Submitted By ,

- Bhagwati Kaiwartya
- Prerna Sharma
- Snehal Patra
- Wilson

prerna sharma

Table of Content

---------------------------------------*The End*-----------------------------------

Project Report

Topic: Advanced Encryption Standard
Subject: Software Engineering
Subject code: CS-118

1. Abstract

Advanced Encryption Standard (AES) is the current standard for secret key encryption. AES was created by two Belgian cryptographers, Vincent Rijmen and Joan Daemen, replacing the old Data Encryption Standard (DES). The Federal Information Processing Standard 197 used a standardized version of the algorithm called Rijndael algorithm for the Advanced Encryption Standard. The algorithm uses a combination of Exclusive-OR operations (XOR), octet substitution with an S-box, row and column rotations, and a 'Mix Column' function. It was successful because it was easy to implement and could run in a reasonable amount of time on a regular computer.

The previous standard, DES, was no longer adequate for security. It had been the standard since November 23, 1976. Computing power had increased a lot since then and the algorithm was no longer considered safe. In 1998 DES was cracked in less than three days by a specially made computer called the DES cracker.

On November 26, 2001 the Federal Information Processing Standards Publication 197 announced a standardized form of the Rijndael algorithm as the new standard for encryption. This standard was called Advanced Encryption Standard and is currently still the standard for encryption.

Hence our aim is to develop a software that implements the Advanced Encryption Standard (AES) on a text and delivers it securely to the receiver.

Such Encryption technique helps to avoid intrusion attacks.

2. Introduction

The more popular and widely adopted symmetric encryption algorithm likely to be encountered nowadays is the Advanced Encryption Standard (AES). It is found to be at least six time faster than the triple DES.

The features of AES are as follows:

- Symmetric key symmetric block cipher
- 128-bit data, 128/192/256-bit keys
- Stronger and faster than Triple-DES
- Provide full specification and design details

- Software implementable in C and Java

3. Working of AES

AES is an iterative cipher. It is based on 'substitution–permutation network'. It comprises of a series of linked operations, some of which involve replacing inputs by specific outputs (substitutions) and others involve shuffling bits around (permutations).

Interestingly, AES performs all its computations on bytes rather than bits. Hence, AES treats the 128 bits of a plaintext block as 16 bytes. These 16 bytes are arranged in four columns and four rows for processing as a matrix −

Unlike DES, the number of rounds in AES is variable and depends on the length of the key. AES uses 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys. Each of these rounds uses a different 128-bit round key, which is calculated from the original AES key.

There is a relationship between the key size and the number of rounds used in the algorithm. The following table depicts the number of rounds needed for the specific key size:

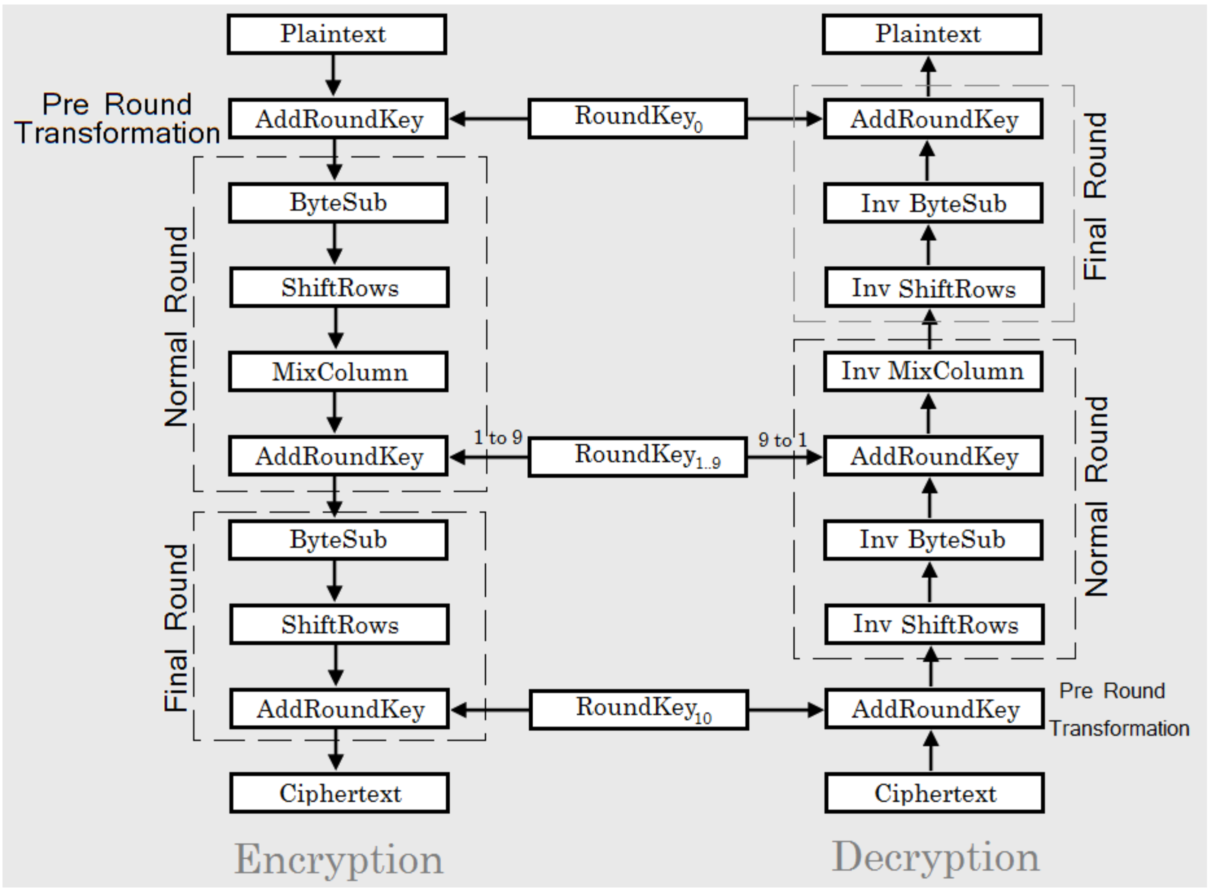
Number of Rounds (N_R)	Key Size
10	128
12	192
14	256

Table 1: Number of Rounds and key size

AES includes three block ciphers: AES-128, AES-192 and AES-256. AES-128 uses a 128-bit key length to encrypt and decrypt a block of messages, while AES-192 uses a 192-bit key length and AES-256 a 256-bit key length to encrypt and decrypt messages. Each cipher encrypts and decrypts data in blocks of 128 bits using cryptographic keys of 128, 192 and 256 bits, respectively. Symmetric, also known as *secret key*, ciphers use the same key for encrypting and decrypting, so the sender and the receiver must both know and use the same secret key. The government classifies information in three categories: Confidential, Secret or Top Secret. All key lengths can be used to protect the Confidential and Secret level. Top Secret information requires either 192- or 256-bit key lengths. There are 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys as shown in the table above. A round consists of several processing steps that include substitution, transposition and mixing of the input plaintext to transform it into the final output of ciphertext.

The AES encryption algorithm defines numerous transformations that are to be performed on data stored in an array. The first step of the cipher is to put the data into an array after which, the cipher transformations are repeated over multiple encryption rounds. The first transformation in the AES encryption cipher is substitution of data using a substitution table; the second transformation shifts data rows, and the third mixes columns. The last transformation is performed on each column using a different part of the encryption key. Longer keys need more rounds to complete.

The following diagram illustrates the entire procedure of AES:



The encryption and the decryption processes together ensure the security of the data being transferred. The decryption process is a reverse operation of the encryption process. We can see in the above diagram that in the encryption process where we have to substitute the bytes of the plaintext, we just need to perform the inverse operation of byte substitution during the decryption process.

If the rows are shifted to the left during the process of encryption, then the encrypted rows are shifted to the right during the process of decryption. The AES key schedule provides a fresh key block at every round of the algorithm.

4. Data Flow Diagram

A Data Flow Diagram (DFD) is a traditional visual representation of the information flows within a system. A neat and clear DFD can depict the right amount of the system requirement graphically. It can be manual, automated, or a combination of both. It shows how data enters and leaves the system, what changes the information, and where data is stored. The objective of a DFD is to show the scope and boundaries of a system as a whole. It may be used as a communication tool between a system analyst and any person who plays a part in the order that acts as a starting point for redesigning a system. The DFD is also called as a data flow graph or bubble chart.

The symbols used in a data flow diagram are as follows:

Circle: A circle (bubble) shows a process that transforms data inputs into data outputs.

Data Flow: A curved line shows the flow of data into or out of a process or data store.

Data Store: A set of parallel lines shows a place for the collection of data items. A data store indicates that the data is stored which can be used at a later stage or by the other processes in a different order. The data store can have an element or group of elements.

Levels of DFD:

- The DFD may be used to perform a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels that represent increasing information flow and functional detail. Levels in DFD are numbered 0, 1, 2 or beyond. Here, we will see primarily three levels in the data flow diagram, which are: level 0 DFD, level 1 DFD and level 2 DFD.

Level 0 DFD (Context Diagram)

It is also known as fundamental system model, or context diagram represents the entire software requirement as a single bubble with input and output data denoted by incoming and outgoing arrows. Then the system is decomposed and described as a DFD with multiple bubbles. Parts of the system represented by each of these bubbles are then decomposed and documented as more and more detailed DFDs. This process may be repeated at as many levels as necessary until the program at hand is well understood. It is essential to preserve the number of inputs and outputs between levels. This concept is called levelling.

The objective of Level-0 DFD is to simplify complex DFDs. Each level provides details about the portion of the level above it. The complexity of the bubble determines the number of levels of the final DFD. Different parts may have different levels but the levels should be consistent. The Level-0 DFD is called the context diagram which represents the entire computation by one large bubble. All the inputs and the outputs are to be specified.

