the *rounds*) in RC5 can be signified mathematically in a very cryptic fashion, as shown in Fig that <<< means circular-left shift.

```
A = A + S [ 0 ]
B = B + S [ 1 ]
For i = 1 to r
A = ( (A XOR B) <<<B) + S [ 2 i ]
B = ( (B XOR A) <<<A) + S[2i + 1]
Next i
```

Note that » > means circular right-shift.

```
For i = r to 1 step-1 (i.e. decrement i each time by 1)
A = ((B - S[2i + 1] ) >>> A) XOR A
B = ( ( A - S [ 2 i ] ) >>> B ) XOR B
Next i
B = B - S [1]
A = A - S[0]
```

Sub key creation: -

- This is a two-step process.
    1. In the first step, the subkeys (denoted by *S[*0], S [l] ,...) are generated.
    2. The original key is called *L.* In the second step, the subkeys (S[0], *S[* 1], ...) are mixed with the corresponding subportions of the original key (i.e. *L[* 0], Z[l], ...).

Sub key Generation: -

- In this step, two constants *P* and *Q* are used.
- The array of subkeys to be generated is called as *S.*
- The first subkey *S[*0] is initialized with the value of *P.*
- Each next subkey (i.e. *S[* 1], S [2],...) is calculated on the basis of the previous sub-key and the constant value *Q*, using the addition mod 232 operations.
- The process is done 2 (r + 1) - 1 times, where *r* is the number of *rounds*, as before.
- Thus, if we have 12 rounds, this process will be done 2 (12 + 1) - 1 times, i.e. 2 (13) - 1 times, i.e. 25 times.
- Thus, we will generate subkeys S[0], *S[* 1], ... £[25].
- The mathematical form of this subkey generation is: -

```
s [0] = P
For i = 1 to 2 (r + 1) - 1
S [i ] = (S[i - 1] + Q) mod 2^32
Next i
```

Sub key Mixing: -

- In the subkey mixing stage, the subkeys *S[*0], S[l], ... are mixed with the subportions of the original key, i.e. L[0], L[1], •• *L[*c].
- Note that *c* is the last subkey position in the original key.
- This process is shown mathematically as: -

```
i = j = 0
A = B = 0
Do 3n times (where n is the :maximum of 2 (r + 1) and c)
A = S [i] = (S [i] + A + B) « < 3
```

```
B = L [i] = (L[i ] + A + B) « < (A + B)
i = (i + 1) mod 2 (r + 1)
j = (j + 1) mod c
End-do
```

RC5 modes

- RC5 block cipher: - This is also called Electronic Codebook (ECB) mode. It involves encryption of fixed size input blocks consisting of 2*w* bits into cipher-text block of the same length.
- RC5- CBC: - This is the cipher chaining Block for RC5. In this, plain-text messages whose length is equal to a multiple of the RC5 block size (i.e. multiple of *2w* bits) are encrypted. This mode offers better security, as the same plain-text blocks in the input produce different cipher-text blocks in the output.
- RC5-CBC-Pad: - This is a modified version of CBC. Here, the input can be of any length. The cipher text is longer than the plain text by at the most the size of a single RC5 block. To handle length mismatches, padding is used. This makes the length of a message equal to a multiple of 2*w* bits. The length of the original message is considered to be a specific number of integer number of bytes. Padding of length 1 to *bb* bytes is added to the end of the message. How many such padding bytes are added is decided by a simple formula $bb = 2w / 8$. Thus, *b* equals the block size for RC5 in bytes. The value of all padding bytes is the same, and is set to a byte that represents the number of bytes used in padding. For instance, if there are 6 padding bytes, each padding byte contains a value of 6 in binary, i.e. 00000110.
- RC5-CTS: - This is called cipher-text stealing mode. This mode is similar to RC5-CBC-Pad. Here, the plain-text input can be of any length. The output cipher text is also of equal length.

BLOWFISH

- Blowfish was developed by Bruce Schneier, and has the reputation of being a very strong symmetric key cryptographic algorithm.

- According to Schneier, Blowfish was designed with the following objectives in mind.
  (a) Fast Blowfish encryption rate on 32-bit microprocessors is 26 clock cycles per byte.
  (b) Compact Blowfish can execute in less than 5 KB memory.
  (c) Simple Blowfish uses only primitive operations, such as addition, XOR and table look-up, making its design and implementation simple.
  (d) Secure Blowfish has a variable key length up to a maximum of 448 bits long, making it both flexible and secure.

- Blowfish suits applications where the key remains constant for a long time (e.g. communications link encryption), but not where the key changes frequently (e.g. packet switching).

Operation

- Blowfish encrypts 64-bit blocks with a variable-length key.
- It contains two parts, as follows.
  (a) Subkey Generation: - This process converts the key up to 448 bits long to subkeys totaling 4168 bits.

(b) Data Encryption: - This process involves the iteration of a simple function 16 times. Each round contains a key-dependent permutation and key- and data-dependent substitution.

Subkey Generation

- Let us understand the important aspects of the subkey generation process step by step.

(a) Blowfish makes use of a very large number of subkeys. These keys have to be ready before encryption and decryption happen. The key size ranges from 32 bits to 448 bits. In other words, the key size ranges from 1 to 14 words, each comprising a word of 32 bits. These keys are stored in an array, as follows: K1,K2, ...,*K n* where $1 < n < 14$

(b) We then have the concept of a *P*-array, consisting of 18 32-bit sub-keys: P1, *P2*... P18 . Creation of the P-array is described subsequently.

(c) Four S-boxes, each containing 256 32-bit entries:

s1,0,s1,1 ...,s1,255
s2,0,s2.1 ....s2,255
s3,0,s3,1 ...,s3,255
s4,0,s3,1 ...,s4,255

Creation of the P-array is described subsequently.

- Now let us examine how all this information is used to generate subkeys.

(a) Initialize the P-array first, followed by the four 5-boxes, with a fixed string. Schneier recommends the usage of the bits of the fractional part (in hexadecimal form) of the constant pi (π) for this purpose. Thus, we will have:

PI = 243F6A88
P2 = 85A308D3
s4.254 = 578FDFE3
s4.255 = 3AC372E6

(b) Do a bitwise XOR of P1 with *K1*,*P 2* with *K 2*, etc., until P18. This works fine till P14 and *K14.* At this stage, the key array is exhausted. Hence, for P15 to P18, reuse *K1* to *K4.* In other words, do the following:

PI = P 1 XOR K1
PI = P2 XOR *K2*
P14 = P14 XOR K14
P15 = P l5 XOR K1
P16 = P16 XOR K2
P 17 = PI7 XOR K3
P18 = P18 XOR K4

(c) Now take a 64-bit block, with all the 64 bits initialized to value of 0. Use the above P-arrays and 5-boxes above (the P-arrays and 5-boxes are called subkeys) to run the Blowfish encryption process (described in the next section) on the 64-bit all-zero block. In other words, to generate the sub keys themselves, the Blowfish algorithm is used. It is needless to say that once the final sub keys are ready, the Blowfish algorithm would be used to encrypt the actual plain text. This step would produce a 64-bit cipher text. Divide this into two 32-bit blocks and replace the original values of PI and P2 with these 32-bit block values, respectively.

(d) Encrypt the output of step (c) above using Blowfish with the modified sub keys. The resulting

output would again consist of 64 bits. As before, divide this into two blocks of 32 bits each. Now, replace P3 and P4 with the contents of these two cipher text blocks.

(e) In the same manner, replace all the remaining P-arrays (i.e. P5 through P18) and then all the elements of the four 5-boxes, in order. In each step, the output of the previous step is fed to the Blowfish algorithm to generate the next two 32-bit blocks of the subkey (i.e. P5 and P6, followed by P7 and *PS*, etc). In all, 521 iterations of the Blowfish algorithm are required to generate all P-arrays and 5-boxes.

Data Encryption and Decryption

- We use the P-arrays and S-boxes during the encryption and decryption processes.

1. Divide X into two blocks: XL and XR, of equal sizes. Thus, both XL and XR will consist of 32 bits each.
2. For i = 1 to 16
   XL = XL XOR Pi
   XR = F (XL) XOR XR
   Swap XL, XR
   Next i
3. Swap XL, XR (i.e. undo last swap) .
4 . XL = XL XOR P18 .
5. Combine XL and XR back into X.

ADVANCED ENCRYPTION STANDARDS (AES): -

Main features of AES are as follows: -

(a) Symmetric and Parallel Structure This gives the implementers of the algorithm a lot of flexibility. It also stands up well against cryptanalysis attacks.
(b) Adapted to Modern Processors The algorithm works well with modem processors (Pentium, RISC, parallel).
(c) Suited to Smart Cards The algorithm can work well with smart cards.

Operation:-

- The basics of Rijndael are in a mathematical concept called Galois field theory.
- Similar to the way DES functions, Rijndael also uses the basic techniques of substitution and transposition (i.e. permutation). The key size and the plain-text block size decide how many rounds need to be executed.
- The minimum number of rounds is 10 (when key size and the plain-text block size are each 128 bits) and the maximum number of rounds is 14.
- One key differentiator between DES and Rijndael is that all the Rijndael operations involve an entire byte, and not individual bits of a byte.
- This provides for more optimized hardware and software implementation of the algorithm

Description of Rijindale: -

(i) Do the following one-time initialization processes:
- (a) Expand the 16-byte key to get the actual *keyblock to* be used.
- (b) Do one time initialization of the 16-byte plain-text block (called *State).*
- (c) XOR the *state* with the *key block.*

(ii) For each round, do the following:
- (a) Apply S-box to each of the plain-text bytes.
- (b) Rotate row *k* of the plain-text block (i.e. *state)* by *kbytes.*
- (c) Perform a *mix columns* operation.
- (d) XOR the *state* with the *key block*

One-time Initialization Process

1. *Expand the 16-byte key to get the actual key block to be used*

- The inputs to the algorithm are the key and the plain text, as usual.
- The key size is 16 bytes in this case.
- This step expands this 16-byte key into 11 arrays, and each array contains 4 rows and 4 columns.
- In other words, the original 16-byte key array is expanded into a key containing 11 x 4 x 4 = 176 bytes.
- One of these, 11 arrays are used in the initialization process and the other 10 arrays are used in the 10 rounds, one array per round.
- The key-expansion process is quite complex, and can be safely ignored.
- Now, let us start using the terminology of *word,* in the context of AES.
- A word means 4 bytes. Therefore, in the current context, our 16-byte initial key (i.e. 16/4 = 4-word key) will be expanded into 176-byte key (i.e. 176/4 words, i.e. 44 words).
  (a) Firstly, the original 16-byte key is copied into the first 4 words of the expanded key (i.e. the first 4 x 4 array of our diagram).
  (b) After filling the first array (for words numbered *Wl* to *W3)* of the expanded key block the remaining 10 arrays (for words numbered *W4* to *W43)* are filled one by one.
- Every time, one such 4 x 4 array (i.e. four words) gets filled.
- Every added key array block depends on the immediately preceding block and the block 4 positions earlier to it.
- That is, every added word *w* [i] depends on *w* [i - 1] and *w [i* - 4]. We have said that this fills four words at a time.
- For filling these four words at a time, the following logic is used:
  (i) If the word in the *w* array is a multiple of four, some complex logic is used, explained later below. That is, for words w [4], w [8], *w* [12]... *w* [40], this complex logic would come into picture.
  (ii) For others, a simple XOR is used.

