# PROJECT REPORT

## "CRYPTOGRAPHY AND IT'S APPLICATION IN RANGE EOTS"


**INTEGRATED TEST RANGE**

**DEFENSE RESEACH AND DEVELOPMENT ORGANISATION**

**CHANDIPUR - 756025**

**Under the Guidance of**

**Dr Arun Kumar Ray**

**(Scientist-G, Group Director, EOTS)**

**Under the Supervision of**

**Shri Ganesh Ramani**

**(Scientist – E, EOTS)**

**Submitted By**

**Rohan Mahapatra**

**(Electronics And Telecommunication Engineering, VSSUT**

**Regd no. – 2202070091)**

**Aryaman Jena**

**(Computer Science Engineering, VSSUT**

**Regd no. – 2202041060)**

**Abhijit Behera**

**(Electronics And Telecommunication Engineering, VSSUT**

**Regd no. – 2202070134)**

# Certificate

This is to certify that Rohan Mahapatra(ETC), Abhijit Behera(ETC) and Aryaman Jena(CSE) from Veer Surendra Sai University of Technology (VSSUT), Burla have undergone a summer internship training and submitted a project report on “Cryptography and It’s Applications in Range EOTS" under the Department of Electro-Optical Tracking System(EOTS) of Integrated Test Range, DRDO, Chandipur from 2nd June 2025 to 1st July 2025. They have shown keen interest in their assignments of the project work and were very sincere, hard-working and inquisitive.

I wish them every success in life.

Dr Arun Kumar Ray
( Scientist-G, Group Director, EOTS )

Shri Ganesh Ramani
(Scientist- E , EOTS)

# Acknowledgment

We owe our sincere gratitude to all those who have been a source of encouragement and support throughout this project. We are extremely grateful to **Dr Arun Kumar Ray**, Scientist-G, GD, EOTS division of Integrated Test Range for having suggested the topic of our project and for his persistent guidance and direction.

We are indebted to **Shri Ganesh Ramani**, Scientist E of EOTS Divison of ITR for their constant help and motivation right from the beginning of this project until this stage. Their valuable suggestions are the most important input which has assisted in all respects. We are immensely grateful to our parents, whose support has led us to successfully utilize this opportunity.

**Rohan Mahapatra**
**(Electronics And Telecommunication Engineering, VSSUT)**

**Aryaman Jena**
**(Computer Science Engineering, VSSUT)**

**Abhijit Behera**
**(Electronics And Telecommunication Engineering, VSSUT)**

# DRDO

DRDO is the R&D wing of Ministry of Defense, Govt of India, with a vision to empower India with cutting-edge defense technologies and a mission to achieve self-reliance in critical defense technologies and systems, while equipping our armed forces with state-of-the-art weapon systems and equipment in accordance with requirements laid down by the three Services. DRDO's pursuit of self-reliance and successful indigenous development and production of strategic systems and platforms such as Agni and Prithvi series of missiles; light combat aircraft, Tejas; multi-barrel rocket launcher, Pinaka; air defense system, Akash; a wide range of radars and electronic warfare systems; etc., have given quantum jump to India's military might, generating effective deterrence and providing crucial leverage.

"Balasya Mulam Vigyanam"—the source of strength is science-drives the nation in peace and war. DRDO has firm determination to make the nation strong and self-reliant in terms of science and technology, especially in the field of military technologies. DRDO was formed in 1958 from the amalgamation of the then already functioning Technical Development Establishment (TDEs) of the Indian Army and the Directorate of Technical Development & Production (DTDP) with the Defense Science Organization (DSO). DRDO was then a small organization with 10 establishments or laboratories. Over the years, it has grown multi-directionally in terms of the variety of subject disciplines, number of laboratories, achievements and stature.

Today, DRDO is a network of around 41 laboratories and 5 DRDO Young Scientist Laboratories (DYSLs), which are actively engaged in the development of advanced defense technologies across diverse disciplines. These include aeronautics, armaments, electronics, combat vehicles, engineering systems, instrumentation, missiles, advanced computing and simulation, special materials, naval systems, life sciences, training, information systems, and agriculture. Several major projects—such as the development of missiles, armaments, light combat aircraft, radars, and electronic warfare systems—are currently underway, and significant achievements have already been made in many of these areas.

# Integrated Test Range

The Integrated Test Range (ITR), a well-equipped Test and Evaluation centre of DRDO, was established to provide safe and reliable launch facilities for the performance evaluation of rockets, missiles, and airborne weapon systems. One of the most critical aspects of design evaluation for various flight vehicles is the precise tracking of flying objects—from take-off to impact—throughout their entire trajectory, along with accurate measurement of their health parameters. To achieve this, various high-performance range instrumentation systems, such as Electro-Optical Tracking Systems (EOTS), radar, telemetry, and central computer systems, are deployed and configured in real-time to capture and process vital data. These systems provide the precise location and health parameters of airborne targets throughout the flight path.

The implementation of an ISO 9001:2015 quality management system at ITR has bolstered user and project confidence in the quality and reliability of its range operations. To support a diverse array of missions, numerous well-established test procedures and simulations are routinely conducted to maintain the Range in a constant state of readiness. These state-of-the-art instruments and support facilities enable ITR to consistently deliver accurate and crucial data to its projects, users, and customers.

# Contents

# Introduction

## Electro-Optical Targeting System (EOTS)

The **Electro-Optical Tracking System (EOTS)** is a critical component used for the **detection, tracking, and recording** of high-speed objects such as missiles, aircraft, or projectiles during flight trials. It uses a combination of optical sensors and electronic systems to observe and analyse the trajectory and performance of flight vehicles in real time.

Key features of EOTS include:

- **High-resolution optical cameras** (daylight and thermal/IR)
- **Precision tracking mechanisms** with auto-tracking capabilities
- **Long-range zoom lenses** for detailed observation
- **Real-time data acquisition and recording systems**
- **Integration with radar and telemetry systems**

## Components crucial to EOTS

| Component | Function |
|---|---|
| **FLIR (Forward-Looking Infrared)** | Thermal imaging for night vision and target identification. |
| **IRST (Infrared Search and Track)** | Passive detection of heat-emitting targets (like aircraft) at long range. |
| **Laser Rangefinder/Designator** | Measures distance and guides laser-guided weapons with pinpoint accuracy. |
| **Optical Sensors** | Provide high-definition visual imagery and magnification of targets. |
| **Image Processing Software** | Assists in real-time object recognition, stabilization, and threat analysis. |

# Range EOTS at ITR, DRDO

At the **Integrated Test Range (ITR), DRDO** in Chandipur, EOTS forms the backbone of **Range Instrumentation Systems**. The **Range EOTS** refers to the strategically deployed electro-optical systems across the missile test range, which are specifically used to track missiles, UAVs, and other test vehicles during flight trials.

**Functions and Importance of Range EOTS:**

- **Tracking Flight Vehicles:** Provides real-time visual tracking data to complement radar and telemetry.
- **Flight Analysis:** Assists in measuring velocity, acceleration, and orientation of missiles during different phases of flight.
- **Safety Monitoring:** Helps in range safety operations by ensuring vehicles follow designated trajectories.
- **Post-Test Evaluation:** Recorded video footage and data assist in analysing performance and verifying mission objectives.
- **High Precision:** Due to the use of IR and visible spectrum tracking, EOTS can track targets even under low visibility conditions (night or cloudy weather).

# Purpose and Role of Cryptography in Test Range Scenario

In missile test ranges like ITR, cryptography ensures the security and integrity of sensitive data such as telemetry, command signals, and video feeds. It protects against unauthorized access, data tampering, and espionage by encrypting real-time transmissions and stored information. Cryptography also verifies the authenticity of users and systems, ensuring that only authorized personnel can access critical test data. This makes it an essential component for maintaining mission confidentiality and national security.