The Level 0 DFD (Context Diagram) for the Encryption process of AES is shown in the figure below:

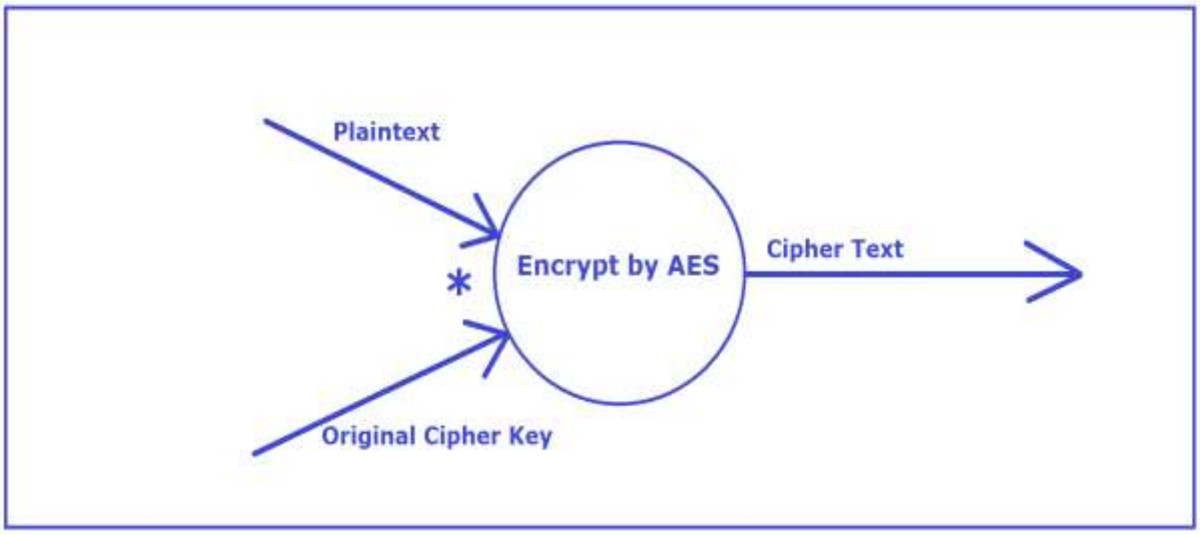


Figure 1: Level-0 DFD (Encryption)

The Level 0 DFD (Context Diagram) for the Decryption process of AES is shown in the figure below:

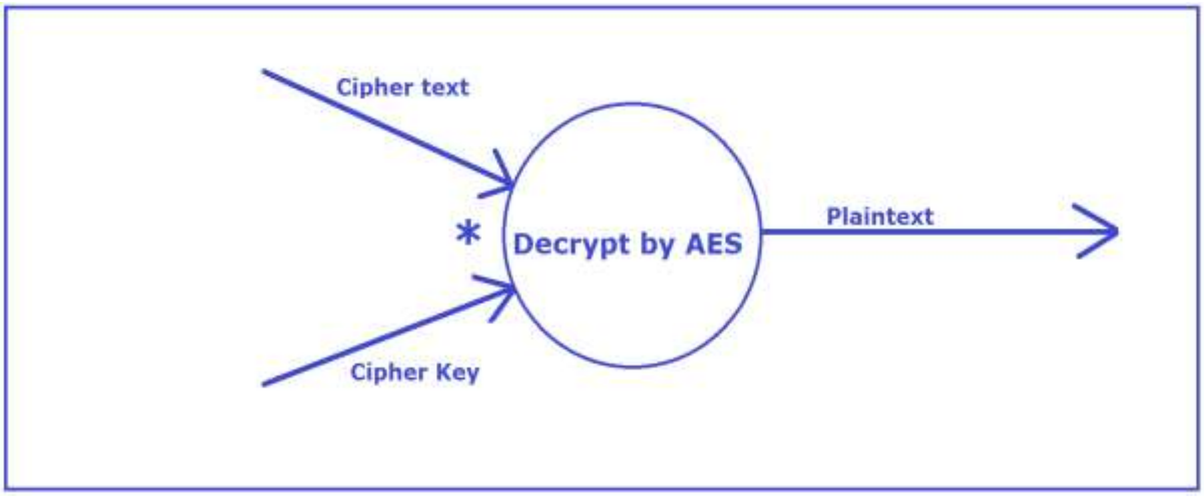


Figure 2: Level-0-DFD (Decryption)

The original cipher key is generated by a key schedule which is then expanded into blocks by a key expansion method which is depicted in the level-1 DFD to follow. The encryption takes place at the sender’s end and the decryption takes place at the receiver's end. The Advanced Encryption Standard completes its' work at the two ends: the sender and the receiver. The file that is to be sent is encrypted at the sender's end and sent to the receiver where the decryption is performed on the encrypted file. The original file is recovered on decryption. The original file has been depicted in the above DFDs as plaintext and the encrypted file has been depicted in the above DFDs as cipher text.

Level 1 DFD

In 1-level DFD, a context diagram is decomposed into multiple bubbles/processes. In this level, we highlight the main objectives of the system and breakdown the high-level process of 0-level DFD into subprocesses. The Level-1 DFD for Encryption is shown in the figure below:

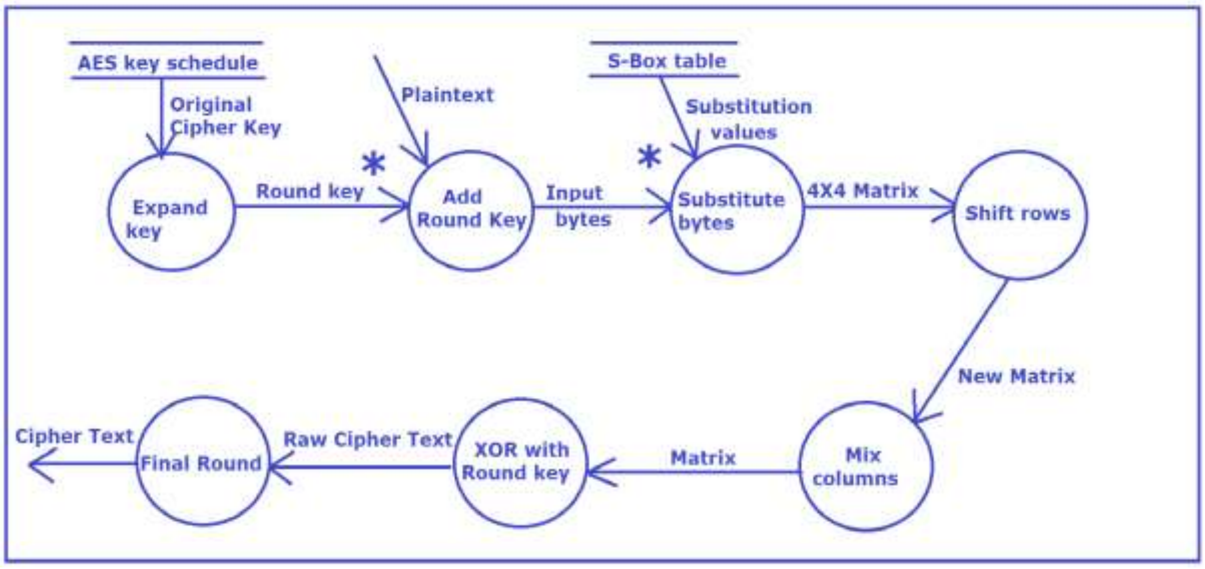


Figure 3: Level-1 DFD (Encryption)

In the above diagram, we see that the processes involved are:

- Expand Key
- Add Round Key
- Substitute bytes
- Shift rows
- Mix columns
- XOR with round key
- Final Round

The input to the system is the plaintext and the output of the system is the cipher text. We make use of the data stores 'AES key schedule' and 'S-Box' to assist the process of encryption. The AES key schedule generates the original cipher key to be used for encrypting the plaintext in the initial rounds. The S-Box is the Substitution Box which generates the Substitute bytes to replace the input bytes. This results in the creation of a 4X4 Matrix passed on to the next phases.

The Level-1 DFD for the decryption process is shown in the figure below:

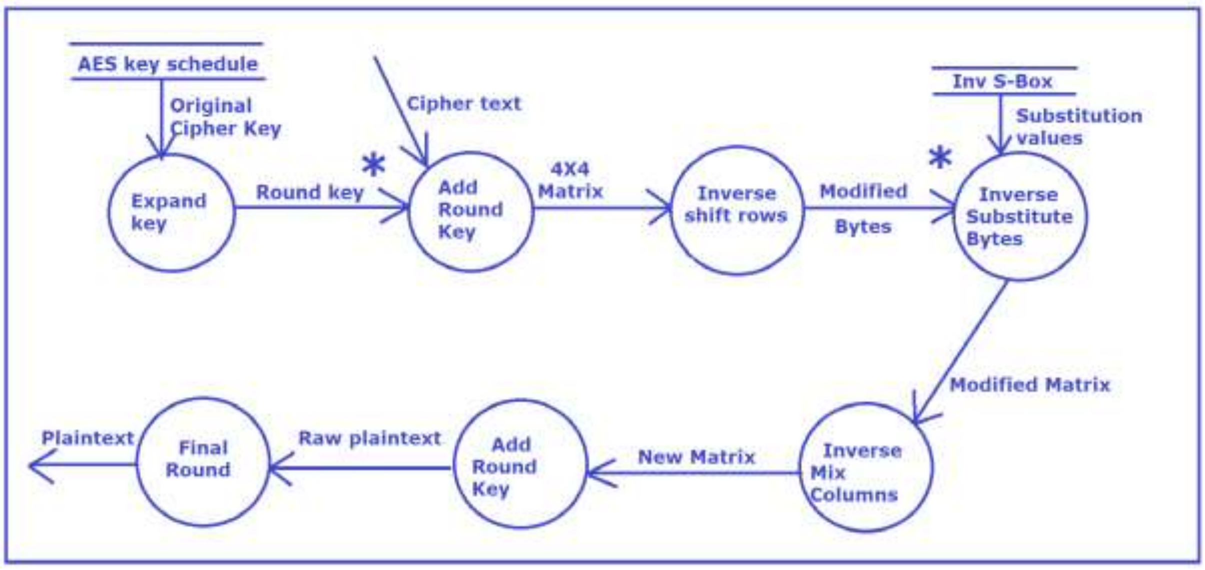


Figure 4: Level-1 DFD (Decryption)

In the above diagram, we see that the processes involved are:

- Expand Key
- Add Round Key
- Inverse Shift Rows
- Inverse Substitute Bytes
- Inverse Mix columns
- Add round key
- Final Round

5. Data Representation

Before we move on to process specifications, we need to understand the way data is represented and used in the AES algorithm. The Algorithm makes use of the block and the key to represent data. AES allows for block sizes of 128, 168, 192, 224, and 256 bits. AES allows key sizes of 128, 192, and 256 bits. The standard encryption uses AES-128 where both the block and key size are 128 bits. The block size is commonly denoted as N_b and the key size is commonly denoted as N_k. N_b refers to the number of columns in the block where each row in the column consists of four cells of 8 bytes each for AES-128.

The following example will show how data is broken up into blocks. Using AES-128 means that each block will consist of 128 bits. N_b can be calculated by dividing 128 by 32. The 32 comes from the number of bytes in each column. The original plaintext is stored in bytes in a block. The block can be an array or a structure containing an array. For example, the text "Here is a text" is stored in a block as shown in the following figure:

H		A	x
e	i		t
r	s	t	
e		e	

Figure 5: Block of plaintext

Each character is stored in a cell of the block. The blank cells shown in the diagram are not really blank as they represent the spaces in the text. Depending on how the algorithm is implemented the characters may be stored as integer values, hexadecimal values, or even binary strings. All three ways represent the same data. Most diagrams show the hexadecimal values, however, integer and string manipulation is much easier to do when actually programming AES. Figure 4 shows the values as characters for demonstration purposes to show how the text is stored into the block. The plain text is stored into blocks column by column and block by block until all the data is stored. In the example used above there were exactly 16 characters used for simplicity. In order to use the algorithm the data must be a multiple of the block size, since all blocks need to be complete. When the data is not a multiple of the block size, some form of padding must be used. Padding is when extra bits are added to the original data. One forms of padding includes adding the same bytes until the desired size is reached. Another option is padding with all zeros and having the last byte represent the number of zeros. Padding with null characters or random characters are also forms of padding that can be used. Once a form of padding is chosen the data is represented as some number of complete blocks. The last thing needed before using the algorithm is the key. The key also known as the cipher key is also the same size as the block in this example. Unlike most data and transformations, the cipher key can have any values chosen by the designer with no restrictions as long as the key is of the correct length. The key is also stored as a block similar to the plain text. When the plain text data is stored into blocks and the key is chosen the encryption algorithm can be applied.