Its logic is: -

```
Expand Key (byte K [16], word W [44]) {
word tmp;

// First copy all the 16 input key blocks into first four words of output key
```

```
for (i = 0; i < 4; i++)
{
W [i] = K [4*i] , K [4*i + 1] , K [4*i + 2] , K [4*i + 3] ;
}

// Now populate the remaining output key words (i.e. W5 to W43)

for (i = 4 ; i < 44; i++)
{
tmp = W [i — 1] ;
if (i mod 4 == 0)
tmp = Substitute (Rotate (temp)) XOR Constant [i/4];
w [i] = w [i — 4] X OR tmp;
}
}
```

- In the second *for* loop, we check if the current word being populated in the output key block is a multiple of 4. If it is, we perform three functions, titled *substitute*, *rotate*, and *constant.*
- Let us now understand the three functions, titled *Substitute*, *Rotate*, and *Constant.*
  (i) Function *Rotate* performs a circular left shift on the contents of the word by one byte. Thus, if an input word contains four bytes numbered [*B*1 , B2, B3, B4] then the output word would contain [B2 ,B3 ,B4 ,**B**1].
  (ii) Function *Substitute* performs a byte substitution on each byte of the input word. For this purpose, it uses an S-box.
  (iii) In the function *Constant*, the output of the above steps is XORed with a constant. This constant is a word, consisting of 4 bytes. The value of the constant depends on the round number. The last three bytes of a constant word always contain 0. Thus, XORing any input word with such a constant is as good as XORing only with the first byte of the input word.

2) Do one-time initialization of the 16-byte plain-text block (called State)

- This step is relatively simple. Here, the 16-byte plain-text block is copied into a two-dimensional 4 x 4 array called *state.*
- The order of copying is in the column order.
- That is, the first four bytes of the plaintext block get copied into the first column of the *state* array, the next four bytes of the plain-text block get copied into the second column of the *state* array, and so on.

3) XOR the state with the key block: -

- Now, the first 16 bytes (i.e. four words *W* [0], *W* [1], *W* [2], and *W* [3]) of the expanded key are XORed into the 16-byte *state* array (51 to 516 shown above).
- Thus, every byte in the *state* array is replaced by the XOR of itself and the corresponding byte in the expanded key.
- At this stage, the initialization is complete, and we are ready for rounds. Processes in Each Round
- The following steps are executed 10 times, one per round.

1. Apply S-box *to each of the plain-text bytes*

This step is very straightforward. The contents of the *state* array are looked up into the S-box. Byteby-byte substitution is done to replace the contents of the *state* array with the respective entries in the S-box. Note that only one S-box is used, unlike DES, which has multiple S-boxes.

2. Rotate row k of the plain-text block (i.e. state) by k bytes

Here, each of the four rows of the *state* array are rotated to the left. Row 0 is rotated 0 bytes (i.e. not rotated at all), row 1 is rotated by 1 byte, row 2 is rotated 2 bytes, and row 2 is rotated 3 bytes. This helps in diffusion of data. Thus, if the original 16 bytes of the *state* array contain values 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 then the rotate operation would change the values as shown below.

Original array	**Modified array**
1 5 9 13	1 5 9 13
2 6 10 14	6 10 14 2
3 711 15	11 15 3 7
4 812 16	16 4 812

3. Perform a mix-columns operation

Now, each column is mixed independent of the other. Matrix multiplication is used. The output of this step is the matrix multiplication of the old values and a constant matrix.

UNIT 3

Symmetric-key cryptography is fast and efficient. However, it suffers from a big disadvantage of the problem of key exchange. The sender and the receiver of an encrypted message use the same key in symmetric-key cryptography, and it is very tough to agree upon a common key without letting anyone else know about it. Asymmetric-key cryptography solves this problem. Here, each communicating party uses two keys to form a key pair—one key (the private key) remains with the party, and the other key (the public key) is shared with everybody.

BRIEF HISTORY OF ASYMMETRIC-KEY CRYPTOGRAPHY

We have discussed the problem of key exchange, also called key distribution or key agreement, at great length. To revise it quickly, in any symmetric-key cryptographic scheme, the main issue is: How can the sender and the receiver of a message decide upon the key to be used for encryption and decryption? In computer-based cryptographic algorithms, this problem is even more serious, because the sender and the receiver may be in different countries. For example, suppose that the seller of some goods has set up an online shopping Web site. A customer residing in a different country wants to place an order (in an encrypted form, so as to maintain confidentiality, for whatever reasons) over the Internet.

How would the customer (i.e. the customer's computer: we shall use these terms interchangeably) encrypt the order details before sending them to the seller? Which key would he/she use? How would the customer then inform the seller about the key, so that the seller can decrypt the message? Remember that the encryption and decryption must be done using the same key. As we have seen, in symmetrickey cryptography, this problem of key exchange cannot be solved. Moreover, we need a unique key per communicating party. That is also quite cumbersome and undesired. In the mid-1970s, Whitfield Diffie, a student at the Stanford University met with Martin Heilman, his professor, and the two began to think about the problem of key exchange. After some research and complicated mathematical analysis, they came up with the idea of asymmetric-key cryptography. Many experts believe that this development is the first—and perhaps the only—truly revolutionary concept in the history of cryptography. Diffie and Heilman can, therefore, be regarded as the fathers of asymmetric-key cryptography.

However, there is a lot of debate regarding who should get the credit for developing asymmetric-key cryptography. It is believed that James Ellis of the British *Communications Electronic Security Group (CSEG)* proposed the idea of asymmetric-key cryptography in the 1960s. His ideas were based on an anonymous paper written at the Bell Labs during the Second World War. However, Ellis could not devise a practical algorithm based on his ideas. He then met with Clifford Cocks, who joined the CSEG in 1973. After a short discussion between Ellis and Cocks, Cocks came up with a practical algorithm that could work! Next year, Malcolm Williamson, another employee at the CSEG, developed an asymmetric-key cryptographic algorithm. However, since the CSEG was a secret agency, these findings were never published, and therefore, it is believed that these people never got the credit that they deserved.

Simultaneously, the US *National Security Agency (NSA)* was also working on asymmetric-key cryptography.
It is believed that the NSA system based on the asymmetric-key cryptography was operational in the mid-1970s.

Based on the theoretical framework of Diffie and Heilman, in 1977, Ron Rivest, Adi Shamir and Len Adleman at MIT developed the first major asymmetric-key cryptography system, and published their results in 1978. This method is called RSA algorithm. The name RSA comes from the first letters of the surnames of the three researchers. Actually, Rivest was working as a professor at the MIT. He had recruited Shamir and Adleman to work on the concept of asymmetric-key cryptography.

Even today, RSA is the most widely accepted public-key solution. It solves the problem of key agreements and distribution. As we mentioned, the approach used here is that each communicating party possesses a key pair, made up of one public key and one private key.

To communicate securely over any network, all one needs to do is to publish one's public key. All these public keys can then be stored in a database that anyone can consult. However, the private key only remains with the respective individuals.

AN OVERVIEW OF ASYMMETRIC-KEY CRYPTOGRAPHY

In *asymmetric-key cryptography*, also called public key cryptography, two *different* keys (which form a *key pair)* are used. One key is used for encryption and only the other corresponding key must be used for decryption. No other key can decrypt the message—not even the original (i.e. the first) key used for encryption! The beauty of this scheme is that every communicating party needs just a key pair for communicating with any number of other communicating parties. Once someone obtains a key-pair, he/she can communicate with anyone else.

There is a simple mathematical basis for this scheme. If you have an extremely large number that has only two factors, which are prime numbers, you can generate a pair of keys. For example, consider a number 10. The number 10 has only two factors, 5 and 2. If you apply 5 as an encryption factor, only 2 can be used as the decryption factor. Nothing else—not even 5 itself—can do the decryption. Of course, 10 is a very small number. Therefore, with minimal effort, this scheme can be broken into. However, if the number is very large, even years of computation cannot break the scheme.

One of the two keys is called the *public key* and the other is the *private key.* Let us assume that you want to communicate over a computer network such as the Internet in a secure manner. You would need to obtain a public key and a private key. We shall study later how these keys can be obtained. The private key remains with you as a secret. You must not disclose your private key to anybody. However, the public key is for the general public. It is disclosed to all parties that you want to communicate with.

In this scheme, in fact, each party or node publishes its public key. Using this, a directory can be constructed where the various parties or nodes (i.e. their ids) and the corresponding public keys are maintained. One can consult this and get the public key for any party that one wishes to communicate with by a simple table search.

Suppose *A* wants to send a message to *B* without having to worry about its security. Then, *A* and *B* should each have a private key and a public key.