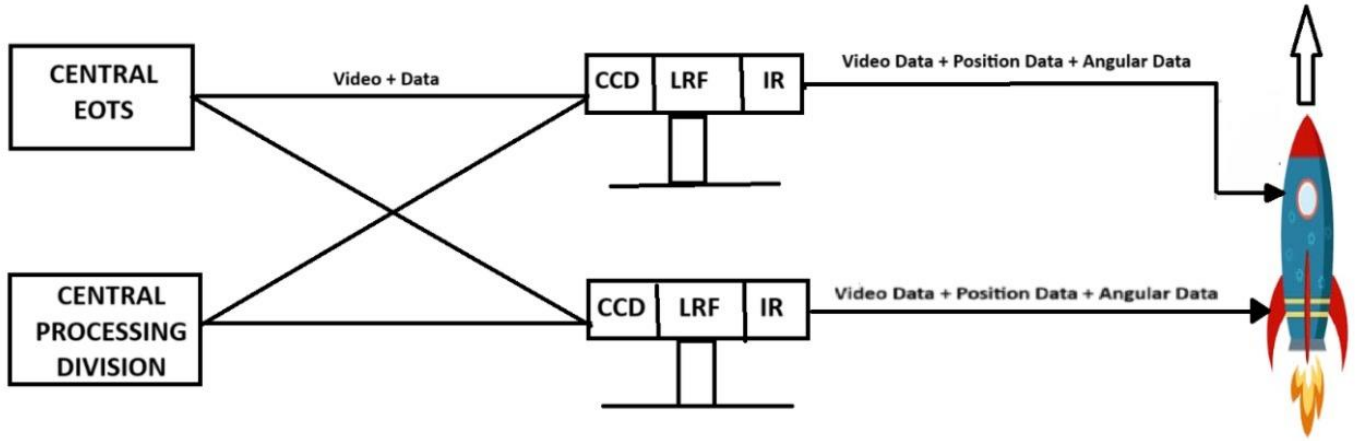



Fig: Range EOTS Operation Scenario

# CRYPTOGRAPHY

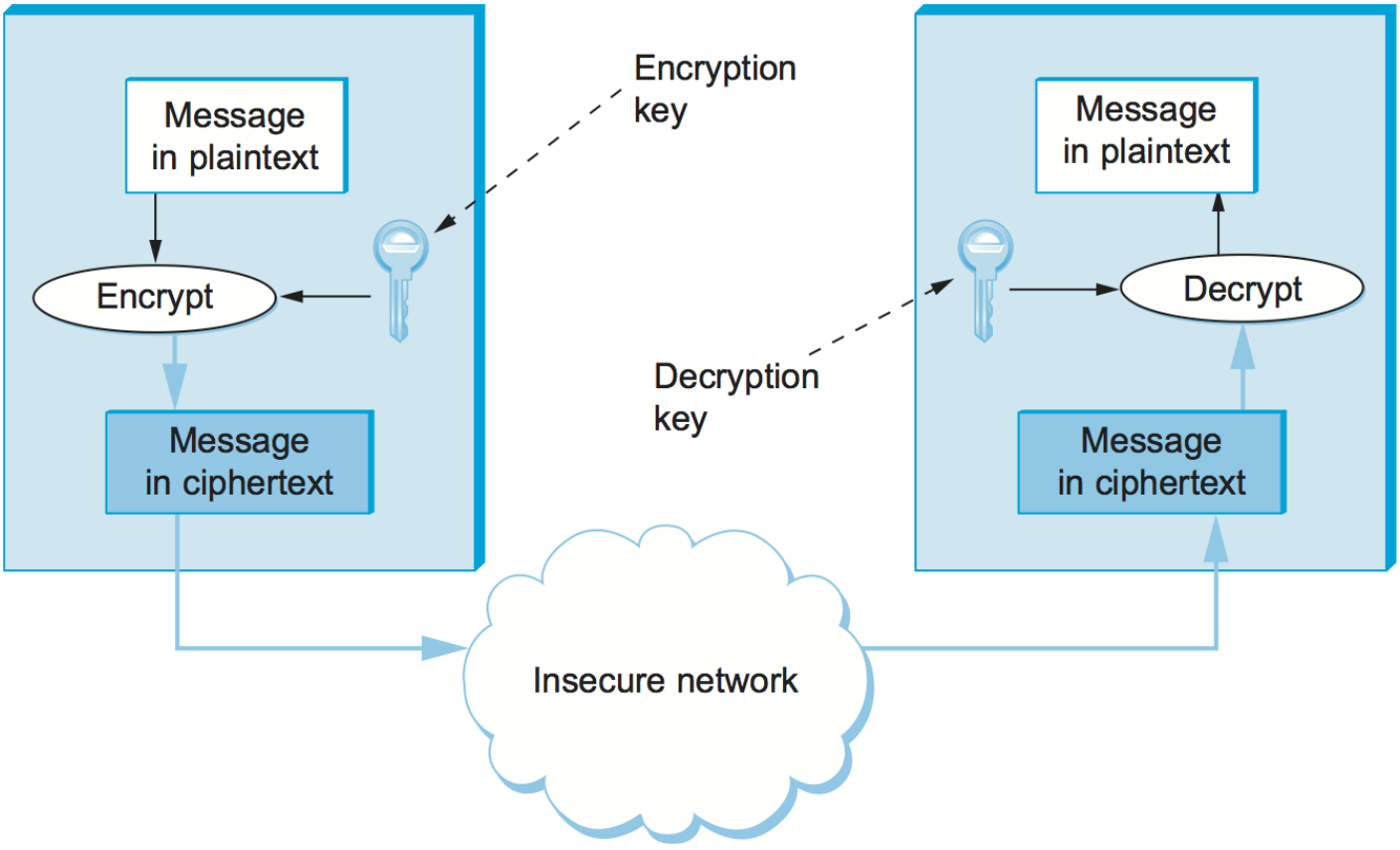
## What is Cryptography?

Cryptography is the science of securing information by transforming it into a form that unauthorized users cannot understand. It enables secure communication in the presence of malicious third parties, commonly known as adversaries.

Derived from the Greek words *kryptos* (hidden) and *graphein* (to write), cryptography essentially means “hidden writing.”

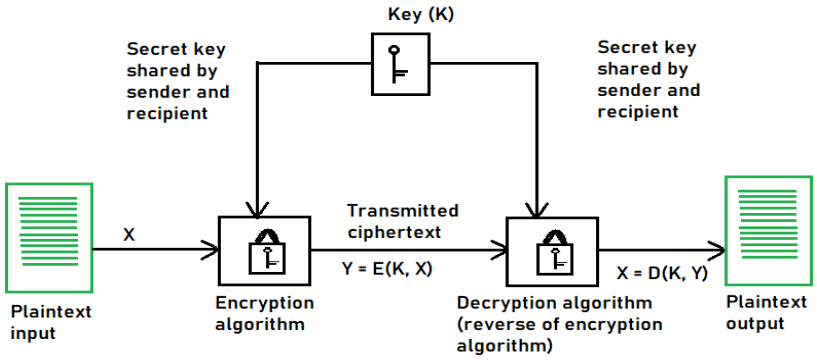
Cryptography, though vital to modern electronic communication, has ancient origins. The earliest known use dates back to around 2000 B.C. in Egypt, where non-standard hieroglyphics were used to conceal messages. Since then, many civilizations with written language have employed some form of secret writing. Notable historical examples include the scytale cipher used in ancient Sparta and the Caesar cipher developed in ancient Rome—both early methods of protecting sensitive information.

While **Cryptography** is the science of secret writing aimed at concealing the meaning of a message, **Cryptanalysis** is the science — and often the art — of breaking cryptographic systems and uncovering hidden information.