6. Process Specification

It defines what must be done in order to transform the inputs to outputs. A process must be expressed in a form that can be verified by the user and the system's analyst. It should also be designed such that it can be effectively communicated to the various audiences involved.

The tools used for process specification are as follows:

(i) Structured English
(ii) Pre-post conditions
(iii) Decision table
(iv) Flowcharts

The tools should be used on the basis of:

- User’s preference
- Analysts’ preference
- Nature of various processes

The specification language should have a few characteristics. It should be:

- Modifiable
- Understandable
- Unambiguous
- Easy to learn and use

Natural language is not used for the purpose of process specification because it is imprecise and ambiguous. A lot of ambiguity prevails while using natural language as it has multiple interpretations. Therefore, the tools of process specification are used to avoid ambiguity.

We shall first use the method of Structured English for process specification.

6.1 Structured English

It is basically English with structure. It’s purpose is to strike a reasonable balance between the precision of a formal programming language and the casual informality and readability of the English language.

A sentence may consist of algebraic equation such as R=(B*B) - (4*A*C)

or a simple sentence consisting of verbs and objects such as:

LET Y=10; ADD 5 to Y

We shall now use Structured English to describe the following processes:

Process: Expand Key

```
Fetch Array key[][]

Fetch Array extended_key[][]

LET temp[4], rcon

LET i=0, j=0

DO-WHILE (i<4)
```

```
     LET j=0
      DO-WHILE (j<4)
            extended_key[i][j]=key[i][j]
            j=j+1
       END DO
   i=i+1
END DO
LET i=4
 DO-WHILE (i<44)
        LET j=0
        DO-WHILE (j<4)
            temp[j]=extended_key[j][i-1]
          j=j+1
         END DO
  IF (i MOD 4 = 0)
      rcon=Rcon((i/4)-1)
                   PRINT rcon
                   rotate_l(temp,1)
                   Substitute_bytes(temp)
                   temp[0]=temp[0] XOR rcon
 ENDIF
DO-WHILE (j<4)
            extended_key[j][i]=(extended_key[j][i-4] XOR temp[j])
          j=j+1
         END DO
i=i+1
END DO
```

Process: Add Round Key

```
RECEIVE r (total number of rounds)
LET i=0
LET j=0
```

```
DO-WHILE (i<4)
      LET j=0
     DO-WHILE (j<4)
         input[j][i] = input[j][i] XOR extended_key[j][(4*r)+i]
          j=j+1
      END DO
     i=i+1
END DO
```

Process: Shift Rows

```
LET i=1
DO-WHILE (i<4)
    Invoke method rotate_l(input[i], i)
    i=i+1
END DO
```

Process: Inverse Shift Rows

```
LET k=1
DO-WHILE (k<=3)
    LET j=1
   DO-WHILE (j<=k)
      temp=input[k][3]
        LET i=3
       DO-WHILE (i>=1)
            input[k][i]=input[k][i-1]
          i=i-1
       END DO
    input[k][0]=temp
   j=j+1
  END DO
k=k+1
END DO
```

Process: Substitute Bytes

```
LET i=0
DO-WHILE (i<4)
    LET y = inp[i] MOD 16
    LET x = inp[i] / 16
    LET inp[i] = box[x][y]
    i=i+1
END DO
```

Process: Inverse Substitute Bytes

```
RECEIVE Array inp[4]
LET l=0
DO-WHILE (l<4)
          LET y=inp[l] MOD 16
          LET x=inp[l] / 16
          inp[l]=Invsbox[x][y]
      PRINT Invsbox[x][y]
    l=l+1
END DO
```

Process: Mix Columns

```
LET i=0
DO-WHILE (i<4)
   LET word[0] = multiply(input[0][i],2) XOR multiply(input[1][i],3) XOR input[2][i] XOR input[3][i]
          LET word[1] = multiply(input[1][i],2) XOR multiply(input[2][i],3) XOR input[3][i] XOR input[0][i]
LET word[2] = multiply(input[2][i],2) XOR multiply(input[3][i],3) XOR input[0][i] XOR input[1][i]
          LET word[3] = multiply(input[3][i],2) XOR multiply(input[0][i],3) XOR input[1][i] XOR input[2][i]
    LET j=0
    DO-WHILE (j<4)
```

```
          input[j][i]=word[j]
        j=j+1
    END DO
i=i+1
END DO
```

Process: Inverse Mix Columns

```
LET i=0
DO-WHILE (i<4)
   LET word[0] = multiply(input[0][i],14) XOR multiply(input[1][i],11) XOR multiply(input[2][i],13) XOR multiply(input[3][i],9)
   LET word[1] = multiply(input[1][i],14) XOR multiply(input[2][i],11) XOR multiply(input[3][i],13) XOR multiply(input[0][i],9)
   LET word[2] = multiply(input[2][i],14) XOR multiply(input[3][i],11) XOR multiply(input[0][i],13) XOR multiply(input[1][i],9)
   LET word[3] = multiply(input[3][i],14) XOR multiply(input[0][i],11) XOR multiply(input[1][i],13) XOR multiply(input[2][i],9)
   LET j=0
   DO-WHILE (j<4)
          input[j][i]=word[j]
        j=j+1
    END DO
i=i+1
END DO
```

Process: Encryption Final Round

```
    Execute process Substitute Bytes
    Execute process Shift Rows
    Execute process Add Round Key
    Generate cipher text
```

Process: Decryption Final Round

```
    Execute process Inverse Shift Rows
    Execute process Inverse Substitute Bytes
    Execute process Add Round Key
    Generate plaintext
```

Some functions that aid the execution of the above processes are listed below:

FUNCTION: multiply

```
RECEIVE inp, x
LET flag=2
LET temp=inp
    IF x=1
      Return inp
   ENDIF
  DO-WHILE (flag<=x)
        IF ( temp AND 0x80 )

                         temp = temp LEFT-SHIFT 1
                          temp = temp XOR 0x1B;

ELSE
    temp = temp LEFT-SHIFT 1
END IF
flag = flag * 2
END DO
x = x – (flag/2)
IF (x >= 1)
    temp = temp XOR multiply(inp,x)
ENDIF
return temp
    // end of function
```

FUNCTION: Invs_box

```
LET i=0
LET row=0
LET col=0
LET p=1
LET q=1
DO-WHILE (p!=1)
                p = p XOR (p LEFT-SHIFT 1) XOR (p AND 0x80 ? 0x1B : 0)
                q = q XOR (q LEFT-SHIFT 1)
                q = q XOR (q LEFT-SHIFT 2)
                q = q XOR (q LEFT-SHIFT 4)
                q = q XOR (q & 0x80 ? 0x09 : 0)
                table[p]=q
      END DO

LET p=1
   LET i=0
 DO-WHILE (i<256)
     IF (i MOD 16 = 0)
      row = row +1
      col = 0
    PRINT "\n"
   ENDIF
p = left_rot(i,1) XOR left_rot(i,3) XOR left_rot(i,6) XOR 0x05
                q = table[p]
                Invsbox[row][col] = q
i=i+1
END DO
// end of function
```

FUNCTION: rotate_l

```
RECEIVE Array word[], shift
LET j=1
```

```
DO-WHILE (j<=shift)
      temp = word[0]
       LET i=0
       DO-WHILE (i<3)
            word[i] = word[i+1]
            i=i+1
         END DO
      word[i] = temp
     j=j+1
END DO
// end of function
```

FUNCTION: transpose

```
RECEIVE Array arr
LET i=0
DO-WHILE (i<4)
      LET j=i+1
       DO-WHILE (j<4)
          arr[i][j] = arr[j][i] + arr[i][j] - (arr[j][i] = arr[i][j])
        j=j+1
      END DO
  i=i+1
END DO
// end of function
```

FUNCTION: Rcon

```
RECEIVE x
LET i=0
LET temp=01
IF x=0
   Return temp
ENDIF
```

```
LET i=1
DO-WHILE (i<=x)
      temp = multiply(temp, 2)
    i=i+1
END DO
Return temp
// end of function
```

FUNCTION: encryption

```
RECEIVE plaintext
Input key
Invoke method extension_of_key()
Execute process Add Round Key
LET k=1
DO-WHILE (k<10)
LET i=0
DO-WHILE (i<4)
    Execute process Substitute Bytes
    i=i+1
  END DO
PRINT input
Execute process Shift Rows
Execute process Mix Columns
Execute process Add Round Key
k=k+1
END DO
Execute Process Encryption Final Round
// end of function
```

FUNCTION: decryption

```
RECEIVE cipher_text
Input key
```

```
Execute Process Expand key
Execute Process Add Round Key
 LET k=1
DO-WHILE (k<10)
     Execute Process Inverse Shift Rows
    LET i=0
   DO-WHILE (i<4)
       Execute Process Inverse Substitute Bytes
       i=i+1
    END DO
 Execute Process Inverse Mix Columns
Execute Process Add Round Key
k=k+1
END DO
Execute Process Decryption Final Round
// end of function
```

FUNCTION: main

```
          // Function controlling all processes involved in the system
       LET con=0
       Invoke s_box()
       Invoke Invs_box()
       DO-WHILE (con==0)
            Input dec
           If (dec=='E' OR dec=='e')
             Invoke encryption()
           else if (dec=='D' OR dec=='d')
               Invoke decryption()
           else
               Display error message
          Input con
```

```
    END DO
// end of function
```

6.2 PRE/POST CONDITIONS

It is a convenient way to describe the function without considering the algorithm and procedure to be used.

It is used in the following conditions:

- There are many different algorithms that can be used.
- There is a tendency of the user to give the algorithm she/he has been using for decades.
- Programmer is supposed to explore different algorithms.
  There are two main parts: PRE Condition and POST Condition.

PRE CONDITION:

- Describes what must be true before the process begins operating.
- It describes what input must be available.
- It describes what relationship must exist among inputs.
- It describes relationship between inputs and data stores.

POST CONDITION:

- Describes what must be true when a process has finished operating.
- It describes the output that must be generated by the process.
- It describes the relationships that must exist between the output values and the original input values.
- It describes output values in one or more stores.
- It describes changes made to the stores.

  We shall use Pre/Post conditions for process specification of each process described in the Data Flow Diagram.

The Pre/Post conditions for the processes that build up the algorithms of **encryption** are detailed below:

Process: Expand Key

Pre-Condition 1: 2-dimensional array key[][] and extended_key[][] must be declared.

Post-Condition 1: Array extended_key[][] is filled with calculated values and made ready for further operation.

Process: Add Round Key

Pre-Condition 1: Array input[][] and Array extended_key[][] are filled with appropriate values from the previously executing processes.

Post-Condition 1: Array input[][] is modified as each element of the array is XOR-ed with each element of the array extended_key[][].

Process: Substitute Bytes

Pre-Condition 1: Array input[][] is modified by previous process and array sbox[][] is declared and filled with appropriate values.

Post-Condition 1: Array input[][] is modified by specific values from sbox[][].

Process: Shift Rows

Pre-Condition 1: Array input[][] is modified by previous processes.