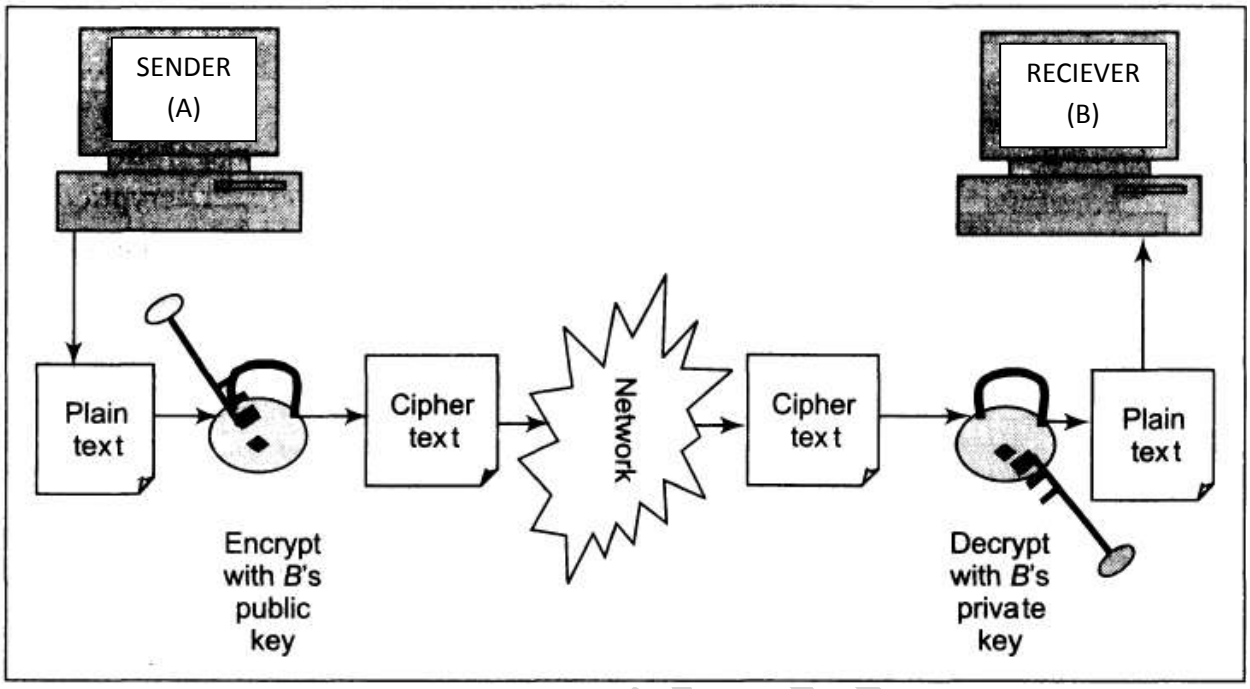
- *A* should keep her private key secret.
- *B* should keep her private key secret.
- *A* should inform *B* about her public key.
- *B* should inform *A* about her public key.

Armed with this knowledge, asymmetric-key cryptography works as follows:

1. When *A* wants to send a message to *B* , *A* encrypts the message using B's public key. This is possible because *A* knows B's public key.

2. *A* sends this message (which was encrypted with *B 's* public key) to *B.*

3. *B* decrypts *A* 's message using *B 's* private key. Note that only *B* knows about her private key. Also note that the message can be decrypted only by *B's* private key and nothing else! Thus, no one else can make any sense out of the message even if one can manage to intercept the message. This is because the intruder (ideally) does not know about B's private key. It is only *B* 's private key that can decrypt the message.



THE RSA ALGORITHM

The RSA algorithm is the most popular and proven asymmetric-key cryptographic algorithm.

1. Choose two large prime numbers *P* and Q.
2. Calculate $N = P \times Q$.
3. Select the public key (i.e. the encryption key) *E* such that it is not a factor of (P - 1) and (Q - 1) .
4. Select the private key (i.e. the decryption key) *D* such that the following equation is true: (D x E) mod (P - 1) x (Q - 1) = 1
5. For encryption, calculate the cipher text *CT* from the plain text *PT* as follows:

$CT = PT^E \bmod N$

6. Send CT as the cipher text to the receiver.
7. For decryption, calculate the plain text *PT* from the cipher text *CT* as follows:

$PT = CT^D \bmod N$

Security of RSA

Although no successful attacks on RSA have been reported so far, chances of such attacks happening in the future have not been ruled out. We discuss some of the main possible attacks on RSA below.

1. Plain text Attacks

The plain-text attacks are classified further into three sub-categories, as discussed next.

(a) Short-message Attack Here, the assumption is that the attacker knows some possible blocks of plain text. If this assumption is true, the attacker can try encrypting each plain-text block to see if it results into the known cipher text. To prevent this short message attack, it is recommended that we pad the plain text before encrypting it.

(b) Cycling Attack Here, the attacker presumes that the cipher text was obtained by doing permutation on the plain text in some manner. If this assumption is true then the attacker can do the reverse process, which is to continuously do permutations on the known cipher text to try and obtain the original plain text. However, the trouble for the attacker could be that the attacker does not know what can be considered as the right plain text while using this method. Hence, the attacker can keep on doing permutation of the cipher text until he/she obtains the cipher text itself, in other words, completing one full cycle of permutations. If the attacker does obtain the original cipher text again by this method, the attacker knows that the text that was obtained in the step just prior to obtaining the original cipher text must be the original plain text. Hence, this attack is called cycling attack. However, this has not been found to be successful so far.

(c) Unconcealed Message Attack It is found in theory that in the case of some very rare plaintext messages, encryption gives cipher text which is the same as the original plain text! If this happens, the original plain-text message cannot be hidden. Hence, this attack is called unconcealed message attack. Hence, it may be prudent to ensure that after RSA encryption, the resulting cipher text is not the same as the original plain text before the cipher text is sent to the receiver, to prevent this attack.

2. Chosen-cipher text attack

In this complicated scheme called chosen-cipher text attack, the attacker is able to find out the plain text based on the original cipher text using what is called *extended Euclidean algorithm.*

3. Factorization Attack

The whole security of RSA is based on the assumption that it is infeasible for the attacker to factor the number N into its two factors P and Q. However, if the attacker is able to find out P or Q from the equation $N = P \times Q$ then the attacker can find out the private key, as we have discussed earlier. Assuming that N is at least 300 digits long in decimal terms, the attacker cannot find P and Q easily. Hence, the factorization attack fails.

4. Attacks on the Encryption Key

People well versed with the mathematics of RSA sometimes feel that it is quite slow because we use a large number for the public key or encryption key E. While this is true, it also makes RSA more secure. Hence, if we decide to try and make the working of RSA faster by using a small value for E, it can lead to potential attacks called attacks on the encryption key and hence it is recommended that we use E as 216 + 1 = 65537 or a value closer to this number.

5. Attacks on the Decryption Key

The attacks on the decryption key can be classified further into two categories, as discussed below.

(a) Revealed Decryption Exponent Attack If the attacker can somehow guess the decryption key *D*, not only are the ciphertexts generated by encrypting the plain texts with the corresponding encryption key *E* are in danger; but even the future messages are also in danger. To prevent this revealed decryption exponent attack, it is recommended that the sender uses fresh values for *P*, *Q*, *N*, and *E* as well.

(b) Low Decryption Exponent Attack Similar to the case explained in the context of the encryption key, it is tempting to use a small value for decryption key *D* to make RSA work faster. This can help the attacker in guessing the decryption key *D* by launching the low decryption exponent attack.

EIGamal CRYPTOGRAPHY

Taher EIGamal created EIGamal cryptography, more popularly known as EIGamal cryptosystem. We leave the complicated mathematics behind and explain the algorithm in a simpler form below. There are three aspects that need to be discussed: EIGamal key generation, EIGamal encryption, and EIGamal decryption.

EIGamal Key Generation

This involves the following steps:

1. Select a large prime number called *P.* This is the first part of the encryption key or public key.

2. Select the decryption key or private key *D.* There are some mathematical rules that need to be

followed here, which we are omitting for keeping things simple.

3. Select the second part of the encryption key or public key *E1*

4. The third part of the encryption key or public key *E2* is computed as $E2 = E1^D \bmod P$.

5. The public key is *(E1*, *E2*, *P)* and the private key is *D.*

For example, $P = 11$, $E1 = 2$, $D = 3$. Then $E2 = E1^D \bmod P = 23 \bmod 11 = 8$.

Hence, the public key is (2, 8, 11) and the private key is 3.

EIGamal Key Encryption

This involves the following steps:

1. Select a random integer *R* that fulfills some mathematical properties, which are ignored here.

2. Compute the first part of the cipher text C1 = $E1^R$ mod P.

3. Compute the second part of the cipher text C2 = $(PT \times E2^R)$ mod P, where PT is the plain text.

4. The final cipher text is (C1, C2).

In our example, let $R = 4$ and plain text $PT = 7$. Then we have:

C1 = $E1^R$ mod P = 24 mod 11 = 16 mod 11= 5

C2 = $(PT \times E2^R)$ mod P = (7 x 28) mod 11 = (7 x 4096) mod 11 = 6

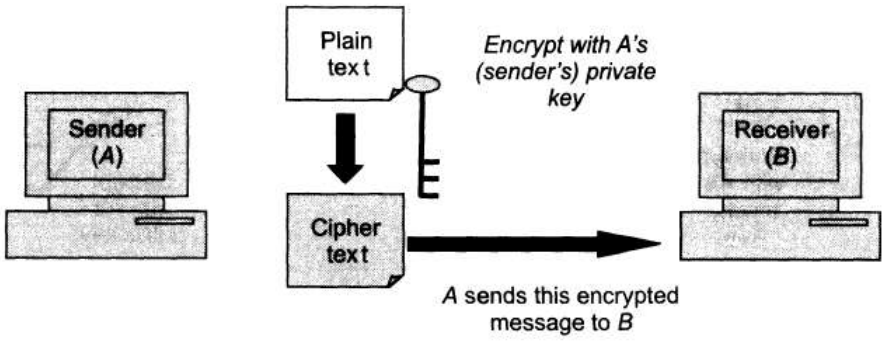
Hence, our cipher text is (5, 6).

Comparison Between Symmetric- and Asymmetric-Key Cryptography

Characteristic	**Symmetric-key Cryptography**	**Asymmetric-key Cryptography**
Key used for encryption/decryption	Same key is used for encryption and decryption	One key used for encryption and another, different key is used for decryption
Speed of encryption/decryption	Very fast	Slower
Size of resulting encrypted text	Usually same as or less than the original clear text size	M ore than the original clear text size
Key agreement/exchange	A big problem	No problem at all
Number of keys required as compared to the number of participants in the message exchange	Equals about the square of the number of participants, so scalability is an issue	Same as the number of participants, so scales up quite well
Usage	Mainly for encryption and decryption(confidentiality), cannot be used for (digital signatures be used for digital signatures checks)	Can be used for encryption and decryption (confidentiality) as well as for digital signatures (integrity and non-repudiation checks)

DIGITAL SIGNATURES

If A is the sender o f a message and B is the receiver, A encrypts the message with A s private key and sends the encrypted message to B.



A encrypts the message with her private key, her intention is not to hide the contents of the message (i.e. not to achieve *confidentiality)*, but it is something else. What can that intention be? If the receiver *(B)* receives such a message encrypted with *A 's* private key, *B* can use *A* 's public key to decrypt it, and therefore, access the plain text. Does this ring a bell? If the decryption is successful, it assures *B* that this message was indeed sent by *A.* This is because if *B* can decrypt a message with *A 's* public key, it means that the message must have been initially encrypted with *A 's* private key (remember that a message encrypted with a public key can be decrypted only with the corresponding private key, and vice versa). This is also because only *A* knows her private key. Therefore, someone posing as *A* (say Q could not have sent a message encrypted with *A* 's private key to *B. A* must have sent it. Therefore, although this scheme does not achieve confidentiality, it achieves *authentication* (identifying and proving *A* as the sender). Moreover, in the case of a dispute tomorrow, *B* can take the encrypted message, and decrypt it with *A 's* public key to prove that the message indeed came from *A.* This achieves the purpose of *non-repudiation* (i.e. *A* cannot refuse that she had sent this message, as the message was encrypted with her private key, which is supposed to be known only to her).