# Symmetric Cryptography

Symmetric cryptography, also known as secret key cryptography, refers to a method where both parties share the same secret key for encryption and decryption. It is best suited for bulk encryption due to its speed and efficiency compared to asymmetric cryptography.




Types of Symmetric Key Cryptography:

1. Stream Ciphers

2. Block Ciphers

Stream Ciphers:

The encryption process begins with the stream cipher's algorithm generating a pseudo-random keystream made up of the encryption key and the unique randomly generated number known as the nonce. The result is a random stream of bits corresponding to the length of the ordinary plaintext. Then, the ordinary plaintext is also deciphered into single bits.

Block Cipher:

The result of a block cipher is a sequence of blocks that are then encrypted with the key. The output is a sequence of blocks of encrypted data in a specific order. When the ciphertext travels to its endpoint, the receiver uses the same cryptographic key to decrypt the ciphertext blockchain to the plaintext message.

The most common block cipher algorithms are:

1.Advanced Encryption Standard (AES):

- It has support for three-length keys: 128 bits, 192 bits, or 256 bits, the most commonly used one is a 128-bit key.
- It includes secure communication, data encryption in storage devices, digital rights management (DRM), and so on.

2. Data Encryption Standard (DES):

- In DES, the 64-bit blocks of plaintext are encrypted using a 56-bit key.
- This weakness caused by the small key size led to the development of a more secure algorithm, called AES.

3. Triple Data Encryption Algorithm (Triple DES):

- The development of the Triple DES, also called Triple-DES or TDEA, was triggered by the weak security resulting from the small key size in the DES.

# Confusion and Diffusion

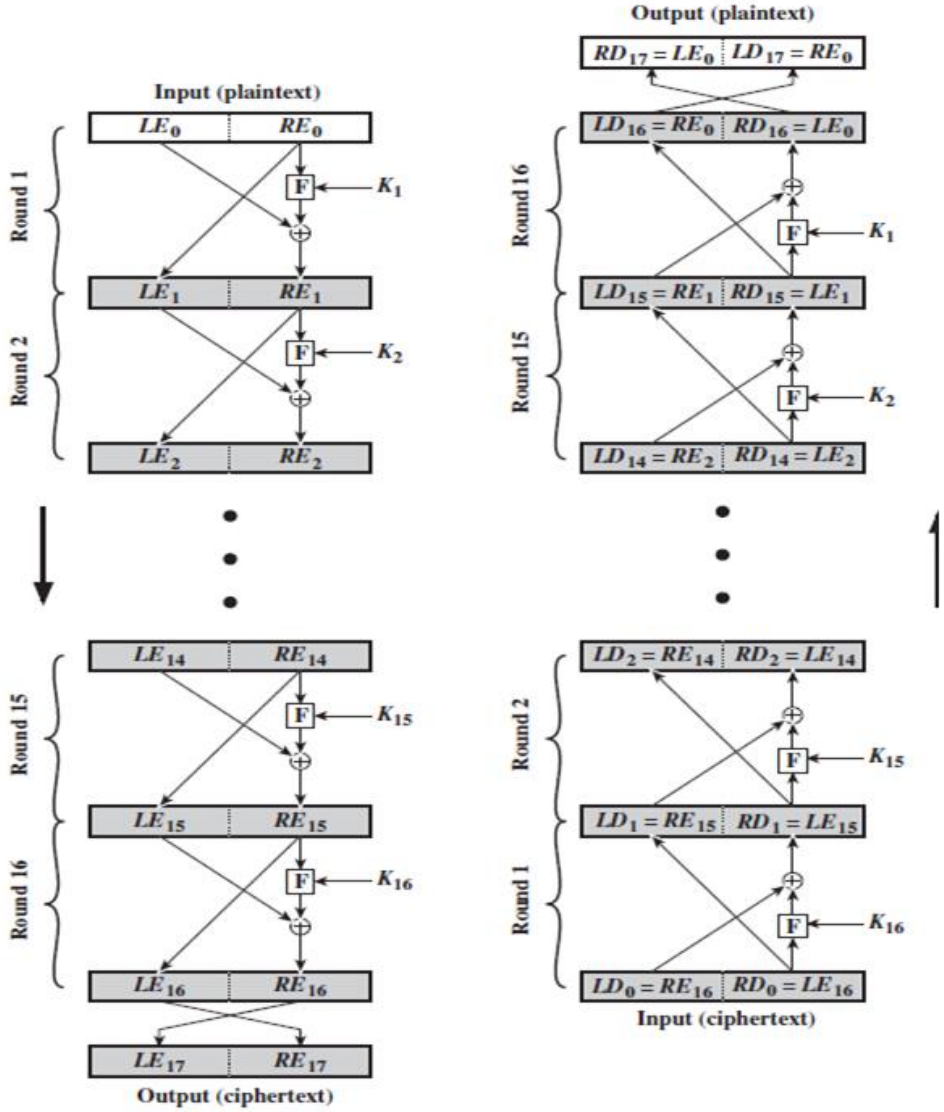
According to the famous information theorist Claude Shannon, there are two primitive operations with which strong encryption algorithms can be built:

1. **Confusion**: It is an encryption operation where the relationship between key and ciphertext is obscured. Today, a common element for achieving confusion is substitution, which is found in both DES and AES.

2. **Diffusion:** It is an encryption operation where the influence of one plaintext symbol is spread over many ciphertext symbols with the goal of hiding statistical properties of the plaintext. A simple diffusion element is the bit permutation, which is used frequently within DES. AES uses the more advanced Mix column operation.

## The Feistel Cipher structure:

Feistel Cipher model is a structure or a design used to develop many block ciphers such as DES. Feistel cipher may have invertible, non-invertible and self-invertible components in its design. Same encryption as well as decryption algorithm is used. A separate key is used for each round. However same round keys are used for encryption as well as decryption.



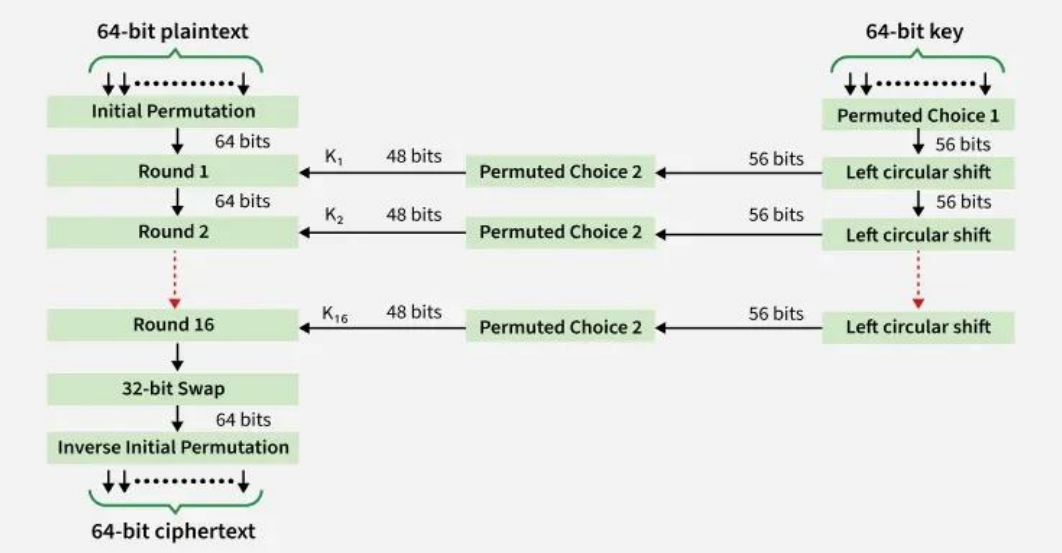
# Data Encryption Standard (DES)

Data Encryption Standard (DES) is a symmetric block cipher. By 'symmetric', we mean that the size of input text and output text (ciphertext) is same (64-bits). The 'block' here means that it takes group of bits together as input instead of encrypting the text bit by bit. It is a block cipher that encrypts data in 64-bit blocks.