Post-Condition 1: Array input[][] is further modified as the rows of the matrix are shifted and rearranged as per the algorithm's specifications.

Process: Mix Columns

Pre-Condition 1: 1-dimensional array word[] is declared and the modified array input[][] is received as modified by the previous processes.

Post-Condition 1: Array word[] is filled with appropriate values as per the specifications of the algorithm and specific elements of array input[][] are replaced with elements of array word[].

Process: XOR with Round Key

Pre-Condition 1: Array input[][] and extended_key[][] are received as modified by previous processes.

Post-Condition 1: Array input[][] is modified as each element of the array is XOR-ed with each element of the array extended_key[][].

Process: Final Round

Pre-Condition 1: All previously mentioned processes have successfully executed.

Post-Condition 1: The cipher text is generated and made ready to be passed on to the processes involved in decryption.

The Pre/Post conditions for the processes that build up the algorithms of **decryption** are detailed below:

Process: Expand Key

Pre-Condition 1: 2-dimensional array key[][] and extended_key[][] must be declared.

Post-Condition 1: Array extended_key[][] is filled with calculated values and made ready for further operation.

Process: Add Round Key

Pre-Condition 1: Array input[][] and Array extended_key[][] are filled with appropriate values from the previously executing processes.

Post-Condition 1: Array input[][] is modified as each element of the array is XOR-ed with each element of the array extended_key[][].

Process: Inverse Shift Rows

Pre-Condition 1: Array input[][] is modified by XOR operation of each element with each element of Array extended_key[][] in the previous process.

Post-Condition 1: Array input[][] is further modified by shifting of rows.

Process: Inverse Substitute Bytes

Pre-Condition 1: 2-dimensional array Invsbox[][] should be declared.

Post-Condition 1: 1-dimensional array inp[] is filled with specific elements of array Invsbox[][].

Process: Inverse Mix Columns

Pre-Condition 1: 1-dimensional array word[] is declared and 2-dimensional array input[][] is available for operations.

Post-Condition 1: Array word[] is filled with appropriate values and specific elements of array input[][] are replaced with elements of array word[].

Process: Final Round

Pre-Condition 1: All previously mentioned processes have successfully executed.

Post-Condition 1: The plaintext is generated.

7. Data Dictionary

A data dictionary is a file or a set of files that includes a database's metadata. The data dictionary holds records about other objects in the database, such as data ownership, data relationships to other objects, and other data. The data dictionary is an essential component of any relational database. Ironically, because of its importance, it is invisible to most database users. Typically, only database administrators interact with the data dictionary.

A data dictionary is helpful in the following ways:

- Create a listing of all data items
- Find a data item name from a description
- Design the software and the test cases

Data dictionary is the logical characteristic of the data of the system. It is the listing of all data items for user/ analyst communication. It contains the meaning of flows/stores and the composition of data packets/stores. It provides understandable data by using the conventions.

The conventions used are listed below:

- * * comments
- [ ] one of the several choices
- @ key field of a store
- | choice
- = composed of
- + and
- ( ) optional
- n{ }m at least n, at most m
- 0{ }m zero or more, at most m
- n{ }   at least n, no limit

The data dictionary for the entire system (metadata) has been depicted in the following table:

Data Item	Data Type	Bytes of storage	Size for display (bytes)
key	uint8_t	1 X size of array	1 X size of array
input	uint8_t	1 X 4 X 4 = 16	1 X 4 X 4 = 16
extended_key	uint8_t	1 X 4 X 44 = 176	1 X 4 X 44 = 176
Invsbox	uint8_t	1 X 16 X 16= 256	1 X 16 X 16= 256
Sbox	uint8_t	1 X 16 X 16= 256	1 X 16 X 16= 256

Table 2: Metadata

The above table represents the metadata of the entire system.

The relation AES Key Schedule is as follows:

AES_Key_Schedule (<u>key</u>, Type)

AES_Key_Schedule = @key + Type

key = alpha + numeric   * Key Field *

Type = [string | integer]

alpha = 0 { }

numeric = 1 { }

8. Software Requirement Specification Document

A Software Requirements Specification (SRS) is a document that describes the nature of a project, software or application. In simple words, SRS document is a manual of a project provided it is prepared before one starts a project or application. This document is also known by the names SRS report, software document. A software document is primarily prepared for a project, software or any kind of application. It is a kind of contract between the client and the developers.
There are a set of guidelines to be followed while preparing the software requirement specification document. This includes the purpose, scope, functional and non-functional requirements, software and hardware requirements of the project. In addition to this, it also contains the information about environmental conditions required, safety and security requirements, software quality attributes of the project etc.

8.1 Purpose

The purpose of the project is to build an encryption-decryption algorithm using Advanced Encryption Standard (AES) so that it can be used to exchange confidential data and information over a platform or network. The project ensures data protection using AES algorithm.

8.2 Document Conventions

The document uses the following conventions:

DB	Database
AES	Advanced Encryption Standard
2D	Two-dimensional
3D	Three-dimensional
1D	One-dimensional
SRS	Software Requirement Specifications
DFD	Data Flow Diagram
DS	Data Store
SDS	Software Design Specifications

8.3 Intended audience and reading suggestions

This project is a prototype for the AES algorithm and it is restricted within the University premises. This project is useful for data protection and exchange of information in a secured manner.

8.4 Project Scope

The project can be extended for encryption of text into images and used in steganography with the help of powerful IDEs and tools like MATLAB. It can be used in practical applications for exchange of data over a local network.

8.5 Product Perspective

Encryption: This includes procedures which convert the plaintext to cipher text using keys and passes on the cipher text to the procedures involved in decryption.

Decryption: This includes procedures which receive the cipher text and use the decryption algorithm to recover the plaintext. Non-availability of the original cipher key will result in failure.

System: The sender shall use the process of encryption to send the file and the receiver shall use the process of decryption to recover the file. The processes of encryption and decryption ensure security of the data being exchanged.

8.6 Product features

User: The intended users consist of the Sender and the Receiver. The message is to be sent by the sender and received by the receiver. The receiver can only decipher the message if the original cipher key is with the receiver.

Sender: The original data(plaintext) is sent to the receiver as cipher text.

Receiver: The cipher text is received by the receiver and deciphered to generate the plaintext.

Interface: The interface is a command user interface where the program execution needs to be done. Any Linux based Operating System can be used to execute the program.

Input: The input to the process of encryption is the plaintext and the input to the process of decryption is the cipher text.

Output: The output of the process of encryption is the cipher text and the output of the process of decryption is the plaintext.

Algorithm: The Advanced Encryption Standard (AES) algorithm is used for the entire system. The implementation has been initially done on the basis of 128-bit key but it can be extended to 256-bit key or more for better security.

8.7 Operating environment

The operating environment specifications for the software system is as follows:

- Pentium IV Processor and above
- Windows or Linux-based operating system
- 32-bit Operating System and above
- Minimum 2GB RAM

8.8 Design and implementation constraints

The coding has been done in C programming language. Modular approach has been maintained so as to ensure low coupling and high cohesion. Low coupling ensures that a module has the least interdependence on other modules. High cohesion ensures that a module makes use of the most resources of itself.

Further details about design and implementation constraints have been mentioned in the design document.

8.9 Assumption dependencies

- Let us assume that the transaction on the sender side and the transaction on the receiver side are single independent transactions.
- Given the development constraints of the project, the project has been developed so as to meet the encryption-decryption standards pertaining to text files only.
- However, the project can be extended to work with image files, encrypting text into images.
- Given time and implementation constraints, the project does not include network related work. File transfer mechanisms are not being considered as a part of the current project.

8.10 Stimulus / Response sequences

- The sender encrypts the plaintext using a key.
- The sender sends the encrypted file to the receiver.

- The receiver receives the encrypted file and the original cipher key.
- The receiver decrypts the cipher text and regenerates the plaintext.

8.11 Functional requirements

- Arrays have been used to contain the key and the plaintext.
- 2-dimensional array has been used to store key, extended key, plaintext and ciphertext.
- The data members have been passed on to functions as formal parameters in most of the cases.
- However, global data members have also been used.
- Modular approach has been followed.

8.12 Non-functional requirements

Performance Requirements: The steps involved to perform the implementation of the system include command-based interface and the modules for encryption and decryption. It also requires the proper declaration and use of data structures for the implementation of the same.

Software Quality Attributes:

Availability: A list of keys should be available to serve the purpose of encryption-decryption operations by multiple users.

Correctness: The plaintext as received by the receiver should be the same as the one sent by the sender. There should be no variation between the sent item and the received item. The correctness of the system is to be maintained.

Maintainability: The AES key schedule should maintain a proper list of keys in order to serve the purpose of a sufficient number of customers. The provision and supply of keys for the purpose is up to the developers.

Usability: The system should satisfy a maximum number of customer needs.

Reliability: The system performs its intended task of encrypting the original file on the sender's side and decrypting the cipher text on the receiver's side. The system needs to maintain the integrity of the original file that is encrypted and sent by the sender to the receiver.

Modularity: The system has been designed using the modular approach of software development. Hence it can be decomposed into modules in order to study the system.

Reusability: The software can be used with changes made to it in order to expand it to implement larger projects such as image and audio encryption projects.

9. Mapping DFD into Structure Charts

Systems whose morphology or overall shape is **transform-centred** tend to be associated with low development costs, low maintenance costs, and low modification costs. Such low-cost systems tend to be highly factored; that is, the high-level modules make most of the decisions and the low-level modules accomplish most of the detailed work.

Transform analysis, or transform-centred design, is a **strategy for deriving initial structural designs** that are quite good (with respect to modularity) and generally require only a modest restructuring to arrive at a final design.

Overall, the purpose of the strategy is to identify the **primary processing functions** of the system, the **high-level inputs** to those functions, and the **high-level outputs**. Transform analysis is an information flow model rather than a procedural model.

The transform analysis strategy consists of the following four major steps:

1. restating the problem as a data flow graph
2. identifying the afferent and efferent data elements
3. first-level factoring
4. factoring of afferent, efferent, and transform branches

   We define afferent data elements as follows:

   Afferent data elements are those high-level elements of data that are furthest removed from physical input, yet still constitute inputs to the system. Thus, afferent data elements are the highest level of abstraction to the term **input to the system**.

Beginning at the other end with the physical outputs, we try to identify the efferent data elements. those furthest removed from the physical output which may still be regarded as outgoing. Such elements might be regarded as **logical output data** that have just been produced by the **main processing** part of the system. Three distinct sub-strategies are used to factor the three types of subordinate modules (afferent, efferent, and transform) into lower-level subordinates. It is not necessary to completely factor one branch down to the lowest level of detail before working on another branch, but it is important to **identify all of the immediate subordinates** of any given module before turning to any other module.

The transform-centred strategy still requires judgement and common sense on the part of the designer; it does not reduce design to a series of mechanical steps.