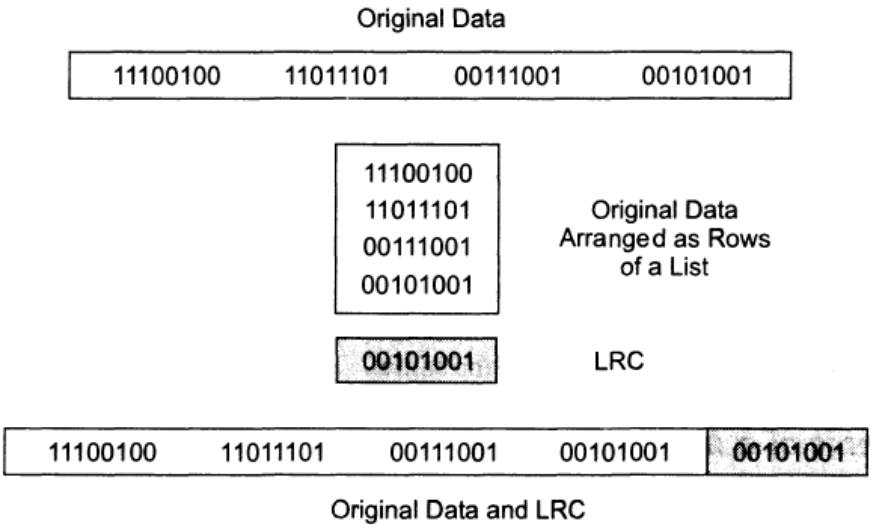
Message Digests

A message digest is a *fingerprint* or the summary of a message. It is similar to the concepts of *Longitudinal Redundancy Check (LRC)* or *Cyclic Redundancy Check (CRC).* That is, it is used to verify the *integrity* of the data (i.e. to ensure that a message has not been tampered with after it leaves the sender but before it reaches the receiver). Let us understand this with the help of an LRC example (CRC would work similarly, but will have a different mathematical base).

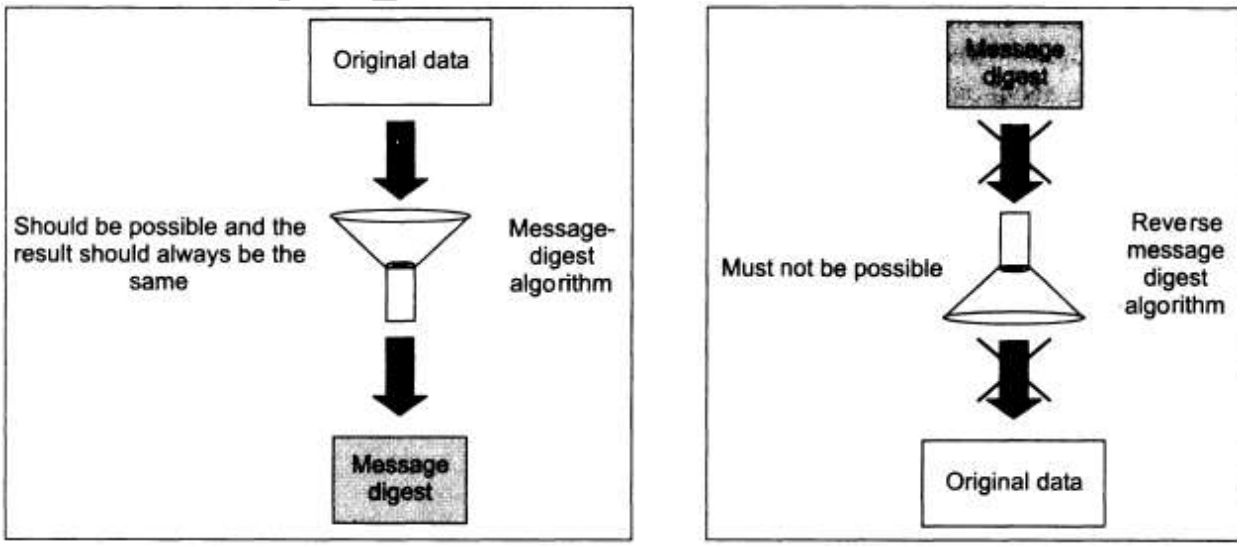
LRC calculation

A block of bits is organized in the form of a list (as rows) in the *Longitudinal Redundancy Check (LRC).* Here, for instance, if we want to send 32 bits, we arrange them into a list of four (horizontal) rows. Then we count how many 1 bits occur in each of the 8 (vertical) columns. [If the number of Is in the column is odd then we say that the column has *odd parity* (indicated by a 1 bit in the shaded LRC row); otherwise if the number of Is in the column is even, we call it *even parity* (indicated by a 0 bit in the shaded LRC row).] For instance, in the first column, we have two Is, indicating an even parity, and

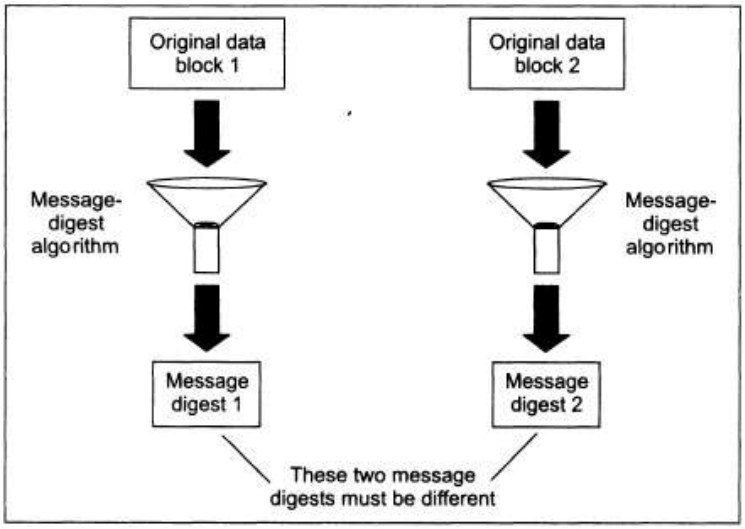
therefore, we have a 0 in the shaded LRC row for the first column. Similarly, for the last column, we have three Is, indicating an odd parity, and therefore, we have a 1 in the shaded LRC row for the last column. Thus, the parity bit for each column is calculated and a new row of eight parity bits is created. These become the parity bits for the whole block. Thus, the LRC is actually a *fingerprint* of the original message.



The data along with the LRC is then sent to the receiver. The receiver separates the data block from the LRC block (shown shaded). It performs its own LRC on the data block alone. It then compares its LRC values with the ones received from the sender. If the two LRC values match then the receiver has a reasonable confidence that the message sent by the sender has not been changed, while in transit.



Given any two messages, if we calculate their message digests, the two message digests must be different.



If any two messages produce the same message digest, thus violating our principle, it is called a collision. That is, if two message digests *collide*, they meet at the digest! As we shall study soon, the message-digest algorithms usually produce a message digest having a length of 128 bits or 160 bits. This means that the chances of any two message digests being the same are one in 2128 or 2160, respectively. Clearly, this seems possible only in theory, but extremely rare in practice.

A specific type of security attack called birthday attack is used to detect collisions in message-digest algorithms. It is based on the principle of the *Birthday Paradox*, which states that if there are 23 people in a room, chances are that more than 50% of the people will share the same birthday. At first, this may seem to be illogical. However, we can understand this in another manner. We need to keep in mind we are just talking about *any two* people (out of the 23) sharing the same birthday. Moreover, we are not talking about this sharing with a specific person. For instance, suppose that we have Alice, Bob, and Carol as three of the 23 people in the room. Therefore, Alice has 22 possibilities to share a birthday with anyone else (since there are 22 pairs of people). If there is no matching birthday for Alice, she leaves. Bob now has 21 chances to share a birthday with anyone else in the room. If he fails to have a match too, the next person is Carol. She has 20 chances, and so on. 22 pairs + 21 pairs + 20 pairs ... + 1 pair means that there is a total of 253 pairs. Every pair has a l/365th chance of finding a matching birthday. Clearly, the chances of a match cross 50% at 253 pairs.

The birthday attack is most often used to attempt to discover collisions in hash functions, such as MD5 or SHA1.

If a message digest uses 64-bit keys then after trying 2A32 transactions, an attacker can expect that for two different messages, he/she may get the same message digests. In general, for a given message, if we can compute up to *N* different message digests then we can expect the first collision after the

number of message digests computed exceeds the square root of *N.* In other words, a collision is expected when the probability of collision exceeds 50%. This can lead to birthday attacks.

MD5

MD5 is a message-digest algorithm developed by Ron Rivest. MD5 actually has its roots in a series of message-digest algorithms, which were the predecessors of MD5, all developed by Rivest. The original message-digest algorithm was called **MD.** Rivest soon came up with its next version, **MD2.** He first developed it, but it was found to be quite weak. Therefore, Rivest began working on **MD3,** which was a failure (and therefore, was never released). Then, Rivest developed **MD4.** However, soon, MD4 was also found to be wanting. Consequently, Rivest released MD5.

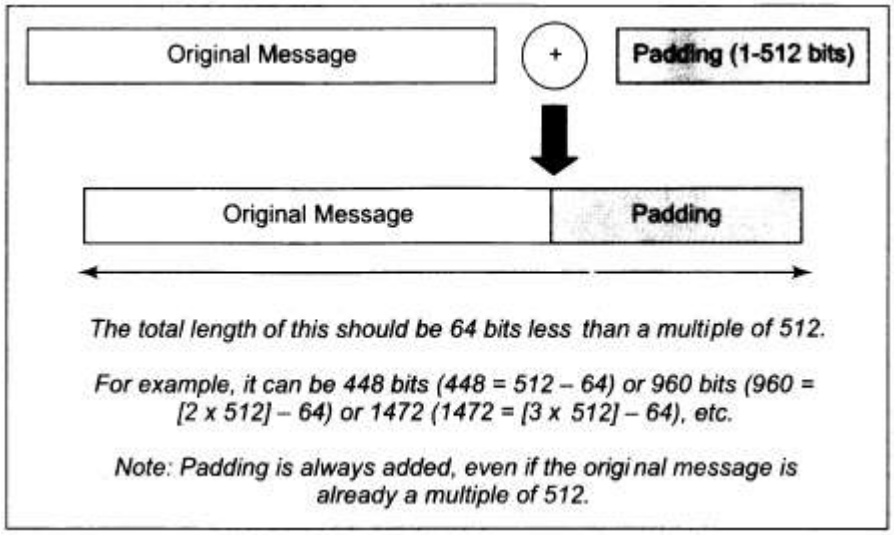
MD5 is quite fast, and produces 128-bit message digests. Over the years, researchers have developed potential weaknesses in MD5. However, so far, MD5 has been able to successfully defend itself against collisions. This may not be guaranteed for too long, though.