- It takes a 64-bit plaintext input and generates a corresponding 64-bit ciphertext output.
- The main key length is 64-bit which is transformed into 56-bits by skipping every 8th bit in the key.
- It encrypts the text in 16 rounds where each round uses 48-bit subkey.
- This 48-bit subkey is generated from the 56-bit effective key.
- The same algorithm and key are used for both encryption and decryption with minor changes.

## Working of Data Encryption Standard (DES):

DES is based on the two attributes of Feistel cipher i.e. Substitution (also called confusion) and Transposition (also called diffusion). DES consists of 16 steps, each of which is called a round. Each round performs the steps of substitution and transposition along with other operations.




The DES (Data Encryption Standard) process involves several steps, organized into a series of rounds within a Feistel network structure. Here's a breakdown of the steps:

**1. Initial Permutation (IP):**

The 64-bit plaintext block undergoes an initial permutation, where the positions of its bits are rearranged according to a predefined table.

IP

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

**2. Key Generation:**

The 56-bit key is expanded and divided into sixteen 48-bit subkeys, one for each round of encryption.

- **Initial Key:** The process begins with a 64-bit key, which is the user-provided secret key used to initiate the encryption.
- **Key Permutation and Compression:** 8-bit is being dropped and the 56-bit key undergoes an **initial permutation (**PC1 permutation**)** and compression. During this step, certain bits are selected and rearranged according to a predefined table. The result is a 56-bit permutation of the original key.

**Permutation Table**

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 0 | 57 | 49 | 41 | 33 | 25 | 17 | 9 | 1 |
| 1 | 58 | 50 | 42 | 34 | 26 | 18 | 10 | 2 |
| 2 | 59 | 51 | 43 | 35 | 27 | 19 | 11 | 3 |
| 3 | 60 | 52 | 44 | 36 | 63 | 55 | 47 | 39 |
| 4 | 31 | 23 | 15 | 7 | 62 | 54 | 46 | 38 |
| 5 | 30 | 22 | 14 | 6 | 61 | 53 | 45 | 37 |
| 6 | 29 | 21 | 13 | 5 | 28 | 20 | 12 | 4 |

- **Key Splitting:** The 56-bit permuted key is then split into two **28-bit** halves: a left half **(C0)** and a right half **(D0)**.
- **Left Shift:** In each round, both the left and right halves of the key are independently rotated or shifted by a certain number of bits, creating a new combination of bits.
- Shift schedule = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]
- **Concatenation and Compression:** After the rotation, the left and right halves are concatenated and compressed to create a **48-bit** round subkey for that specific round.
- **Key Compression:** The compression PC2 permutation changes the 56-bit key to the 48-bit key, which is used as a key for the corresponding round. Example: for round 1 we'll use K1 and so on.

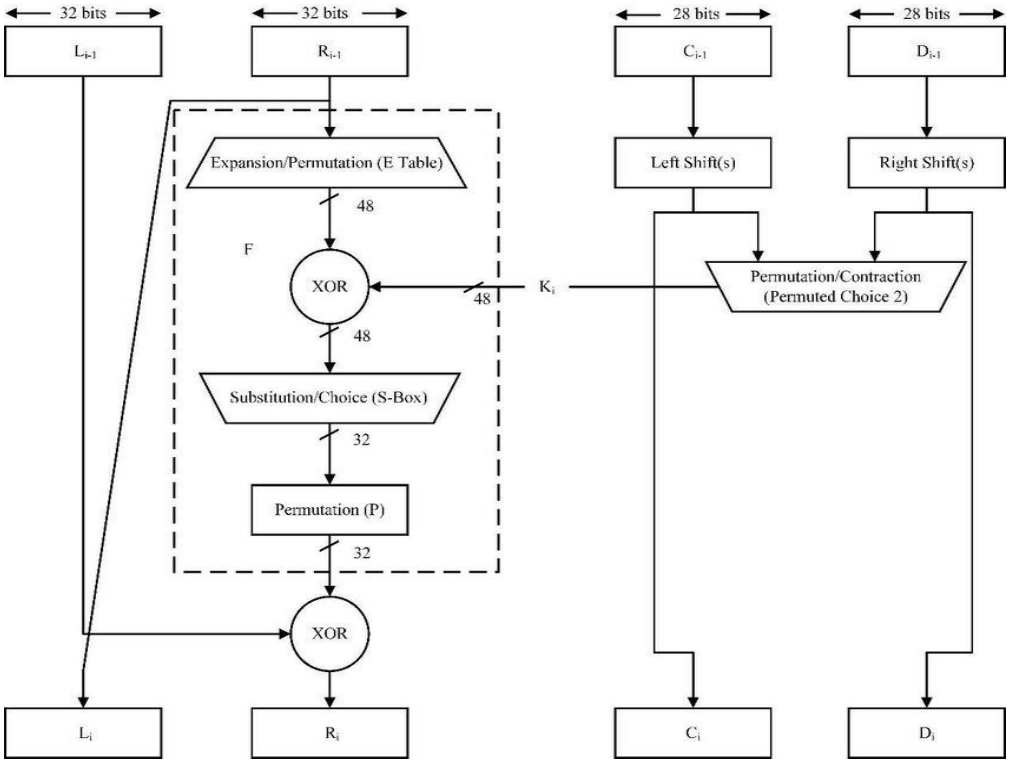
**Key Compression Table**

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| 1 | 14 | 17 | 11 | 24 | 01 | 05 | 03 | 28 |
| 2 | 15 | 06 | 21 | 10 | 23 | 19 | 12 | 04 |
| 3 | 26 | 08 | 16 | 07 | 27 | 20 | 13 | 02 |
| 4 | 41 | 52 | 31 | 37 | 47 | 55 | 30 | 40 |
| 5 | 51 | 45 | 33 | 48 | 44 | 49 | 39 | 56 |
| 6 | 34 | 53 | 46 | 42 | 50 | 36 | 29 | 32 |

- **Round-Specific Subkey Generation:** The key expansion process involves creating **sixteen round-specific subkeys**, one for each round of encryption for C1 and D1 we'll use C0 and D0 and for C2 and D2 we'll use C1 and D1, and so on till C15 and D15.
- **Sixteen Subkeys:** This process is repeated for each of the sixteen rounds, resulting in a set of sixteen **48-bit subkeys (K1, K2, …, K16)**.

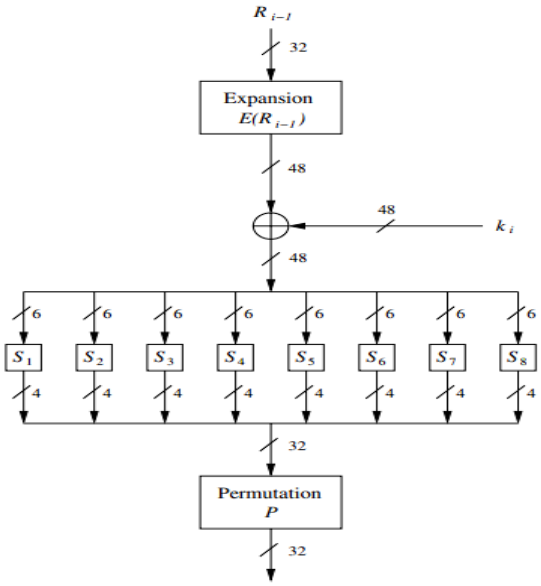
**3. Rounds of Encryption:**

- The data block is divided into two halves (L and R), and each round involves the transformation of one half based on the other half and the round key.