The **Main Module** and its data flow can be understood from the following figure:

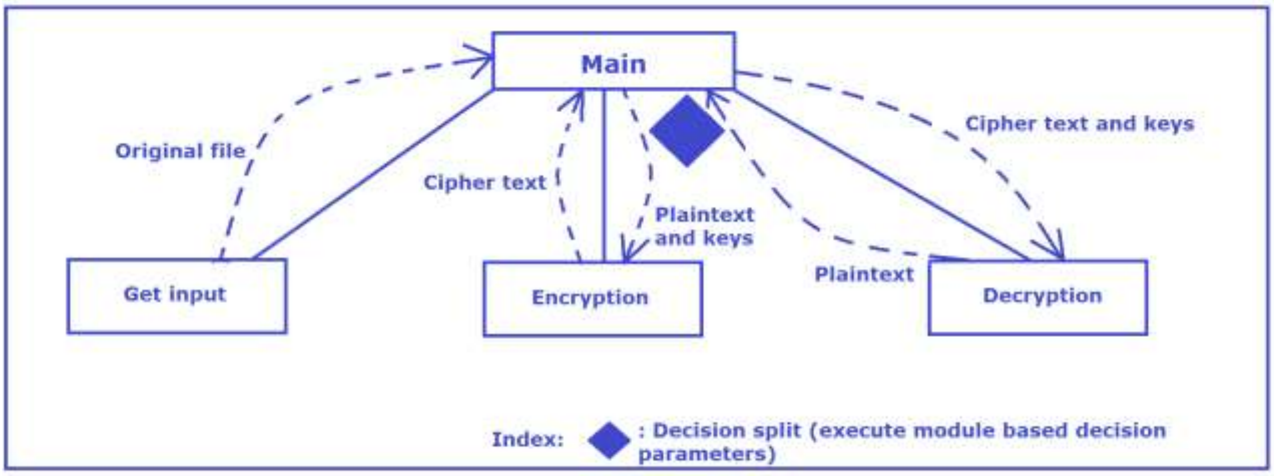


Figure 5: Understanding Main Module

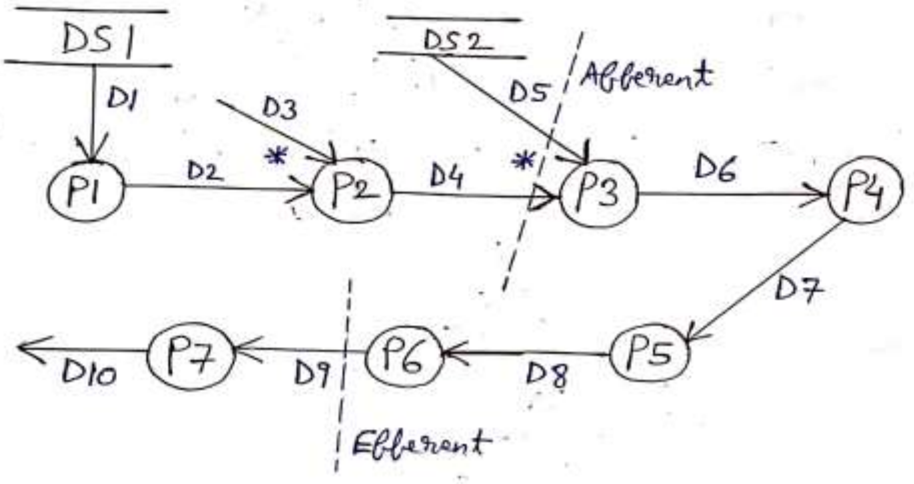
The details of the modules **Get Input, Encryption, Decryption** have been discussed in **Section 11** in details.

Now we shall perform the first cut factor, factoring afferent, factoring efferent and central transforms on the DFDs of Encryption and Decryption.

The following images show the work on first cut factor, factoring afferent, factoring efferent and central transforms on the DFDs of Encryption and Decryption. The images related to the DFD for Encryption are as follows:

Module : Encryption

DFD :



First Cut Factor :

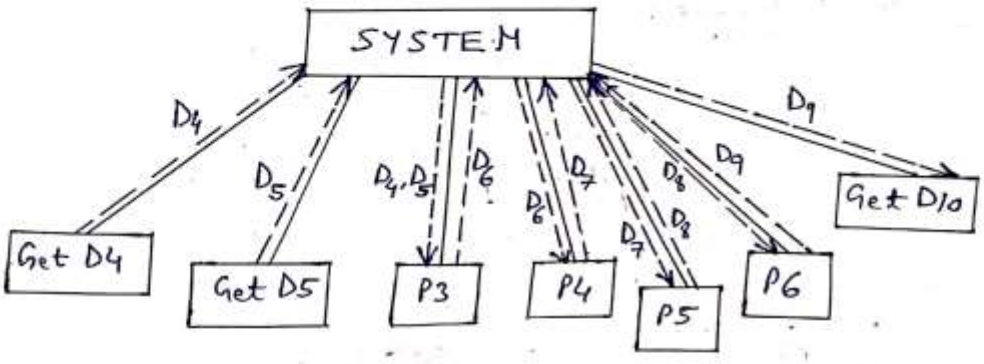


Image 1: DFD and First Cut Factor of Encryption

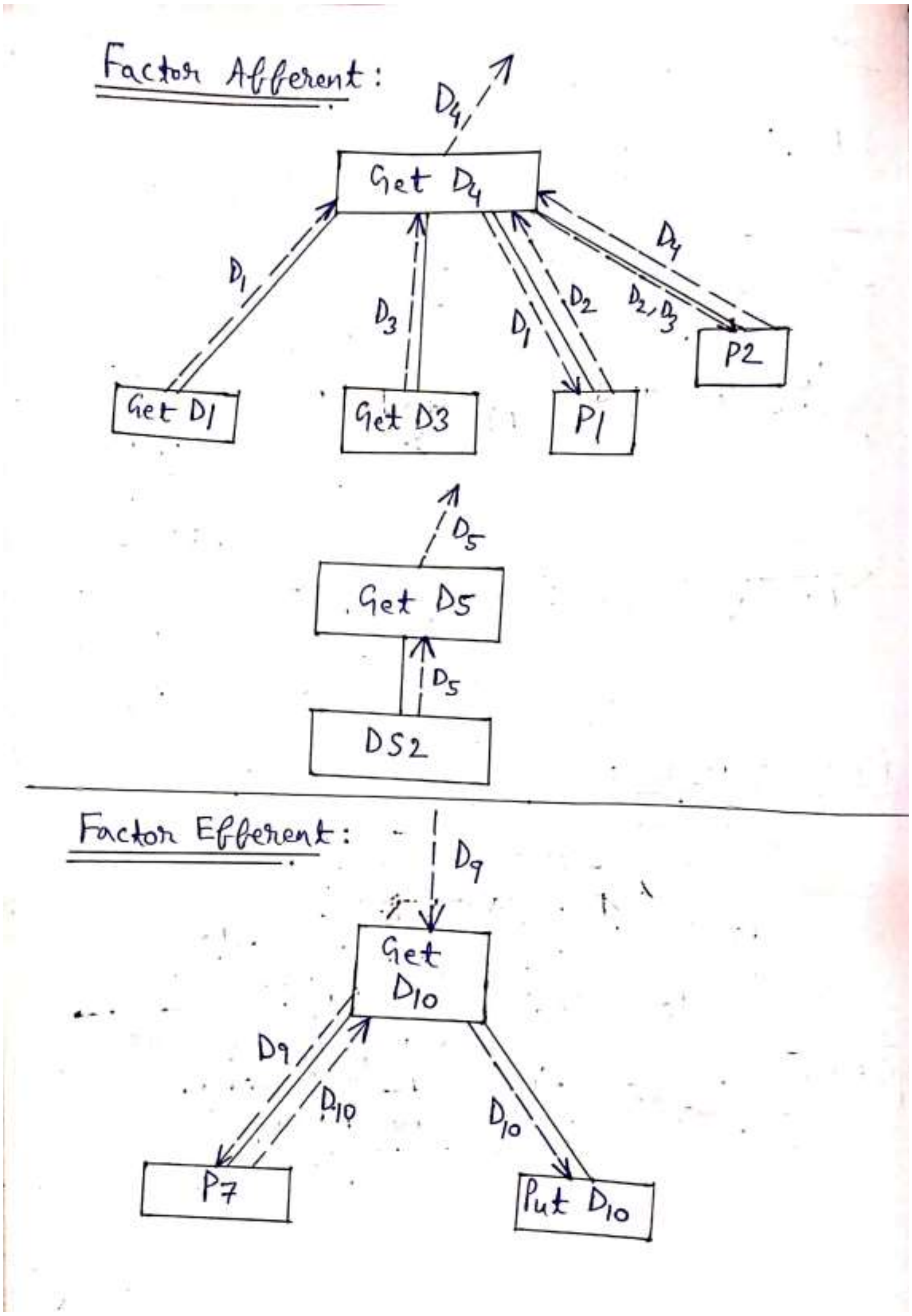


Image 2: Factor Afferent and Factor Efferent of Encryption

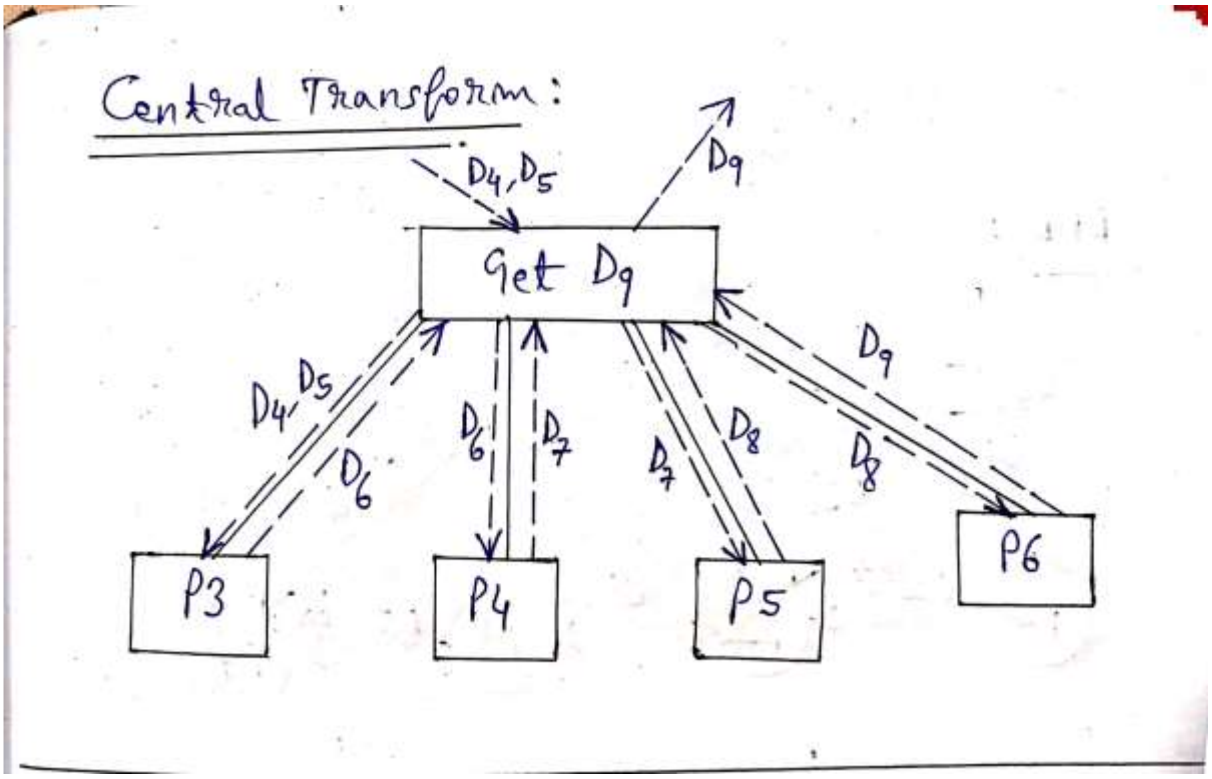


Image 3: Central Transform of Encryption

Index (Encryption):