After some initial processing, the input text is processed in 512-bit blocks (which are further divided into 16 32-bit sub-blocks). The output of the algorithm is a set of four 32-bit blocks, which make up the 128-bit message digest.

The Working of MD5

Step 1: Padding The first step in MD5 is to add padding bits to the original message. The aim of this step is to make the length of the original message equal to a value, which is 64 bits less than an exact multiple of 512. For example, if the length of the original message is 1000 bits, we add a padding of 472 bits to make the length of the message 1472 bits. This is because, if we add 64 to 1472, we get 1536, which is a multiple of 512 (because 1536 = 512 x 3).
Thus, after padding, the original message will have a length of 448 bits (64 bits less than 512), 960 bits (64 bits less than 1024), 1472 bits (64 bits less than 1536), etc.

The padding consists of a single 1 bit, followed by as many 0 bits, as required. Note that padding is always added, even if the message length is already 64 bits less than a multiple of 512. Thus, if the message were already of a length of say 448 bits, we will add a padding of 512 bits to make its length 960 bits. Thus, the padding length is any value between 1 and 512.

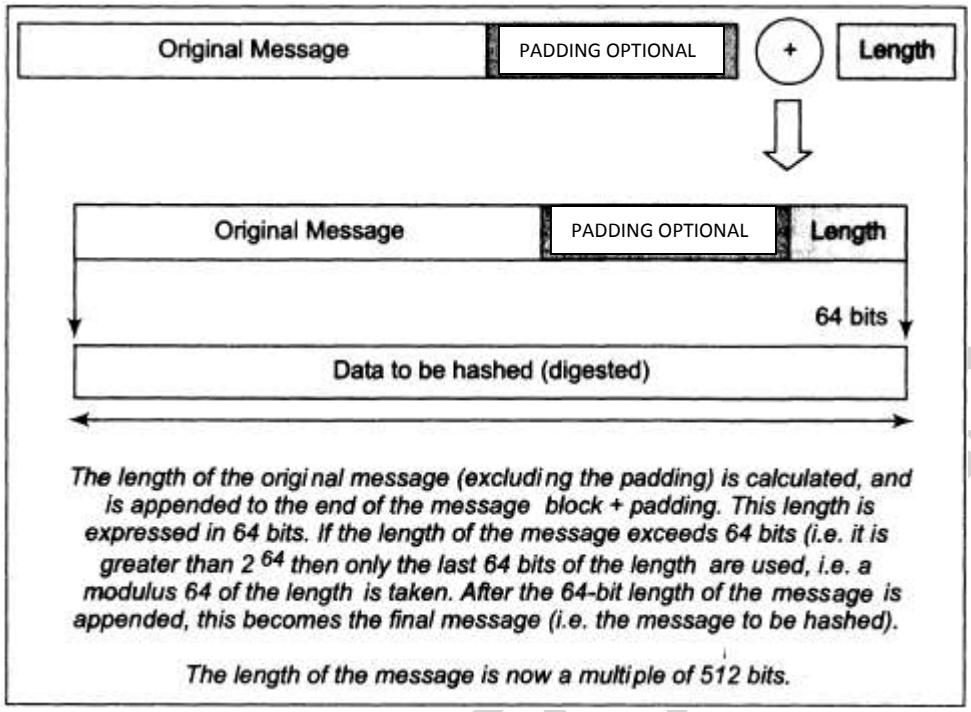


Step 2: Append Length After padding bits are added, the next step is to calculate the original length of the message, and add it to the end of the message, after padding. How is this done?

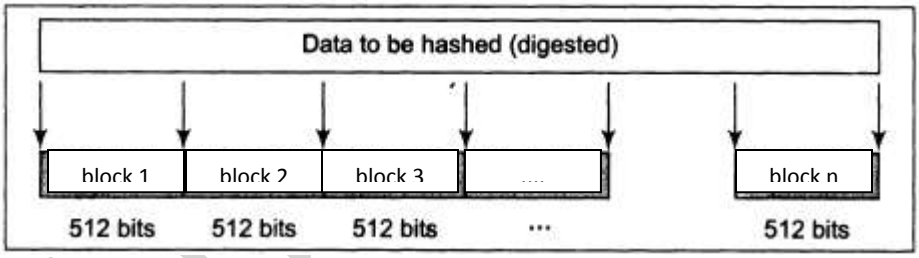
The length of the message is calculated, excluding the padding bits (i.e. it is the length before the padding bits were added). For instance, if the original message consisted of 1000 bits, and we added a padding of 472 bits to make the length of the message 64 bits less than 1536 (a multiple of 512), the length is considered 1000, and not 1472, for the purpose of this step.

This length of the original message is now expressed as a 64-bit value, and these 64 bits are appended to the end of the original message + padding. This is shown in Fig. 4.26. Note that if the length of the message exceeds 264 bits (i.e. 64 bits are not enough to represent the length, which is possible in the case of a really long message), we use only the low-order 64 bits of the length. That is, in effect, we calculate the length mod 264 in that case.

We will realize that the length of the message is now an exact multiple of 512. This now becomes the message whose digest will be calculated.



Step 3: Divide the Input into 512-bit Blocks Now, we divide the input message into blocks, each of length 512 bits.

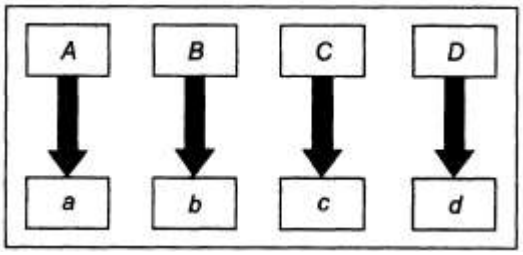


Step 4: Initialize Chaining Variables In this step, four variables (called chaining variables) are initialized. They are called *A* , *B*, C and *D.* Each of these is a 32-bit number. The initial hexadecimal values of these chaining variables.

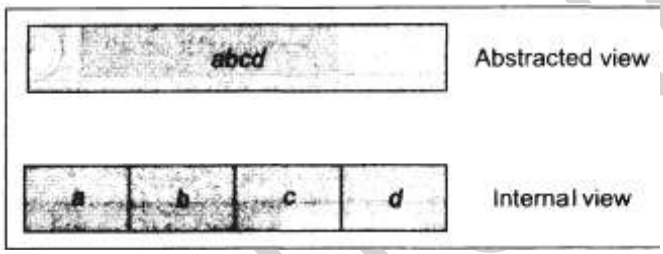
A	Hex	01	23	45	67
B	Hex	89	AB	CD	EF
C	Hex	FE	DC	BA	98
D	Hex	76	54	32	10

Step 5: Process Blocks After all the initializations, the real algorithm begins. It is quite complicated, and we shall discuss it step by step to simplify it to the maximum extent possible. There is a loop that runs for as many 5 12-bit blocks as are in the message.

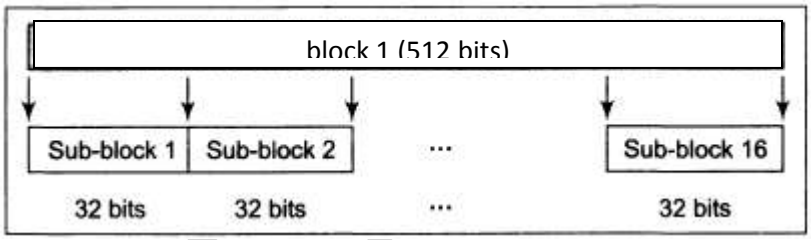
Step 5.1 Copy the four chaining variables into four corresponding variables, *a*, *b*, *c* and *d* (note the smaller case). Thus, we now have *a - A*, *b = B*, *c* = C and *d = D.*



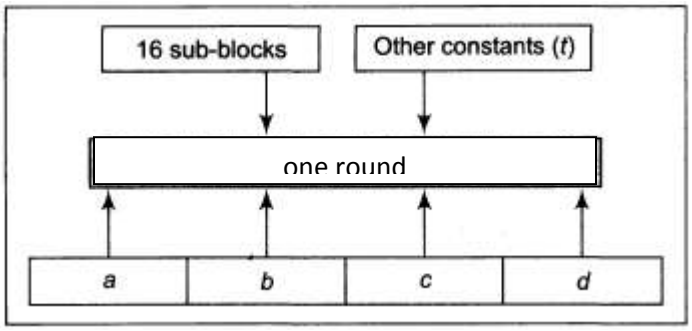
Actually, the algorithm considers the combination of *a*, *b*, *c* and *d* as a 128-bit single register (which we shall call *abcd).* This register *(abcd)* is useful in the actual algorithm operation for holding intermediate as well as final results.



Step 5.2 Divide the current 512-bit block into 16 sub-blocks. Thus, each sub-block contains 32 bits



Step 5.3 Now, we have four *rounds.* In each round, we process all the 16 sub-blocks belonging to a block. The inputs to each round are (a) all the 16 sub-blocks, (b) the variables a, *b*, c, *d*, and (c) some constants, designated as *t.*



What is done in these four rounds? All the four rounds vary in one major way: step 1 of the four rounds has different processing. The other steps in all the four rounds are the same.

- In each round, we have 16 input sub-blocks, named M[0], M[1], ..., M[15], or in general, *M[i]*, where i varies from 1 to 15. As we know, each sub-block consists of 32 bits.
- Also, *t* is an array of constants. It contains 64 elements, with each element consisting of 32 bits. We denote the elements of this array *t* as t[1], *t[*2],... *t[*64], or in general as *t[k]*, where *k* varies from 1 to 64. Since there are four rounds, we use 16 out of the 64 values of *t* in each round.

Let us summarize these iterations of all the four rounds. In each case, the output of the intermediate as well as the final iteration is copied into the register *abed.* Note that we have 16 such iterations in each round.