**Single Round of Encryption:**

- **The f-Function:**
  the f-function plays a crucial role for the security of DES. In round 1 it takes the right half Ri−1 of the output of the previous round and the current round key ki as input. The output of the f-function is used as an XOR-mask for encrypting the left half input bits Li−1.




- **Expansion/Permutation**: The right half (R0) is subjected to an **expansion operation**, **increasing its size (32-bit to 48-bit)**. This expanded right half is then XORed (bitwise exclusive OR) with the round-specific subkey (Ki) derived from the key schedule.

**E**

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

- **Substitution (S-boxes):** The result of the XOR operation is divided into **eight 6-bit blocks**. Each block is fed into a corresponding S-box (substitution box), which replaces it with a 4-bit output based on the S-box's predefined substitution table. The outputs from all S-boxes are concatenated to form a **32-bit output**.
- **Permutation (P-Box):** The 32-bit output from the S-boxes is subjected to a fixed permutation (P-box permutation). This rearranges the bits according to a predetermined permutation table.

**P**

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

- **XOR with Left Half:** The output of the permutation step is XORed with the original left half (L0).
- **Swap Left and Right Halves:** The result of the XOR operation becomes the new right half (R1), and the original right half (R0) becomes the new left half (L1) for round 2 we'll repeat these steps with respective key on L1 and R1 and so on.
- The transformation is accomplished through a combination of substitution (using S-boxes), permutation, and bitwise operations.
- The use of a different subkey for each round adds variability to the encryption process, enhancing security.

**5. Final Permutation (FP):** After the 16th round, the left and right halves are swapped one last time. The final result undergoes a **final permutation** (inverse of the initial permutation) to generate the **64-bit block of ciphertext.**

**$IP^{-1}$**

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

# Implementation of DES using Python

## Code for Encryption:

```
# Hexadecimal to binary conversion
def hex2bin(s):
    mp = {'0': "0000", '1': "0001", '2': "0010", '3': "0011",
          '4': "0100", '5': "0101", '6': "0110", '7': "0111",
          '8': "1000", '9': "1001", 'A': "1010", 'B': "1011",
          'C': "1100", 'D': "1101", 'E': "1110", 'F': "1111"}
    bin_str = ""
    for i in range(len(s)):
        bin_str += mp[s[i]]
    return bin_str

# Binary to hexadecimal conversion
def bin2hex(s):
    mp = {"0000": '0', "0001": '1', "0010": '2', "0011": '3',
          "0100": '4', "0101": '5', "0110": '6', "0111": '7',
          "1000": '8', "1001": '9', "1010": 'A', "1011": 'B',
          "1100": 'C', "1101": 'D', "1110": 'E', "1111": 'F'}
    hex_str = ""
    for i in range(0, len(s), 4):
        ch = s[i:i+4]
        hex_str += mp[ch]
    return hex_str

# Binary to decimal conversion
def bin2dec(binary):
    decimal, i = 0, 0
    while binary != 0:
        dec = binary % 10
        decimal += dec * pow(2, i)
        binary //= 10
        i += 1
    return decimal

# Decimal to binary conversion
def dec2bin(num):
    res = bin(num).replace("0b", "")
    if len(res) % 4 != 0:
        div = len(res) // 4
        counter = (4 * (div + 1)) - len(res)
        res = '0' * counter + res
    return res

# Permute function to rearrange the bits
def permute(k, arr, n):
    permutation = ""
    for i in range(0, n):
        permutation += k[arr[i] - 1]
    return permutation
```

```
# Shifting the bits towards left by nth shifts
def shift_left(k, nth_shifts):
    s = ""
    for i in range(nth_shifts):
        for j in range(1, len(k)):
            s += k[j]
        s += k[0]
        k = s
        s = ""
    return k

# Calculating XOR of two binary strings
def xor(a, b):
    ans = ""
    for i in range(len(a)):
        ans += "0" if a[i] == b[i] else "1"
    return ans

# Convert string to hexadecimal
def string_to_hex(s):
    hex_str = ''
    for char in s:
        hex_str += format(ord(char), '02x').upper()
    return hex_str

# Convert hexadecimal to string
def hex_to_string(hex_str):
    try:
        if len(hex_str) % 2 != 0:
            hex_str += '0'
        bytes_obj = bytes.fromhex(hex_str)
        return bytes_obj.decode('ascii', errors='ignore')
    except ValueError:
        return "Invalid hex string"

# Table of Position of 64 bits at initial level: Initial Permutation Table
initial_perm = [58, 50, 42, 34, 26, 18, 10, 2,
                60, 52, 44, 36, 28, 20, 12, 4,
                62, 54, 46, 38, 30, 22, 14, 6,
                64, 56, 48, 40, 32, 24, 16, 8,
                57, 49, 41, 33, 25, 17, 9, 1,
                59, 51, 43, 35, 27, 19, 11, 3,
                61, 53, 45, 37, 29, 21, 13, 5,
                63, 55, 47, 39, 31, 23, 15, 7]

# Expansion D-box Table
exp_d = [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]
```

```
# Straight Permutation Table
per = [16, 7, 20, 21,
       29, 12, 28, 17,
       1, 15, 23, 26,
       5, 18, 31, 10,
       2, 8, 24, 14,
       32, 27, 3, 9,
       19, 13, 30, 6,
       22, 11, 4, 25]

# S-box Table
sbox = [[[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]],
        [[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]],
        [[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]],
        [[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]],
        [[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]],
        [[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]],
        [[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]],
        [[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]]]

# Final Permutation Table
final_perm = [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]
```

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

# Number of bit shifts
shift_table = [1, 1, 2, 2,
               2, 2, 2, 2,
               1, 2, 2, 2,
               2, 2, 2, 1]

# Key compression table
key_comp = [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]

def encrypt(pt, rkb, rk):
    pt = hex2bin(pt)
    pt = permute(pt, initial_perm, 64)
    print("After initial permutation", bin2hex(pt))

    left = pt[0:32]
    right = pt[32:64]
    for i in range(0, 16):
        right_expanded = permute(right, exp_d, 48)
        xor_x = xor(right_expanded, rkb[i])

        sbox_str = ""
        for j in range(0, 8):
            row = bin2dec(int(xor_x[j * 6] + xor_x[j * 6 + 5]))
            col = bin2dec(int(xor_x[j * 6 + 1] + xor_x[j * 6 + 2] + xor_x[j * 6 + 3] + xor_x[j * 6 + 4]))
            val = sbox[j][row][col]
            sbox_str += dec2bin(val)

        sbox_str = permute(sbox_str, per, 32)
        result = xor(left, sbox_str)
        left = result

        if i != 15:
            left, right = right, left
        print(f"Round {i + 1} {bin2hex(left)} {bin2hex(right)} {rk[i]}")

    combine = left + right
    cipher_text = permute(combine, final_perm, 64)
    return cipher_text
```

```python
def main():
    pt = input("Enter plaintext (up to 8 characters): ")
    key = input("Enter key (up to 8 characters): ")

    pt = pt[:8].ljust(8, '\0')
    key = key[:8].ljust(8, '\0')

    pt_hex = string_to_hex(pt)
    key_hex = string_to_hex(key)

    print(f"Plaintext in hex: {pt_hex}")
    print(f"Key in hex: {key_hex}")

    key_bin = hex2bin(key_hex)
    key_bin = permute(key_bin, keyp, 56)

    left = key_bin[0:28]
    right = key_bin[28:56]

    rkb = []
    rk = []
    for i in range(0, 16):
        left = shift_left(left, shift_table[i])
        right = shift_left(right, shift_table[i])
        combine_str = left + right
        round_key = permute(combine_str, key_comp, 48)
        rkb.append(round_key)
        rk.append(bin2hex(round_key))

    print("\nEncryption")
    cipher_text = bin2hex(encrypt(pt_hex, rkb, rk))
    print("Cipher Text (hex): ", cipher_text)
    print("Cipher Text (ASCII, may not be readable): ", hex_to_string(cipher_text))

if __name__ == "__main__":
    main()
```

# OUTPUT:

```
Enter plaintext (up to 8 characters): HELLO_W!
Enter key (up to 8 characters): 1234@BCD
Plaintext in hex: 48454C4C4F5F5721
Key in hex: 3132333440424344

Encryption
After initial permutation 7F607EF200803D70
Round 1 00803D70 4E250033 2038444720C6
Round 2 4E250033 AFE7ED81 00345440A343
Round 3 AFE7ED81 5CCBEB73 44445476A408
Round 4 5CCBEB73 1300AA6D 46C12068154A
Round 5 1300AA6D 6F2E59E3 8A81230CF02A
Round 6 6F2E59E3 2124333D A9020B645C60
Round 7 2124333D C75E67C2 21128888887A
Round 8 C75E67C2 F17F0381 1018D085DE10
Round 9 F17F0381 00C13D8A 104850891A08
Round 10 00C13D8A 314C5EBC 046914D05234
Round 11 314C5EBC 72ABC1D6 062505110AAC
Round 12 72ABC1D6 221A0E54 4B0421903891
Round 13 221A0E54 7C8E3EC5 C980A8232235
Round 14 7C8E3EC5 C61479F8 90828A332982
Round 15 C61479F8 C8E7B189 301A02240117
Round 16 377413C7 C8E7B189 303A00B601C0
Cipher Text (hex):  6F6571825C78B1AB
Cipher Text (ASCII, may not be readable):  oeq\x
```

# Code for Decryption

```python
# Hexadecimal to binary conversion
def hex2bin(s):
    mp = {'0': "0000", '1': "0001", '2': "0010", '3': "0011",
          '4': "0100", '5': "0101", '6': "0110", '7': "0111",
          '8': "1000", '9': "1001", 'A': "1010", 'B': "1011",
          'C': "1100", 'D': "1101", 'E': "1110", 'F': "1111"}
    bin_str = ""
    for i in range(len(s)):
        bin_str += mp[s[i]]
    return bin_str


# Binary to hexadecimal conversion
def bin2hex(s):
    mp = {"0000": '0', "0001": '1', "0010": '2', "0011": '3',
          "0100": '4', "0101": '5', "0110": '6', "0111": '7',
          "1000": '8', "1001": '9', "1010": 'A', "1011": 'B',
          "1100": 'C', "1101": 'D', "1110": 'E', "1111": 'F'}
    hex_str = ""
    for i in range(0, len(s), 4):
        ch = s[i:i + 4]
        hex_str += mp[ch]
    return hex_str


# Binary to decimal conversion
def bin2dec(binary):
    decimal, i = 0, 0
    while binary != 0:
        dec = binary % 10
        decimal += dec * pow(2, i)
        binary //= 10
        i += 1
    return decimal


# Decimal to binary conversion
def dec2bin(num):
    res = bin(num).replace("0b", "")
    if len(res) % 4 != 0:
        div = len(res) // 4
        counter = (4 * (div + 1)) - len(res)
        res = '0' * counter + res
    return res
```

```python
# Permute function to rearrange the bits
def permute(k, arr, n):
    permutation = ""
    for i in range(0, n):
        permutation += k[arr[i] - 1]
    return permutation


# Shifting the bits towards left by nth shifts
def shift_left(k, nth_shifts):
    s = ""
    for i in range(nth_shifts):
        for j in range(1, len(k)):
            s += k[j]
        s += k[0]
        k = s
        s = ""
    return k


# Calculating XOR of two binary strings
def xor(a, b):
    ans = ""
    for i in range(len(a)):
        ans += "0" if a[i] == b[i] else "1"
    return ans


# Convert ASCII string to hexadecimal
def string_to_hex(s):
    hex_str = ''
    for char in s:
        hex_str += format(ord(char), '02x').upper()
    return hex_str


# Convert hexadecimal to ASCII string
def hex_to_string(hex_str):
    try:
        if len(hex_str) % 2 != 0:
            hex_str += '0'
        bytes_obj = bytes.fromhex(hex_str)
        return bytes_obj.decode('ascii', errors='ignore')
    except ValueError:
        return "Invalid hex string"
```

```
def hex_to_string(hex_str):
    except ValueError:
        return "Invalid hex string"
# DES Tables
initial_perm = [58, 50, 42, 34, 26, 18, 10, 2,
                60, 52, 44, 36, 28, 20, 12, 4,
                62, 54, 46, 38, 30, 22, 14, 6,
                64, 56, 48, 40, 32, 24, 16, 8,
                57, 49, 41, 33, 25, 17, 9, 1,
                59, 51, 43, 35, 27, 19, 11, 3,
                61, 53, 45, 37, 29, 21, 13, 5,
                63, 55, 47, 39, 31, 23, 15, 7]

exp_d = [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]

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

sbox = [[[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]],
        [[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]],
        [[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]],
        [[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]],
        [[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]],
        [[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]],
        [[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]],
        [[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]]]
```

```
final_perm = [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]

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

shift_table = [1, 1, 2, 2,
               2, 2, 2, 2,
               1, 2, 2, 2,
               2, 2, 2, 1]

key_comp = [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]

def decrypt(cipher_text, rkb, rk):
    ct = hex2bin(cipher_text)
    ct = permute(ct, initial_perm, 64)
    print("After initial permutation", bin2hex(ct))

    left = ct[0:32]
    right = ct[32:64]
    for i in range(0, 16):
        right_expanded = permute(right, exp_d, 48)
        xor_x = xor(right_expanded, rkb[i])

        sbox_str = ""
        for j in range(0, 8):
            row = bin2dec(int(xor_x[j * 6] + xor_x[j * 6 + 5]))
            col = bin2dec(int(xor_x[j * 6 + 1] + xor_x[j * 6 + 2] +
                              xor_x[j * 6 + 3] + xor_x[j * 6 + 4]))
            val = sbox[j][row][col]
            sbox_str += dec2bin(val)

        sbox_str = permute(sbox_str, per, 32)
        result = xor(left, sbox_str)
        left = result
        if i != 15:
            left, right = right, left
        print(f"Round {i + 1} {bin2hex(left)} {bin2hex(right)} {rk[i]}")

    combine = left + right
    plain_text = permute(combine, final_perm, 64)
    return plain_text


def main():
    # Input ciphertext as hex, key as ASCII
    ct_hex = input("Enter ciphertext (16 hex characters): ").upper()
    key = input("Enter key (up to 8 characters): ")

    # Validate hex input
    if len(ct_hex) != 16 or not all(c in '0123456789ABCDEF' for c in ct_hex):
        print("Error: Ciphertext must be 16 hexadecimal characters")
        return

    # Pad or truncate key to 8 characters
    key = key[:8].ljust(8, '\0')

    # Convert key to hexadecimal
    key_hex = string_to_hex(key)

    print(f"Ciphertext in hex: {ct_hex}")
    print(f"Key in hex: {key_hex}")

    # Key generation
    key_bin = hex2bin(key_hex)
    key_bin = permute(key_bin, keyp, 56)

    left = key_bin[0:28]
    right = key_bin[28:56]
```

```
    rkb = []
    rk = []
    for i in range(0, 16):
        left = shift_left(left, shift_table[i])
        right = shift_left(right, shift_table[i])
        combine_str = left + right
        round_key = permute(combine_str, key_comp, 48)
        rkb.append(round_key)
        rk.append(bin2hex(round_key))