DS1: AES Key Schedule

DS2: S-Box Table

D1: Original Cipher Key

P1: Expand Key

D2: Round Key

D3: Plaintext

P2: Add Round Key

D4: Input Bytes

D5: Substitution values

P3: Substitute Bytes

D6: 4X4 Matrix

P4: Shift Rows

D7: New Matrix

P5: Mix Columns

D8: Matrix

P6: XOR with Round Key

D9: Raw Cipher Text

P7: Final Round

D10: Cipher Text

The images related to the DFD for Decryption are as follows:

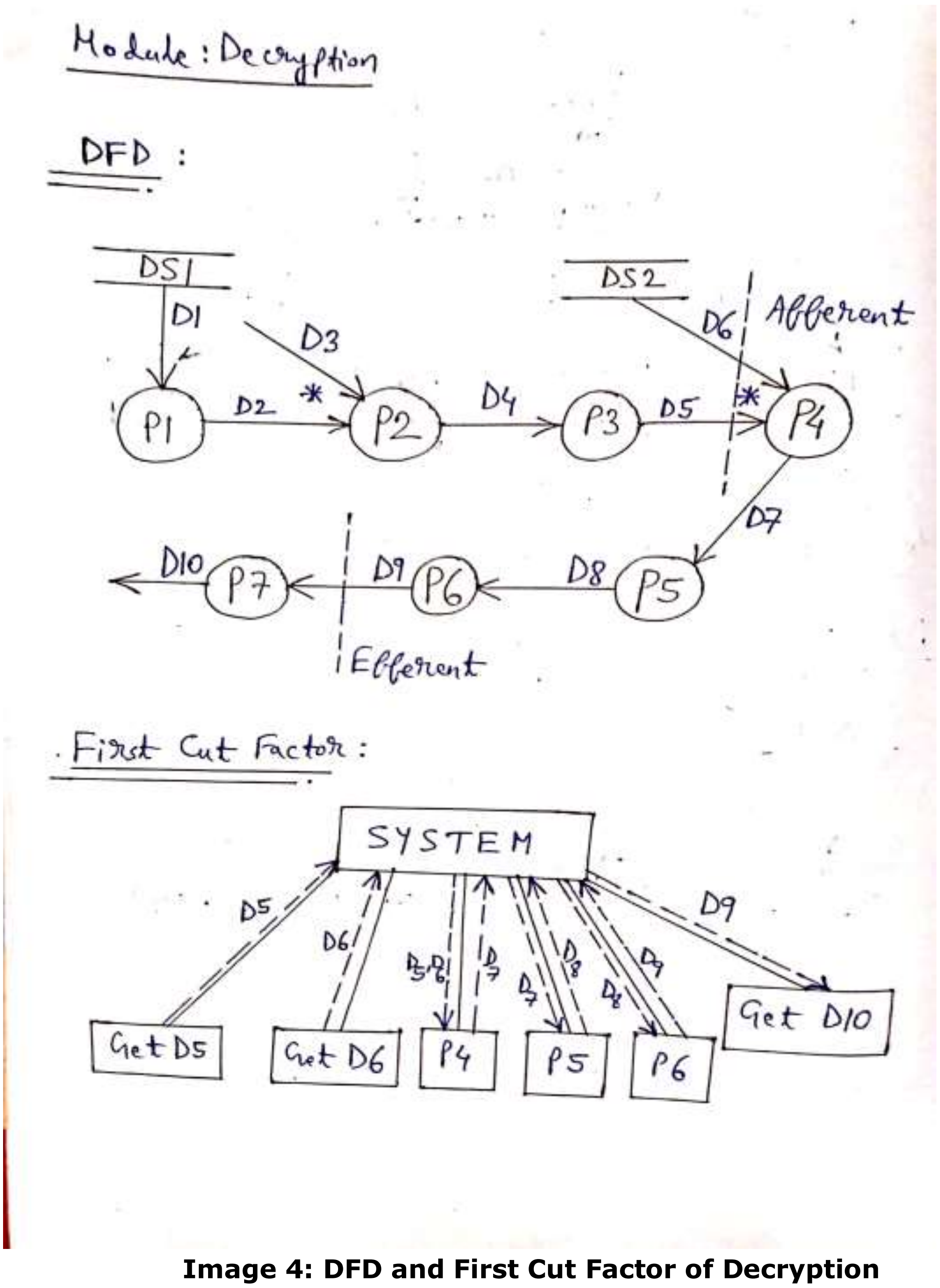


Image 4: DFD and First Cut Factor of Decryption

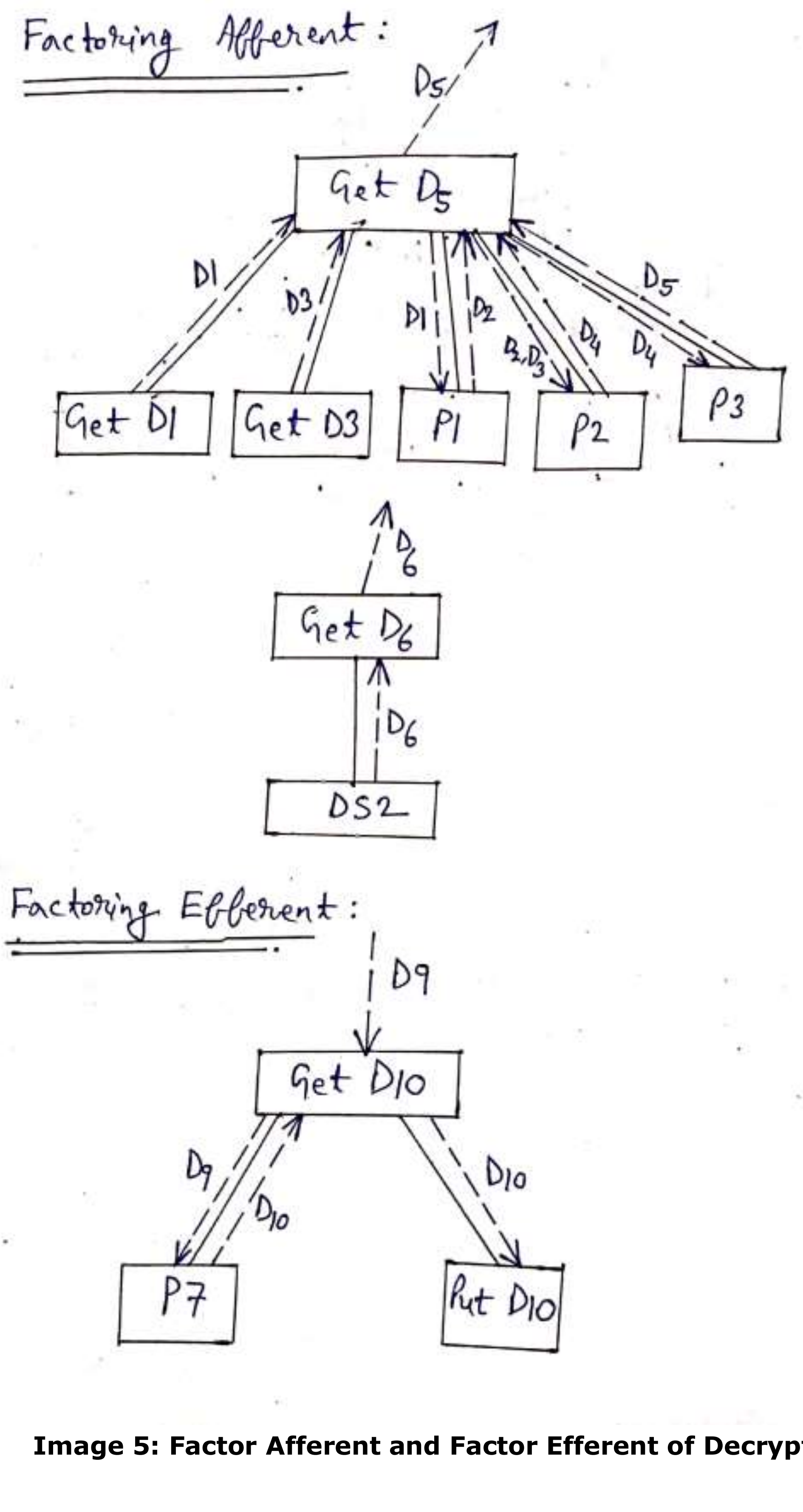


Image 5: Factor Afferent and Factor Efferent of Decryption

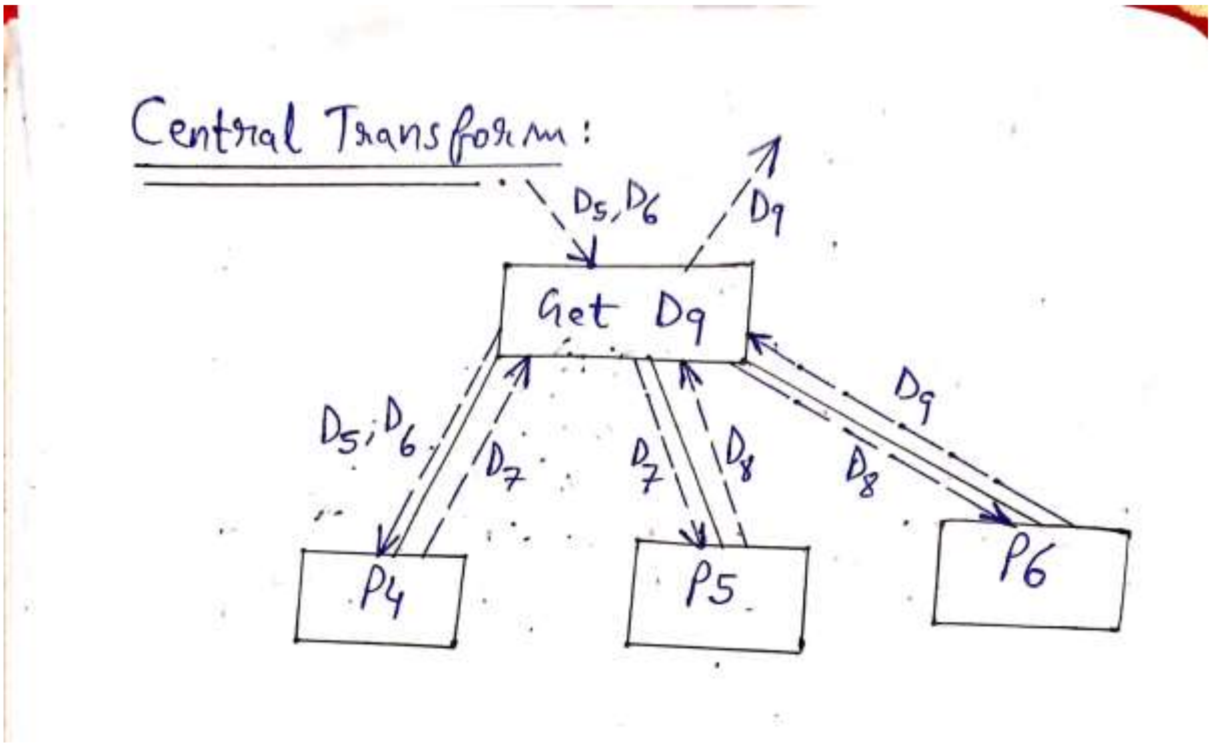


Image 6: Central Transform of Decryption

Index (Decryption):

DS1: AES Key Schedule

DS2: Inverse S-Box Table

D1: Original Cipher Key

P1: Expand Key

D2: Round Key

D3: Cipher text

P2: Add Round Key

D4: 4 X 4 Matrix

D5: Modified Bytes

P3: Inverse Shift Rows

D6: Substitution values

P4: Inverse Substitute Bytes

D7: Modified Matrix

P5: Inverse Mix Columns

D8: New Matrix

P6: Add Round Key

D9: Raw Plaintext

P7: Final Round

D10: Plaintext

10. Principles of Cohesion and Coupling - Design optimisation

The principles of coupling and cohesion have been used to optimise the software design. The principles of coupling and cohesion require that the system has low coupling and high cohesion. Coupling is the measure of the degree of interdependence between the modules. A good software will have low coupling.