1. A process *P* is first performed on *b*, *c* and *d.* This process *P* is different in all the four rounds.
2. The variable *a* is added to the output of the process *P* (i.e. to the register *abcd).*
3. The message sub-block *M[i]* is added to the output of step 2 (i.e. to the register *abcd).*
4. The constant *t[k]* is added to the output of step 3 (i.e. to the register *abcd).*
5. The output of step 4 (i.e. the contents of register *abcd)* is circular-left shifted by 5 bits. (The value of s keeps changing).
6. The variable *b* is added to the output of step 5 (i.e. to the register *abcd).*
7. The output of step 6 becomes the new *abcd* for the next step.

We can mathematically express a single MD5 operation as follows:

a = *b* + *((a* + Process *P* (b, *c*, *d)* + *M[i]* + *T[k])* <<< *s)*
where,

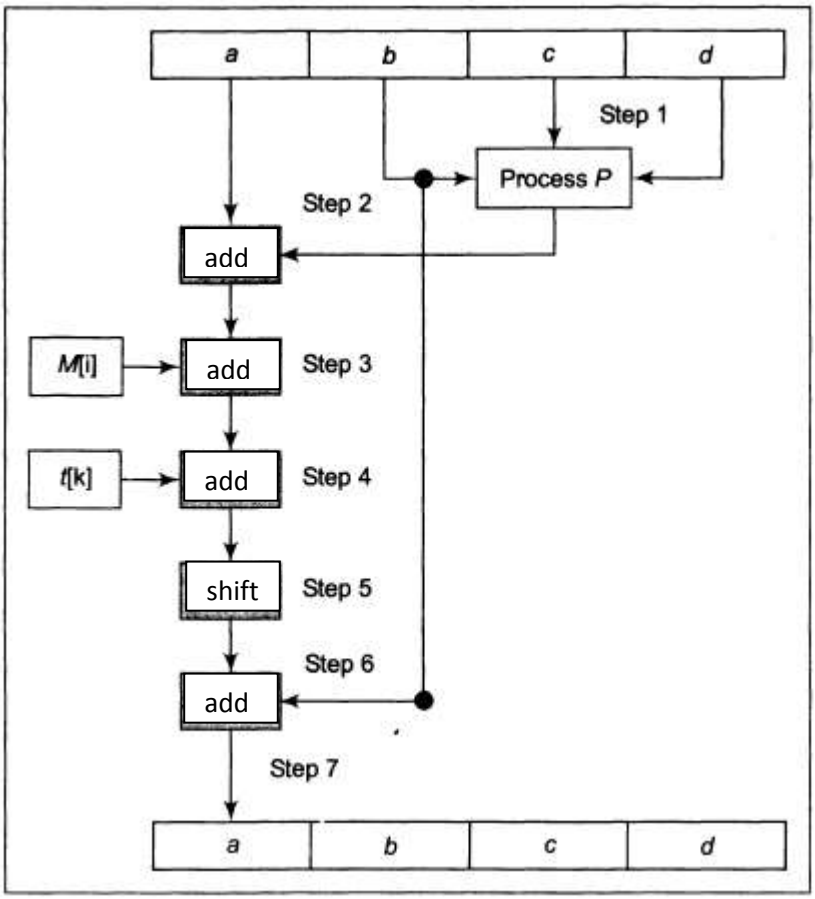
a, *b* ,*c* ,*d*	= Chaining variables, as described earlier
Process *P*	= A non-linear operation, as described subsequently
M[i]	= *M[q* x 16 + i], which is the ith 32-bit word in the *q*th 512-bit block of the message
t[k]	= A constant, as discussed subsequently
<<<*s*	= Circular-left shift by *s* bits

Understanding the Process P As we can see, the most crucial aspect here is to understand the process *P*, as it is different in the four rounds. In simple terms, the process *P* is nothing but some basic Boolean operations on *c* and *d.*

Note that in the four rounds, only the process *P* differs. All the other steps remain the same. Thus, we can substitute the actual details of process *P* in each of the round, and keep everything else constant.

Round	Process P
1	(b AND c) OR ((NOT b) AND (d))
	(b AND d) OR (c AND (NOT d))
3	B XOR c XOR d
4	C XOR (b OR (NOT d))

One MD5 operation



Secure Hash Algorithm (SHA)

1. Introduction

The National Institute of Standards and Technology (NIST) along with NSA developed the Secure Hash Algorithm (SHA). In 1993, SHA was published as a Federal Information Processing Standard (FIPS PUB 180). It was revised to FIPS PUB 180-1 in 1995, and the name was changed to SHA-1.SHA is a modified version of MD4, and its design closely resembles MD4.

SHA works with any input message that is less than 2s4 bits in length. The output of SHA is a message digest, which is 160 bits in length (32 bits more than the message digest produced by MD5). The word Secure in SHA was decided based on two features. SHA is designed to be computationally infeasible to

(a) obtain the original message, given its message digest, and
(b) find two messages producing the same message digest.

2. *The Working of SHA*

As we have mentioned before, SHA is closely modeled after MD5. Therefore, we shall not discuss in detail those features of SHA, which are similar to MD5. Instead, we shall simply mention them and point out the differences. The reader can go back to the appropriate descriptions of MD5 to find out more details.

Step 1: Padding Like MD5, the first step in SHA is to add padding to the end of the original message in such a way that the length of the message is 64 bits short of a multiple of 512. Like MD5, the padding is always added, even if the message is already 64 bits short of a multiple of 512.

Step 2: Append Length The length of the message excluding the length of the padding is now calculated and appended to the end of the padding as a 64-bit block.

Step 3: Divide the Input into 512-bit Blocks The input message is now divided into blocks, each of length 512 bits. These blocks become the input to the message-digest processing logic.

Step 4: Initialize Chaining Variables Now, five *chaining variables A* through *E* are initialized. Remember that we had four chaining variables, each of 32 bits in MD5 (which made the total length of the variables 4 x 32 = 128 bits). Recall that we stored the intermediate as well as the final results into the combined register made up of these four chaining variables, i.e. *abcd.* Since in the case of SHA, we want to produce a message digest of length 160 bits, we need to have five chaining variables here (5 x 32 = 160 bits). In SHA, the variables *A* through *D* have the same values as they had in MD5. Additionally, *E* is initialized to Hex C3 *D2 El F0.*

Step 5: Process Blocks Now the actual algorithm begins. Here also, the steps are quite similar to those in MD5.

Step 5.1 Copy the chaining variables *A-E* into variables *a-e.* The combination of *a-e*, called *abcde*, will be considered as a single register for storing the temporary intermediate as well as the final results.

***Step* 5.2** Now, divide the current 5 12-bit block into 16 sub-blocks, each consisting of 32 bits.

Step 5.3 SHA has four rounds, each round consisting of 20 steps. Each round takes the current 5 12-bit block, the register *abcde* and a constant *K[t]* (where t = 0 to 79) as the three inputs. It then updates the contents of the register *abcde* using the SHA algorithm steps. Also notable is the fact that we had 64 constants defined as *t* in MD5. Here, we have only four constants defined for *K[t]*, one used in each of the four rounds.

Step 5.4 SHA consists of four rounds, each round containing 20 iterations. This makes it a total of 80 iterations. The logical operation of a single SHA iteration looks as shown.

Mathematically, an iteration consists of the following operations:

abcde = *(e* + Process *P* + s^5 *(a)* + *W[t]* + *K[t])*, *a*, s^{30} *(b)*, *c*, *d*

where,

abcde = The register made up of the five variables *a*, *b*, c, and *e*

Process *P* = The logical operation, which we shall study later

s^t = Circular-left shift of the 32-bit sub-block by *t* bits

W[t] = A 32-bit derived from the current 32-bit sub-block, as we shall study later

K[t] = One of the five additive constants, as defined earlier

We will notice that this is very similar to MD5, with a few variations (which are induced to try and make SHA more complicated in comparison with MD5). We have to now see what are the meanings of the process *P* and *W[t]* in the above equation.

illustrates the process *P. (*Process P in each SHA-1 *round)*

Round	Process P
1	(b AND c) O R ((NOT b) AND (d))
2	B XOR c XOR d
3	(b AND c) OR (b and D) OR (c AND d)
4	B XOR c XOR d

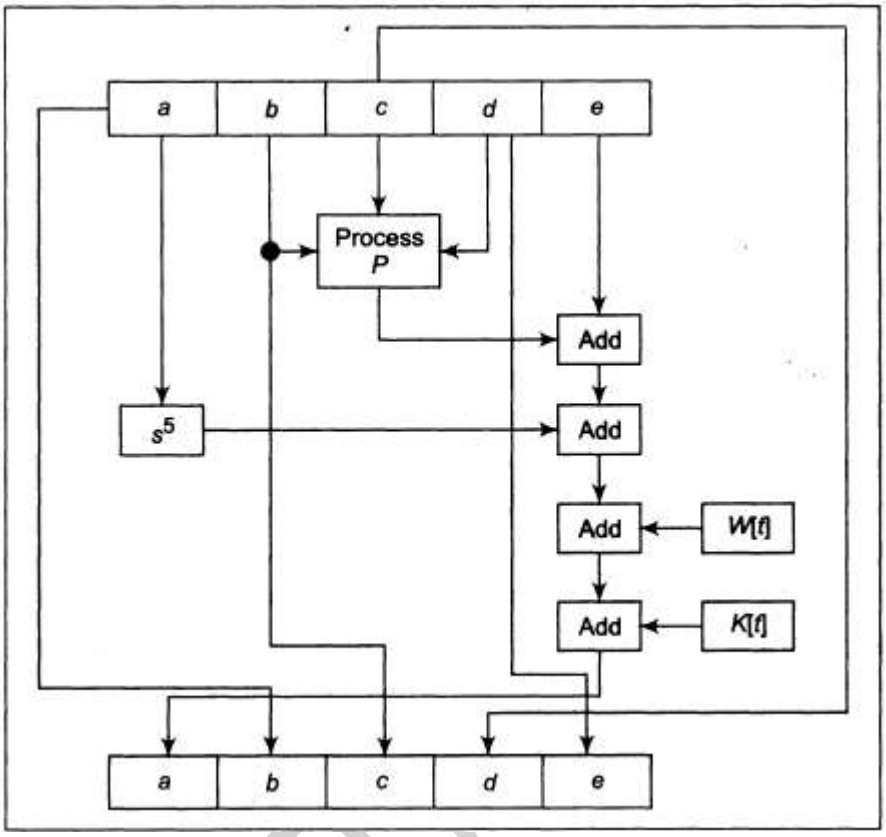
The values of *W[t*] are calculated as follows:

For the first 16 words of *W* (i.e. *t* = 0 to 15), the contents of the input message sub-block *M[t]* become the contents of *W[t]* straightaway. That is, the first 16 blocks of the input message M are copied to *W.* The remaining 64 values of *W* are derived using the equation:

W[t] = s^1 *(W [t*- 16] XOR *W[t* - 14] XOR *W[t* - 8] XOR *W[t* - 3])

As before, s^1 indicates a circular-left shift (i.e. rotation) by 1 bit position.