    # Reverse round keys for decryption
    rkb_rev = rkb[::-1]
    rk_rev = rk[::-1]

    print("\nDecryption")
    plain_text_hex = bin2hex(decrypt(ct_hex, rkb_rev, rk_rev))
    print("Plain Text (hex): ", plain_text_hex)
    print("Plain Text (ASCII): ", hex_to_string(plain_text_hex))


if __name__ == "__main__":
    main()
```

# OUTPUT:

```
Enter ciphertext (16 hex characters): 6F6571825C78B1AB
Enter key (up to 8 characters): 1234@BCD
Ciphertext in hex: 6F6571825C78B1AB
Key in hex: 3132333440424344

Decryption
After initial permutation 377413C7C8E7B189
Round 1 C8E7B189 C61479F8 303A00B601C0
Round 2 C61479F8 7C8E3EC5 301A02240117
Round 3 7C8E3EC5 221A0E54 90828A332982
Round 4 221A0E54 72ABC1D6 C980A8232235
Round 5 72ABC1D6 314C5EBC 4B0421903891
Round 6 314C5EBC 00C13D8A 062505110AAC
Round 7 00C13D8A F17F0381 046914D05234
Round 8 F17F0381 C75E67C2 104850891A08
Round 9 C75E67C2 2124333D 1018D085DE10
Round 10 2124333D 6F2E59E3 21128888887A
Round 11 6F2E59E3 1300AA6D A9020B645C60
Round 12 1300AA6D 5CCBEB73 8A81230CF02A
Round 13 5CCBEB73 AFE7ED81 46C12068154A
Round 14 AFE7ED81 4E250033 44445476A408
Round 15 4E250033 00803D70 00345440A343
Round 16 7F607EF2 00803D70 2038444720C6
Plain Text (hex):  48454C4C4F5F5721
Plain Text (ASCII):  HELLO_W!
```

# Asymmetric Cryptography

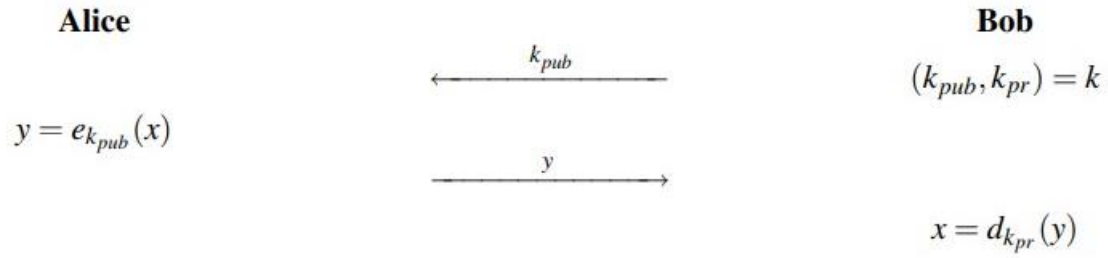
## Definition

**Asymmetric cryptography**, also known as **public-key cryptography**, uses a **pair of keys** — a **public key** and a **private key** — to encrypt and decrypt data. The public key is shared openly and used for encryption, while the private key is kept secret and used for decryption. This eliminates the need to share a single secret key and significantly enhances security, especially for digital communications over open networks. Asymmetric cryptography is widely used in secure email, digital signatures, and online transactions. Common algorithms include **RSA (Rivest–Shamir–Adleman)** and **Elliptic Curve Cryptography (ECC)**.

It is widely used in:

- Secure communication
- Digital signatures
- SSL/TLS for secure websites
- End-to-end encryption in messaging apps
- Verifying documents and software updates

Here's An Example of Alice and Bob using Asymmetric Cryptography




**Bob generates a key pair:**

- Public key (kpub) – shared with everyone
- Private key (kpr) – kept secret

**Alice wants to send a secret message to Bob:**

- She encrypts the message using Bob's public key (kpub)

**Bob receives the encrypted message:**

- He decrypts it using his private key (kpr)

# RSA

In 1977, Ronald Rivest, Adi Shamir and Leonard Adleman proposed a scheme which became the most widely used asymmetric cryptographic scheme called RSA. **RSA** is a **public-key cryptosystem** that is used for **secure data transmission**, especially in applications like secure emails, digital signatures, and online transactions. It is based on the **mathematical difficulty of factoring large prime numbers** — a problem that is easy to verify but extremely hard to reverse.

RSA uses **two keys**:

- A **public key** for encryption (shared openly)
- A **private key** for decryption (kept secret)

Anyone can use the public key to encrypt data, but only the holder of the private key can decrypt it. Before the application of RSA, some prerequisite knowledge on Fermat's(little) Theorem and Euler's Theorem are required.

## Fermat's (little) Theorem

Fermat's Little Theorem is a fundamental result in number theory. It states:

If p is a prime number and a is an integer such that a is not divisible by p, then:

*Let a be an integer and p be a prime, then:*

$$a^p \equiv a \pmod{p}.$$

Fermat's Little Theorem is fundamental to public-key cryptography, particularly:

- RSA algorithm (used in modular arithmetic and modular inverses)
- Fast modular exponentiation
- Finding large prime numbers
- Modular inverse calculation (e.g., computing private keys)

## Modulo Multiplicative Inverse

The **modulo multiplicative inverse** of an integer **a modulo n** is an integer **x** such that:

$$a \times x \equiv 1 \pmod{n}$$

# Euler's Theorem

**Euler's Theorem** is a generalization of **Fermat's Little Theorem**. It states:

$$\textit{Let } a \textit{ and } m \textit{ be integers with } \gcd(a,m) = 1, \textit{ then:}$$

$$a^{\Phi(m)} \equiv 1 \pmod{m}.$$

Where:

- ϕ(n) is Euler's totient function, which counts the number of integers less than n that are coprime to n.

It is used in **RSA** to compute the **modular inverse** of the public exponent **e**, which helps generate the **private key d**:

$$d \equiv e^{-1} \mod \phi(n)$$

# Relationship with Fermat's Theorem

Fermat's Little Theorem is a **special case** of Euler's Theorem when **n** is a **prime number**:

$$a^{p-1} \equiv 1 \mod p \quad \text{(Fermat)}$$

$$a^{\phi(n)} \equiv 1 \mod n \quad \text{(Euler)}$$

# ALGORITHM

## 1. Key Generation

The process of creating a matching pair of public and private keys for secure communication.

1. Choose two large primes $p$ and $q$.
2. Compute $n = p \cdot q$.
3. Compute $\Phi(n) = (p-1)(q-1)$.
4. Select the public exponent $e \in \{1, 2, \ldots, \Phi(n)-1\}$ such that

$$\gcd(e, \Phi(n)) = 1.$$

5. Compute the private key $d$ such that

$$d \cdot e \equiv \bmod \Phi(n)$$

🔒 **Public Key: (e, n) – shared with everyone**

🔒 **Private Key: (d, n) – kept secret**

## 2. Encryption

The process of converting a plaintext message into an unreadable ciphertext using the recipient's public key.

1. Convert the plaintext message $m$ into an integer such that:

$$0 < m < n$$

2. Use the public key $(e, n)$ to encrypt:

$$c = m^e \mod n$$

Where:

c is the **ciphertext**

# 3. Decryption

The process of converting the ciphertext back into the original plaintext using the private key.

$$m = c^d \mod n$$

Where:

c = the encrypted message (ciphertext)

d = private exponent

n = modulus

m = original plaintext message

# Implementation of RSA using Python

## 1. Key Generation