Types of Coupling:

- **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled. In data coupling, the components are independent to each other and communicating through data. Module communications don't contain tramp data.
- **Stamp Coupling** In stamp coupling, the complete data structure is passed from one module to another module. Therefore, it involves tramp data. It may be necessary due to efficiency factors.
- **Control Coupling:** If the modules communicate by passing control information, then they are said to be control coupled. It can be bad if parameters indicate completely different behaviour and good if parameters allow factoring and reuse of functionality. Example- sort function that takes comparison function as an argument.
- **External Coupling:** In external coupling, the modules depend on other modules, external to the software being developed or to a particular type of hardware. Ex- protocol, external file, device format, etc.
- **Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change. So, it has got disadvantages like difficulty in reusing modules, reduced ability to control data accesses and reduced maintainability.
- **Content Coupling:** In a content coupling, one module can modify the data of another module or control flow is passed from one module to the other module. This is the worst form of coupling and should be avoided.

Our software design implements data coupling, common coupling and control coupling.

Data coupling in the current project:

The modules of Encryption and Decryption along with their subordinate modules operate on homogeneous data elements. The key and the arrays that the modules work on are of homogeneous data type. Further details have been disclosed in the Design Document in Section 11.

Common coupling in the current project:

The following global 2-D arrays are used by the modules: extended_key[4][44], sbox[16][16], Invsbox[16][16].

This contributes to common coupling in the software design.

Further details have been disclosed in the Design Document in Section 11.

Cohesion: Cohesion is a measure of the degree to which the elements of the module are functionally related. It is the degree to which all elements directed towards performing a single task are contained in the component. Basically, cohesion is the internal glue that keeps the modules together. A good software design will have high cohesion.

Types of Cohesion:

- **Functional Cohesion:** Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.
- **Sequential Cohesion:** An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.
- **Communicational Cohesion:** Two elements operate on the same input data or contribute towards the same output data.
- **Procedural Cohesion:** Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable.
- **Temporal Cohesion:** The elements are related by their timing involved. In a module connected with temporal cohesion, all the tasks must be executed in the same time-span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at the same time.
- **Logical Cohesion:** The elements are logically related and not functionally. Ex- A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.
- **Coincidental Cohesion:** The elements are not related(unrelated). The elements have no conceptual relationship other than location in source code. It is accidental and the worst form of cohesion.

The current project makes use of functional cohesion and sequential cohesion and hence ensure the most desirable cohesion in the system.

Functional cohesion in the current project:
The subordinate modules of Encryption and Decryption each achieves a single goal and ensures functional cohesion within its module.

Module Encryption is implemented with the help of the following sub-modules:

- Expand Key
- Add Round Key
- Substitute Bytes
- Shift Rows
- Mix columns
- XOR with Round Key
- Encryption Final Round

Module Decryption is implemented with the help of the following sub-modules:

- Expand Key
- Add Round Key
- Inverse Shift Rows
- Inverse Substitute Bytes
- Inverse Mix Columns
- Decryption Final Round

Each of the subordinate modules executes and achieves a single goal in the order specified above.

Sequential cohesion in the current project:
The encryption module executes its subordinate modules as expressed in the following diagram:

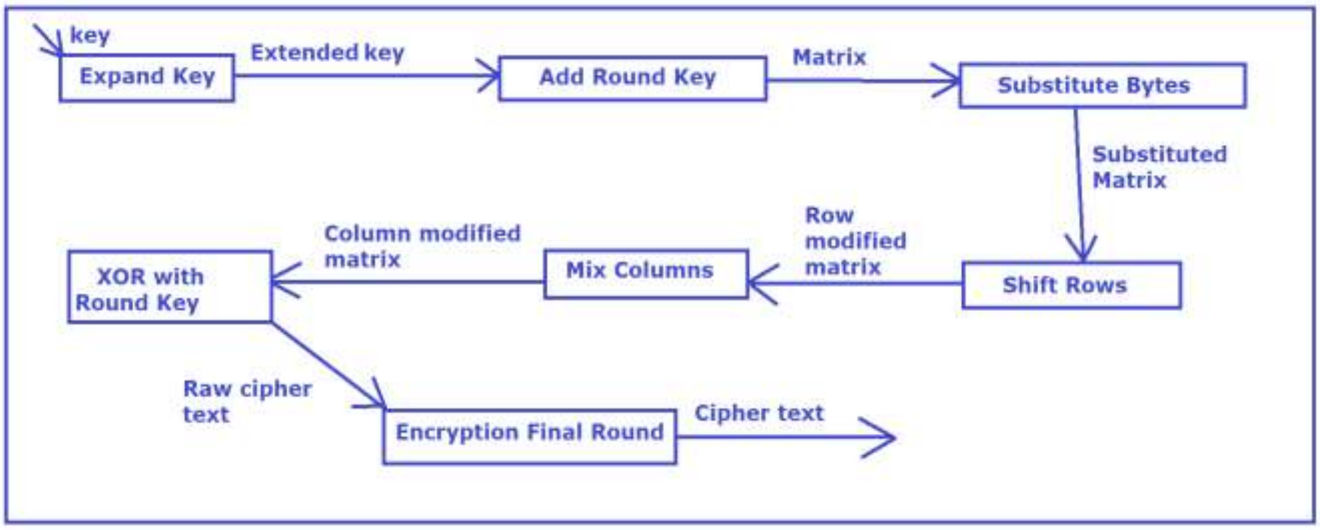


Figure 5: Sequence of execution of subordinate modules of Encryption

The modules of Encryption operate in a sequence as depicted in Figure 5. The output of one becomes the input of the other as shown above in Figure 5. The Decryption module also performs a similar task. Hence sequential cohesion is maintained by all the modules of Encryption and Decryption, thus ensuring high cohesion of the system.

11. Design Document

The software design describes the solution domain. It is a bridge between SRS and the final solution. It takes the SRS as input and produces the system design (top-level design) and the detailed design (logical design) as the output.

We shall frame the design document for the current project.

11.1 System Design (Top Level Design)

A few features of the System Design are as follows:

- It controls the major characteristics of the system.
- It considers modules required to implement the design solution.
- It includes specification of these modules.
- It includes the interconnection between these modules.
- It has impact on testability, modifiability and efficiency.

In System Design, the attention is on "what components are needed".

The components of the System Design are:

(i) Problem Specification
(ii) Major Data Structure
(iii) Modules and their specification
(iv) Design decisions

11.1.1 Problem Specification

The problem deals with sending the original file to the receiver in an encrypted way by the sender followed by deciphering the encrypted file to recover the original file. The entire process of encryption-decryption is implemented by the Advanced Encryption Standard algorithm. It is to be noted that the project does not deal with network requirements or operations. It is a basic project dealing in the encryption of the plaintext and the recovering of the original message from the cipher text.

11.1.2 Major Data Structure

The algorithm uses 1-dimensional and 2-dimensional arrays to implement the modules and processes. The arrays extended_key[4][44], sbox[16][16] and Invsbox[16][16] are declared as global variables whereas some 1-dimensional arrays have been passed and received as formal parameters. The input has been considered as an array while testing but the model can be extended to taking files as input and operating on them. The algorithm uses uint8_t as the data type of the variables on which operations are being performed.

11.1.3 Modules and their specification

The Main Module consists of the following modules:

- Get Input
- Encryption
- Decryption

The module Get Input is an atomic module whereas the other two modules further consist of the following sub-modules:

Module Encryption is implemented with the help of the following sub-modules:

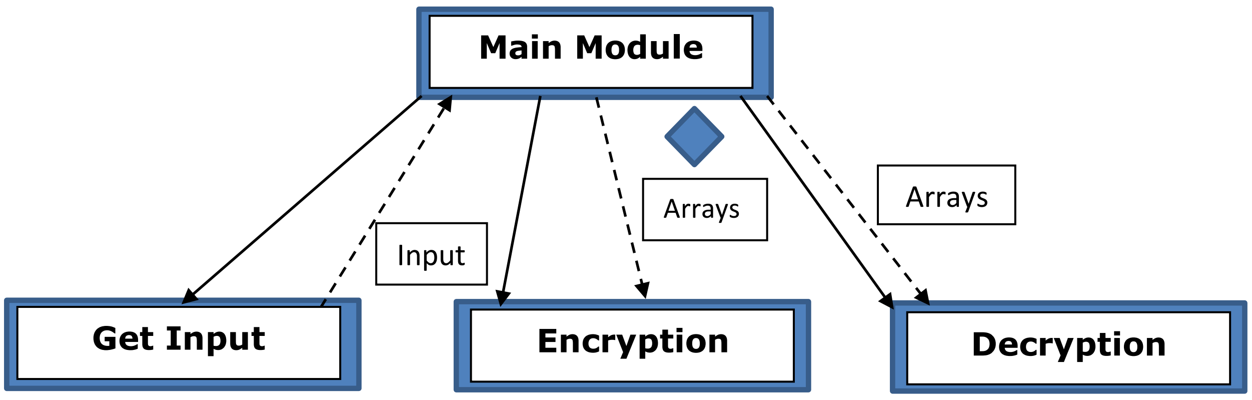
- Expand Key
- Add Round Key
- Substitute Bytes
- Shift Rows
- Mix columns
- XOR with Round Key
- Encryption Final Round

Module Decryption is implemented with the help of the following sub-modules:

- Expand Key
- Add Round Key
- Inverse Shift Rows
- Inverse Substitute Bytes
- Inverse Mix Columns
- Decryption Final Round