Single SHA-1 iteration



Comparison of MDS and SHA-1

Point of discussion	MD5	SHA
Message-digest length in bits	128	160
Attack to try and find the original message given a message digest	Requires 2^{128} operations to break in	Requires 2^{160} operations to break in, therefore more secure
Attack to try and find two messages producing the same message digest	Requires 2^{64} operations to break in	Requires 2^{80} operations to break in
Successful attacks so far	There have been reported attempts to some extent	No such claims so far
Speed	Faster (64 iterations, and 128-bit buffer)	Slower (80 iterations, and 160-bit buffer)
Software implementation	Simple, does not need any large programs or complex tables	Simple, does not need any large programs or complex tables

SHA-512

The SHA-512 algorithm takes a message of length 2^{128} bits, and produces a message digest of size 512 bits. The input is divided into blocks of size 1024 bits each.

SHA-512 is closely modeled after SHA-1, which itself is modeled on MD5. Therefore, we shall not discuss in detail those features of SHA-512, which are similar to these two algorithms. Instead, we shall simply mention them and point out the differences.

Step 1: Padding

Like MD5 and SHA-1, the first step in SHA is to add padding to the end of the original message in such a way that the length of the message is 128 bits short of a multiple of 1024. Like MD5 and SHA-1, the padding is always added, even if the message is already 128 bits short of a multiple of 1024.

Step 2: Append Length

The length of the message excluding the length of the padding is now calculated and appended to the end of the padding as a 128-bit block. Hence, the length of the message is exactly a multiple of 1024 bits.

Step 3: Divide the Input into 1024-bit Blocks

The input message is now divided into blocks, each of length 1024 bits. These blocks become the input to the message-digest processing logic.

Step 4: Initialize Chaining Variables

Now, eight *chaining variables*, a through *h*, are initialized. Remember that we had (a) four chaining variables, each of 32 bits in MD5 (which made the total length of the variables 4 x 32 = 128 bits), and (b) five chaining variables each of 32 bits (which made the total length of the variables 5 x 32 = 160 bits) in SHA-1. Recall that we stored the intermediate as well as the final results into the combined register made up of these chaining variables, i.e. *abed* in MD5 and abcde in SHA-1. Since in the case of SHA-256, as we want to produce a message digest of length 512 bits, we need to have eight chaining variables, each containing 64 bits here (8 x 64 = 512 bits). In SHA-512, these eight variables have values as shown.

A = 6AQ9E661F3BCC908	B = BB67AE85S4CAA73B
C = 3C6EF372FE94F82B	D = A54FF53A5F1D36F1
E = 51QE521FADE6B2D1	F -9B05688C2B3E6C1F
G = 1F83D9ABFB41BD6B	H = 5BE0CD19131E2H9

Step 5: Process Blocks

Now the actual algorithm begins. Here also, the steps are quite similar to those in MD5 and SHA-1.

Step 5.1 Copy the chaining variables *A-H* into variables *a-h.* The combination of *a-h*, called *abcdefgh*, will be considered as a single register for storing the temporary intermediate as well as the final results.

Step 5.2 Now, divide the current 1024-bit block into 16 sub-blocks, each consisting of 64 bits.

Step 5.3 SHA-512 has 80 rounds. Each round takes the current 1024-bit block, the register *abcdefgh* and a constant *K[t*] (where *t* = 0 to 79) as the three inputs. It then updates the contents of the register *abcdefgh* using the SHA-512 algorithm steps. The operation of a single round is shown.

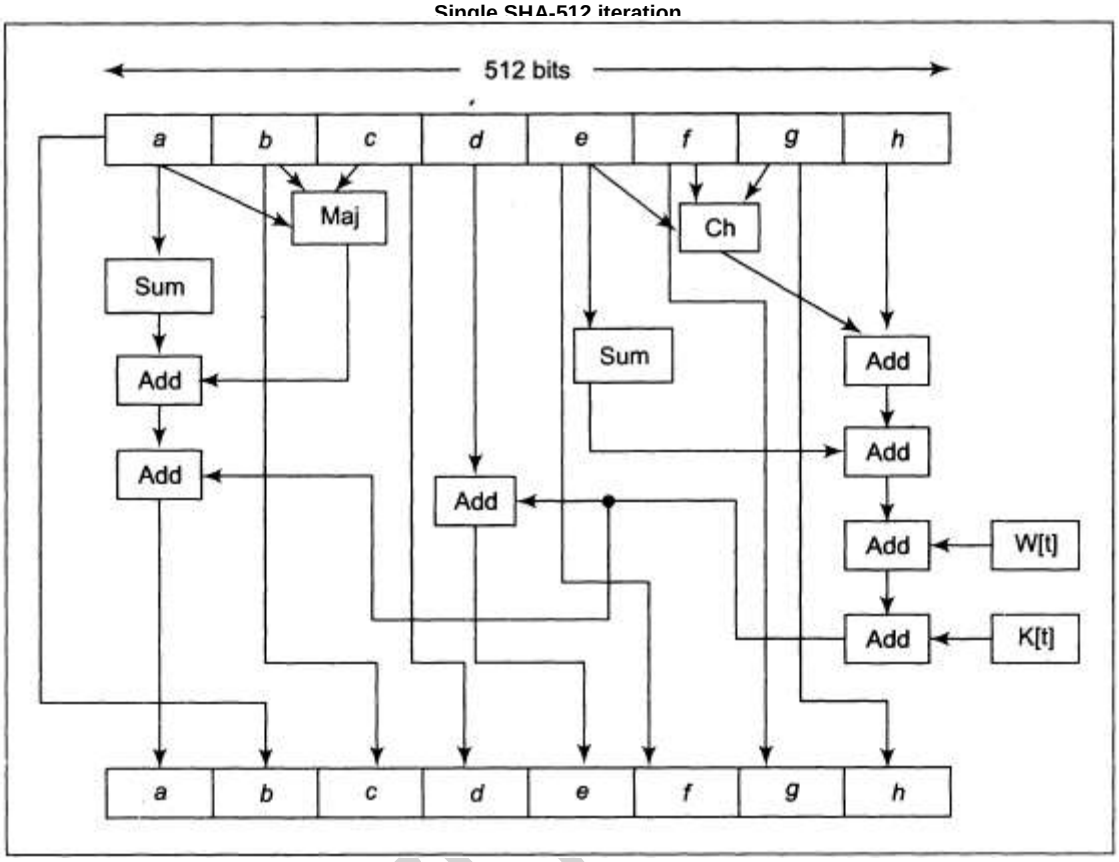
Each round consists of the following operations

Temp1 = *h* + Ch *(e ,f g)* + Sum *(e_i* for *i* = 1 to 512) + W_t + K_t
Temp2 = Sum *(a_i* for i = 0 to 512) + Maj *(a, b, c)*
a = Temp1 + Temp2
b = *a*
c = *b*
d= *c*
e = *d*+ Temp1
f = *e*
g = *f*
h = *g*

where:

t = Round number
Ch *(e,f, g)* = *(e* AND *f)* XOR (NOT *e* AND *g)*
Maj (a, *b, c)* = *(a* AND *b)* XOR *(,a* AND c) XOR (ft AND *c)*
Sum *(a_i)* = ROTR (a by 28 bits) XOR ROTR *(a.* by 34 bits) XOR (ROTR *a.* by 39 bits)
Sum *(e_i)* = ROTR (e by 14 bits) XOR ROTR *(e.* by 18 bits) XOR (ROTR *e.* by 41 bits)
ROTR (x) = Circular right shift, i.e. rotation, of the 64-bit array x the specified number of bits
W_t = 64-bit word derived from the current 512-bit input block
K_t = 64-bit additive constant
+ (or Add) = Addition mod 2^{64}

Single SHA-512 iteration



The 64-bit word values for W_t are derived from the 1024-bit message using certain mappings, which we shall not describe here. Instead, we will simply point out this:

- For the first 16 rounds (0 to 15), the value of W_t is equal to the corresponding word in the message block.
- For the remaining 64 steps, the value of W_t is equal to the circular left shift by one bit of the XOR of the four preceding values of W_t with two of them subjected to shift and rotate operations.

This makes the message digest more complex and difficult to break.

SHA-3

The MD5 algorithm is broken, but nobody has been able to break SHA-1 so far. However, because SHA-1 is quite similar in nature to MD5, it is feared that in the future, it can be broken as well. Hence, more and more people have started using the next version of the SHA family of algorithms, collectively known as **SHA-2** (i.e. SHA-256, SHA-384, and SHA-512). SHA-512 is considered to be the toughest of them to break. However, considering the fact that at some point of time in the future, SHA-2 would also get broken, a search for the next leap in message-digest algorithms is being taken by searching for **SHA-3.**

1. The application need not have to make extensive changes to replace SHA-2 with SHA-3. In other words, we should be able to simply replace the earlier algorithms with SHA-3. This automatically means that SHA-3 must support message digests of lengths 224, 256, 384, and 512 bits.

2. The basic characteristic of being able to process smaller blocks of original text to generate the message digest in SHA-2 needs to be carried forward in SHA-3. Other expectations from SHA-3 are that it should be as secure as possible while defeating attacks that are likely to succeed on SHA-2. In other words, SHA-3 should be different in its operation from MD5 and SHA-1/SHA-2. It is also expected to work quite fast while consuming minimum resources. Also, it should be simple and it should be possible to add parameters to it to make it more flexible.

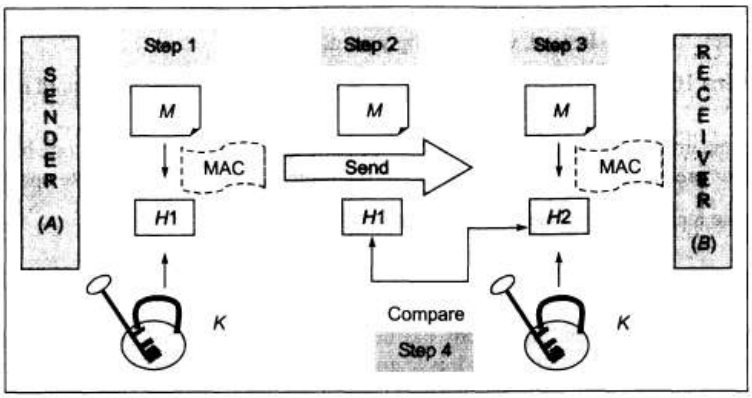
Message Authentication Code (MAC)

The concept of **Message Authentication Code (MAC)** is quite similar to that of a message digest. However, there is one difference. As we have seen, a message digest is simply a fingerprint of a message.