```python
import random, math, json

def str2ascii(st):
    return [ord(c) for c in st]

def is_prime(n):
    if n <= 1: return False
    if n <= 3: return True
    if n % 2 == 0 or n % 3 == 0: return False
    i = 5
    while i * i <= n:
        if n % i == 0 or n % (i + 2) == 0: return False
        i += 6
    return True

def generate_prime(bits):
    while True:
        candidate = random.getrandbits(bits)
        candidate |= (1 << bits - 1) | 1  # Ensure it's of proper length and odd
        if is_prime(candidate): return candidate

def generate_keys(bits):
    p = generate_prime(bits)
    q = generate_prime(bits)
    while p == q:
        q = generate_prime(bits)
    n = p * q
    phi = (p - 1) * (q - 1)
    e = random.randrange(3, phi, 2)
    while math.gcd(e, phi) != 1:
        e = random.randrange(3, phi, 2)
    d = pow(e, -1, phi)
    return p, q, e, d, n
```

# 2. Encryption

```
def encrypt(ascii_list, e, n):
    return [pow(m, e, n) for m in ascii_list]

# --- MAIN LOGIC ---
bit_length = int(input("🔢 Enter bit length for primes (e.g., 16, 32, 64): "))
message = input("✉️  Enter message to encrypt: ")
ascii_msg = str2ascii(message)

p, q, e, d, n = generate_keys(bit_length)
cipher = encrypt(ascii_msg, e, n)

# Save to file
with open("rsa_encrypted_data.json", "w") as f:
    json.dump({
        "ciphertext": cipher,
        "public_key": {"e": e, "n": n},
        "private_key": {"d": d, "n": n},
        "primes": {"p": p, "q": q},
        "original_ascii": ascii_msg
    }, f, indent=2)

# --- Display Info ---
print("\n🔑 Keys Generated:")
print(f"  p = {p}")
print(f"  q = {q}")
print(f"  n = {n}")
print(f"  e (public exponent) = {e}")
print(f"  d (private exponent) = {d}")

print(f"\n🔐 Public Key: (e={e}, n={n})")
print(f"🔓 Private Key: (d={d}, n={n})")

print(f"\n✉️  Original Message: '{message}'")
print(f"📄 ASCII Encoding: {ascii_msg}")
print(f"🔒 Encrypted Ciphertext: {cipher}")
print(f"\n✅ Data saved to 'rsa_encrypted_data.json'")
```

## Output

🔢 Enter bit length for primes (e.g., 16, 32, 64): 18

✉️ Enter message to encrypt: NEWPROJECT

🔑 Keys Generated:

p = 164299

q = 235177

n = 38639345923

e (public exponent) = 25962231703

d (private exponent) = 34470784807

🔐 Public Key: (e=25962231703, n=38639345923)

🔏 Private Key: (d=34470784807, n=38639345923)

📨 Original Message: 'NEWPROJECT'

🔠 ASCII Encoding: [78, 69, 87, 80, 82, 79, 74, 69, 67, 84]

🔒 Encrypted Ciphertext: [28992306408, 13413407240, 17344499723,
4526239047, 35332784591, 15496763047, 18893364312,
13413407240, 33727538030, 19625484893]

✅ Data saved to 'rsa_encrypted_data.json'

## 3. Decryption

```
def ascii2str(ascii_list):
    """Converts a list of ASCII values back into a string."""
    return ''.join(chr(code) for code in ascii_list)

def endecrypt_message(m, key, n):
    """Performs RSA modular exponentiation (used for both encryption/decryption)."""
    result = 1
    m = m % n
    if m == 0:
        return 0
    while key > 0:
        if key % 2 == 1:
            result = (result * m) % n
        key = key >> 1
        m = (m * m) % n
    return result

# --- INPUT SECTION ---
# Input encrypted message
encrypted_str = input("Enter encrypted message (comma-separated integers): ")
encrypted = [int(x.strip()) for x in encrypted_str.split(",")]

# Input private key
d = int(input("Enter private key exponent d: "))
n = int(input("Enter modulus n: "))

# --- DECRYPTION ---
decrypted_ascii = [endecrypt_message(char, d, n) for char in encrypted]
decrypted_message = ascii2str(decrypted_ascii)

# --- OUTPUT ---
print("\n✅ Decryption Successful!")
print(f"Decrypted ASCII: {decrypted_ascii}")
print(f"Original Message: {decrypted_message}")
```

## OUTPUT

Enter encrypted message (comma-separated integers): 28992306408, 13413407240, 17344499723, 4526239047, 35332784591, 15496763047, 18893364312, 13413407240, 33727538030, 19625484893

Enter private key exponent d: 34470784807

Enter modulus n: 38639345923

✅ Decryption Successful!

Decrypted ASCII: [78, 69, 87, 80, 82, 79, 74, 69, 67, 84]

Original Message: NEWPROJECT

# Comparison between Symmetric and Asymmetric Cryptography

Symmetric cryptography uses the **same secret key** for both encryption and decryption, making it very **fast and efficient** for large data. However, sharing the key securely is a challenge. Examples include **AES**, **DES**, and **Blowfish**.
Asymmetric cryptography uses a **public key** to encrypt and a **private key** to decrypt, so the private key never needs to be shared. This provides **better security** for key exchange and is often used for **digital signatures** and **SSL/TLS**, though it's **slower** and best for small data. Examples include **RSA**, **ECC**, and **DSA**.

Both symmetric and asymmetric cryptography yielded distinct types of outputs that highlight their different purposes and efficiencies in securing Range EOTS data. In the DES example, the output after multiple rounds of substitutions and permutations was a 64-bit ciphertext (6F6571825C78B1AB), which was then successfully decrypted back into the plaintext (HELLO_W!). The intermediate rounds showed how data was split into left and right halves and progressively mixed using the same secret key, demonstrating that symmetric encryption is highly effective and fast at producing encrypted blocks and recovering the original message with minimal computational overhead. In contrast, the RSA example showed the use of large numbers as outputs — for instance, the encrypted message was a list of very large integers ([28992306408, 13413407240, ...]), and the private exponent d was much larger than the DES key, indicating the complexity of public-key math. Decryption successfully recovered the ASCII representation of the message (NEWPROJECT), but required substantial computation and mathematical operations like modular exponentiation. These outputs clearly illustrate that while symmetric cryptography produces compact, block wise encrypted data efficiently and quickly, asymmetric cryptography generates large numeric outputs and is slower, making it more practical for securely exchanging keys or small control data. Together, they offer a balanced approach in a Range EOTS environment, where DES can rapidly protect continuous telemetry or video feeds once a secret key is securely established via RSA.

# Conclusion

Cryptography plays a **critical role** in safeguarding sensitive data in **Range EOTS** at test facilities like ITR, DRDO. Both **symmetric** and **asymmetric** encryption techniques contribute to protecting communications, telemetry, video feeds, and control signals. **Symmetric cryptography** enables **fast and efficient encryption** of large volumes of real-time data—ensuring that video streams, sensor readings, and test-range data remain confidential and intact. Meanwhile, **asymmetric cryptography** provides a robust mechanism for **secure key exchange, authentication, and access control**, ensuring that only authorized personnel can decrypt data or send valid commands to range instruments. Together, these methods guarantee the **integrity, confidentiality, and authenticity** of all test-range information, strengthening the security and operational reliability of Range EOTS systems in high-stakes defence environments.

In modern systems, **both methods are often used together in a hybrid approach**. Asymmetric encryption is used to exchange a symmetric session key, and then symmetric encryption takes over to handle the data transfer efficiently. This combination leverages the **strengths of both techniques**, ensuring both **security and performance**.

As technology advances, especially with the rise of quantum computing and interconnected devices, cryptography will continue to evolve — but its core purpose remains the same: to protect information in a world that increasingly depends on it.

## Reference


- Understanding Cryptography (Christof Paar, Jan Pelzl)
- Cryptography and Network Security by William Stallings
- Handbook of Applied Cryptography (Alfred Menezes, Paul van Oorschot, and Scott Vanstone)
- GeeksForGeeks