There is no cryptographic process involved in the case of message digests. In contrast, a MAC requires that the sender and the receiver should know a shared symmetric (secret) key, which is used in the preparation of the MAC. Thus, MAC involves cryptographic processing.

Let us assume that the sender *A* wants to send a message M to a receiver *B.* How the MAC processing works is shown

1. *A* and *B* share a symmetric (secret) key K, which is not known to anyone else. A calculates the MAC by applying key *K* to the message *M.*
2. A then sends the original message *M* and the MAC *H1* to *B.*
3. When *B* receives the message, *B* also uses *K* to calculate its own MAC *H2* over M
4. *B* now compares *H1* with *H2.* If the two match, *B* concludes that the message *M* has not been changed during transit. However, if $H1 \neq H2$, *B* rejects the message, realizing that the message was changed during transit.



The significances of a MAC are as follows:

1. The MAC assures the receiver (in this case, *B)* that the message is not altered. This is because if an attacker alters the message but does not alter the MAC (in this case, *H1)* then the receiver's calculation of the MAC (in this case, *H2)* will differ from it. Why does the attacker then not also alter the MAC? Well, as we know, the key used in the calculation of the MAC (in this case, *K)* is assumed to be known only to the sender and the receiver (in this case, *A* and *B).* Therefore, the attacker does not know the key, AT, and therefore, she cannot alter the MAC.

2. The receiver (in this case, *B)* is assured that the message indeed came from the correct sender (in this case, *A).* Since only the sender and the receiver *(A* and *B* , respectively, in this case) know the secret key (in this case, *K),* no one else could have calculated the MAC (in this case, HI) sent by the sender (in this case, *A).*

Interestingly, although the calculation of the MAC seems to be quite similar to an encryption process, it is actually different in one important respect. As we know, in symmetric-key cryptography, the cryptographic process must be reversible. That is, the encryption and decryption are the mirror images of each other. However, note that in the case of MAC, both the sender and the receiver are performing encryption process only. Thus, a MAC algorithm need not be reversible—it is sufficient to be a oneway function (encryption) only.

2. *The Working of HMAC*

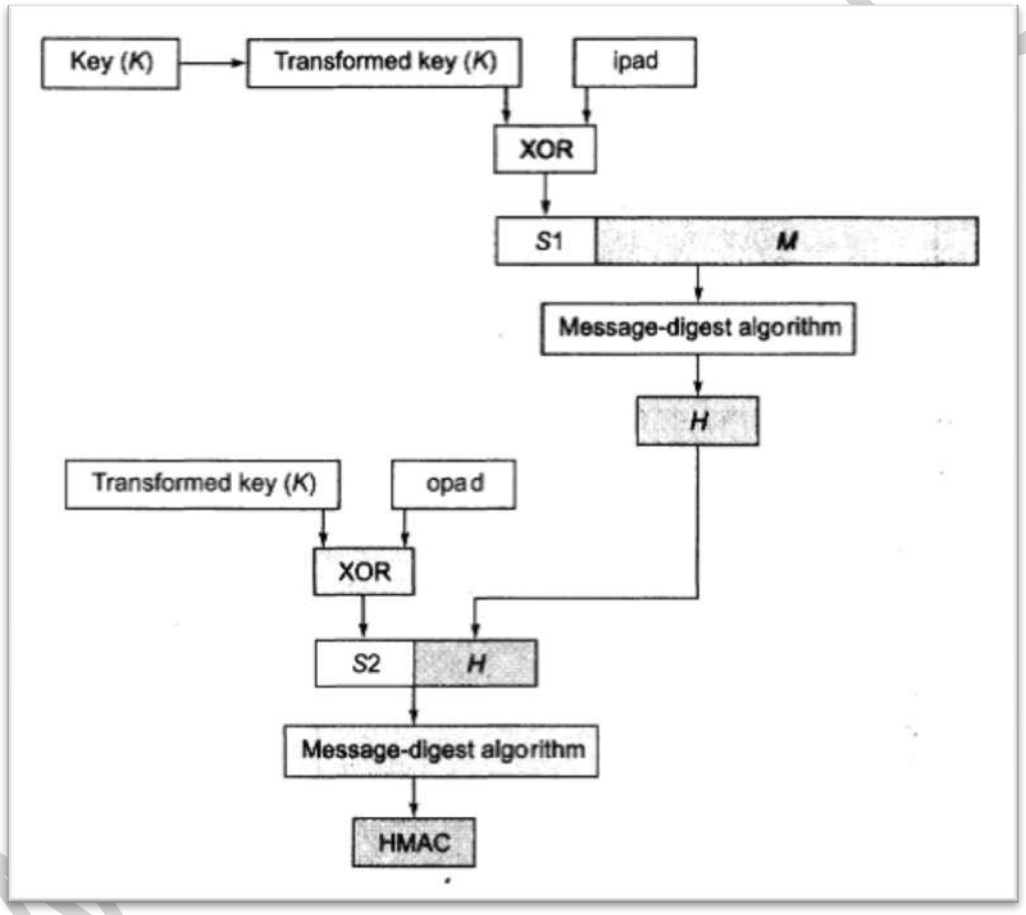
Let us now take a look at the internal working of HMAC. For this, let us start with the various variables that will be used in our HMAC discussion.

MD	=	The message digest/hash function used (e.g. MD5, SHA-1, etc.)
M	=	The input message whose MAC is to be calculated
L	=	The number of blocks in the message *M*
b	=	The number of bits in each block
K	=	The shared symmetric key to be used in HMAC

ipad = A string 00110110 repeated *b*/8 times
opad = A string 01011010 repeated *b*/8 times

Step 1: Make the length of *K* equal to *b* The algorithm starts with three possibilities, depending on the length of the key *K:*

Length o f K < b In this case, we need to expand the key *(.K)* to make the length of *K* equal to the number of bits in the original message block (i.e. *b).* For this, we add as many 0 bits as required to



WORKING OF HMAC

KNAPSACK ALGORITHM

Actually, Ralph Merkle and Martin Heilman developed the first algorithm for public-key encryption, called the **Knapsack algorithm.** It is based on the **Knapsack problem.** This is actually a simple problem. Given a pile of items, each with different weights, is it possible to put some of them in a bag (i.e. knapsack) in such a way that the knapsack has a certain weight?

That is, if M1, *M2*, ..., *Mn* are the given values and *S* is the sum, find out *bi* so that:
S = b1M1 + b2M2 + ... + bnMn

Each *bi* can be 0 or 1. A 1 indicates that the item is in the knapsack, and a 0 indicates that it is not. A block of plain text equal in length to the number of items in the pile would select the items in the knapsack. The cipher text is the resulting sum. For example, if the knapsack is 1,7, 8, 12, 14, 20 then the plain text and the resulting cipher text.

Plain text	0 1 1 0 1 1	1 1 1 0 0 0	0 1 0 1 1 0
Knapsack	1 7 8 12 14 20	1 7 8 12 14 20	1 7 8 12 14 20
Cipher text	7 + 8 + 14 + 20 = 49	1 + 7 + 8 = 16	7 + 12 + 14 = 33

ELLIPTIC CURVE CRYPTOGRAPHY

The main problem of conventional public key cryptography systems is that the key size has to be sufficient large in order to meet the high-level security requirement.

This results in lower speed and consumption of more bandwidth

Solution:

Elliptic Curve Cryptography system

- An elliptic curve is a curve that's also naturally a group.
- The group law is constructed geometrically.
- Elliptic curves have (almost) nothing to do with ellipses, so put ellipses and conic sections out of your thoughts.
- Elliptic curves appear in many diverse areas of mathematics, ranging from number theory to complex analysis, and from cryptography to mathematical physics.

Points on Elliptic Curves

The Equation of an Elliptic Curve

An Elliptic Curve is a curve given by an equation of the form

$$y^2 = x^3 + Ax + B$$

ELGAMAL DIGITAL SIGNATURE

The **ElGamal digital-signature** scheme uses the same keys, but a different algorithm. The algorithm creates two digital signatures. In the verification step, these two signatures are tallied. The key-generation process here is the same as what we had discussed earlier and hence we would not repeat the discussion. The public key remains (E1, *E2*, *P)* and the private key continues to be *D.*

Signature

The signature process works as follows:

1. The sender selects a random number *R.*
2. The sender computes the first signature S1 using the equation S1 = $E1^R$ mod *P.*
3. The sender computes the second signature *S2* using the equation *S2* = *(M - D x S1) x* R^{-1} mod *(P* - 1), where *M* is the original message that needs to be signed.
4. The sender sends M, *S1*, and *S2* to the receiver.

For example, let *E1* = 10, *E2* = 4, *P* = 19, *M* = 14, *D* = 16, and *R* = 5.
Then we have:

S1 = E1 mod P = 10^5 mod 19 = 3

- Elliptic curves can have points with coordinates in any field, such as $\mathbb{F}_p$, $\mathbb{Q}$, $\mathbb{R}$, or $\mathbb{C}$.
- Elliptic curves with points in $\mathbb{F}_p$ are finite groups.
- **Elliptic Curve Discrete Logarithm Problem (ECDLP)** is the discrete logarithm problem for the group of points on an elliptic curve over a finite field.
- The best known algorithm to solve the ECDLP is exponential, which is why elliptic curve groups are used for cryptography.
- More precisely, the best known way to solve ECDLP for an elliptic curve over $\mathbb{F}_p$ takes time $O(\sqrt{p})$.
- The goal of these talks is to tell you something about the theory of elliptic curves, with an emphasis on those aspects that are of interest in cryptography.

S2 = *(M - D* x *S1) x* R^{-1} mod *(P* - 1) = (14 - 16 x 3) x 5^{-1} mod 18 = 4
Hence, the signature is (S1, S2) i.e. (3, 4).
This is sent to the receiver.

Verification

The verification process works as follows:

1. The receiver performs the first part of verification called *V1* using the equation *V1* = $E1^M$ mod *P.*
2. The receiver performs the second part of verification called as *V2* using the equation *V2* = $E2^{S1}$ *x*

$S1^{S2}$ mod *P.*

In our example:

$V1 = E1^{M} \ mod \ P = 10^{14}$ mod 19= 16

$V2 = E2^{S1} \ x \ S1^{S2}$ mod P = 4^3 x 3^4 mod 19 = 5184 mod 19= 16

Since *V1 = V2*, the signature is considered valid.

PROBLEMS WITH THE PUBLIC-KEY EXCHANGE

man in middle attack is the problem associated with the public key exchange