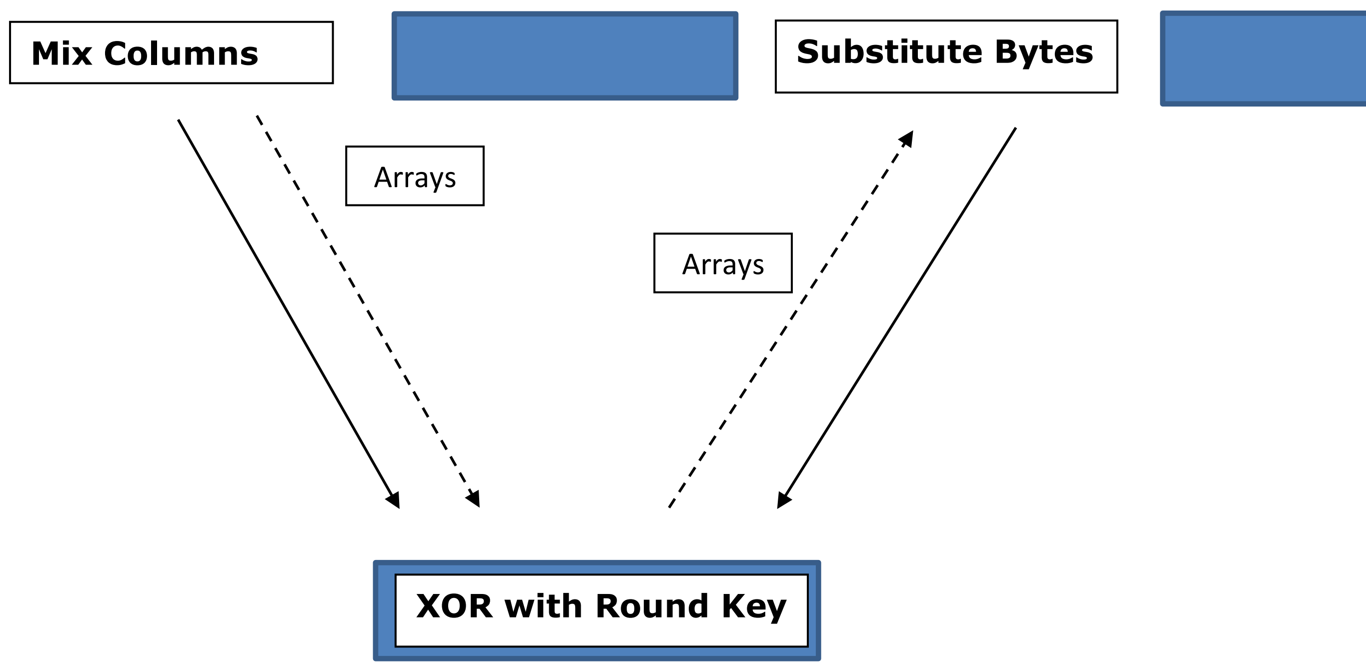
Coordinate Module: Main Module- It takes information from a subordinate module, optionally processes it and passes it on to another subordinate module.

Immediate subordinate modules of Main Module: Get Input, Encryption, Decryption. The following diagram expresses the idea of a coordinate module:



Dotted line represents data.

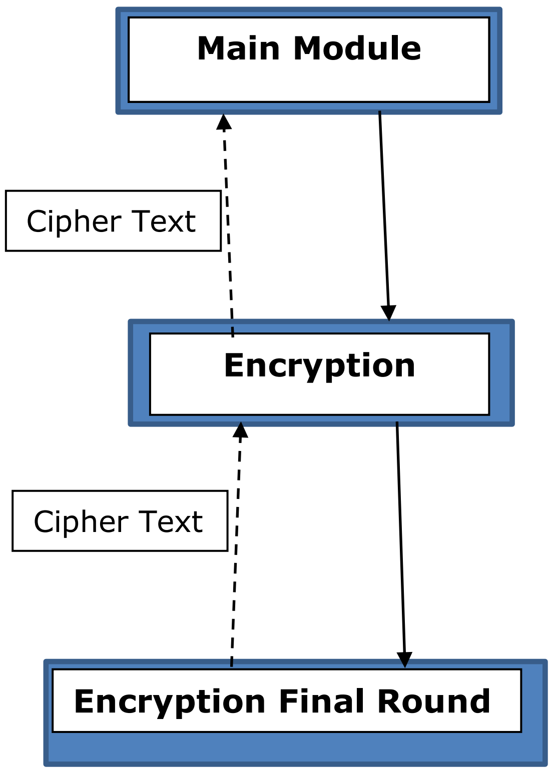
Transform module: Takes information from a superordinate module, optionally processes it and passes it on to another superordinate module. The module **XOR with Round Key** is an example of Transform module which aids the encryption process. There are other transform modules in the course of the entire project. The following diagram expresses the idea of a transform module:



Afferent module: It is a module that takes input from a subordinate module, optionally processes it and passes it to the superordinate module.

Let us consider an example. The module **Encryption** acts as an afferent module as it takes data from the subordinate module **Encryption Final Round,** optionally processes it and passes it on to the superordinate module, that is, the **Main Module**.

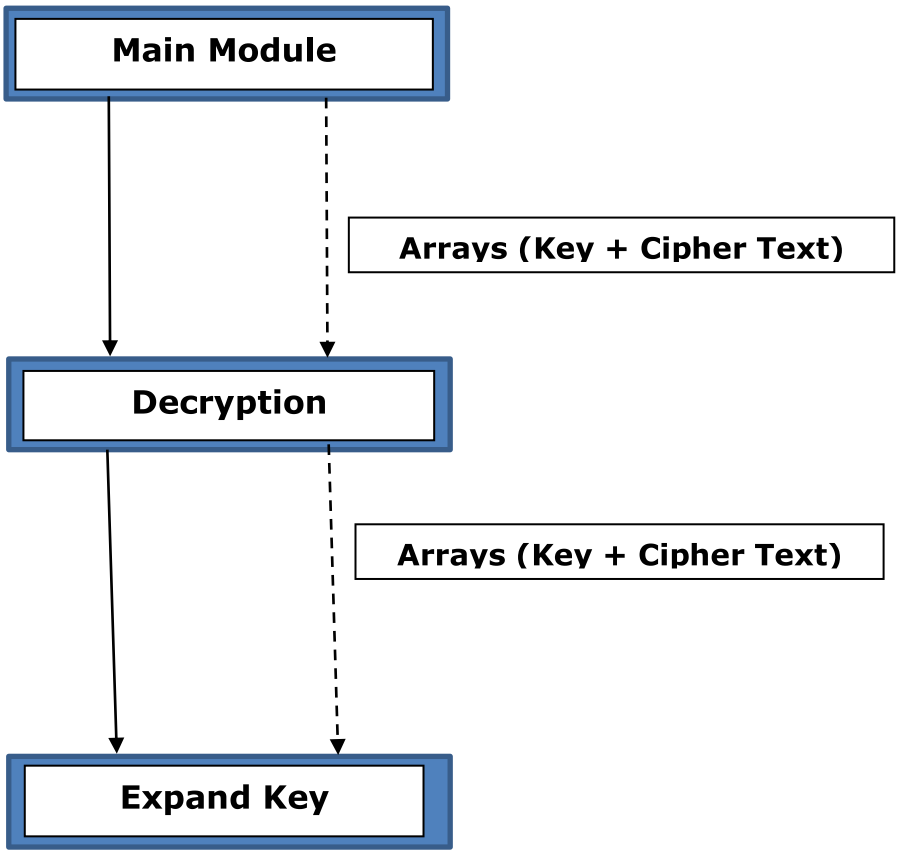
The following diagram expresses the idea of an afferent module:



Efferent module: It is a module that takes input from a superordinate module, optionally processes it and passes it on to the subordinate module. Let us consider an example.

The module **Decryption** acts as an efferent module as it takes data from the superordinate module **Main Module,** optionally processes it and passes it on to the subordinate module, that is, the **Expand Key**.

The following diagram expresses the idea of an afferent module:



It is to be noted that both the Encryption and the Decryption processes act as afferent module in one way and as efferent module the other way.

We have shown only one aspect as example. The conclusion is that the main module acts as the coordinator module, the Generate S-box module acts as the transform module and all other subordinate modules of Encryption and Decryption (as mentioned in the previous sections) are either afferent or efferent modules, depending on the data flow as shown in the DFDs (Figures 3 and 4). The Encryption module acts as the **efferent module** when it invokes its subordinate modules (**Expand Key**, for example) and it in turn is invoked by the **Main module**. The same Encryption module acts as **afferent module** when it takes data from the **Encryption Final Round** module and reports to the **Main module**. The same goes for the Decryption module.

Figures 3 and 4 depict the flow of data.

11.1.4 Design Decisions

The design decisions to be implemented are as follows:

- The programming language has been considered to be C.
- The interface has been considered as a command user interface.
- Modular design approach has been used.
- The entire system has been designed in a bottom-up fashion.
- The subordinate modules have been created independently and then linked with the other superordinate modules.
- C libraries appropriate for the working of the system, such as <stdio.h>, <stdlib.h>, <stdbool.h>, <stdint.h> have been included.
- Global variables have been made use of along with other formal parameters.
- Function calling techniques include call by value most of the times.
- Atomic functions (module) take care of the input-taking process and report to their superordinate modules.

11.2 Detailed Design (Logical Design)

The detailed design deals with the data structure of the individual modules and the algorithms involved in implementing them.

We shall consider some of the superordinate modules and their detailed design.

Main Module:

Data Structure:

Local variables of primitive data type: int, char

2-dimensional arrays of data type: uint8_t

The pre-requisites have been depicted in the pre/post conditions of the processes described in **Section 6.2**

```
Algorithm Main {

Declare Array input[][], key[][]: unsigned 8-bit integer (uint8_t in C)

Invoke Method s_box(sbox)

do {

Print (“ENTER E OR e TO ENCRYPT AND D or d TO DECRYPT”)

Input dec

If (dec=='E' OR dec=='e')

{
    PRINT ("Enter input of 128 bits \n")
                Input input[][]          // taking input
         PRINT ("Enter the key of 128 bits \n")
          Input key          //taking input of key
                              encryption(input,key,sbox)
             }
else if (dec=='D' OR dec=='d')
{
         PRINT ("Enter input of 128 bits \n");
             Input input[][]                              // taking input
                  PRINT ("Enter the key of 128 bits \n")
                           Input key                //taking input of key
                                    decryption(input,key,sbox,Invsbox)
                  }
else
                  PRINT ("\n PLEASE! ENTER THE CORRECT INPUT \n")
                  PRINT ("\n ENTER 0 TO CONITNUE \n")
                  Input con
         } while(con==0)
} // end of main
```

Module Get Input:

Data Structure: Primitive data types, 2-D Array inp[][], 1-D Array st[]

```
Algorithm get_input (inp[][])
{
    Declare Array  st[100]: character
     Input st
     k=0
    for(i=0;i<4;i++) {
         for(j=0;j<4;j++) {
                           inp[j][i]=hex_to_deci(st,k);
```

```
                    k=k+2;
                }
        }
} // end of Algorithm
```

Algorithm hex_to_deci (char* st,int k)

```
{
        Declare temp: unsigned 8-bit integer (uint8_t in C);
                if(st[k]-'0'>9)
                    {
                        temp=st[k+0]-'a';
                        temp+=10;
                    }
                    else
                        temp=st[k+0]-'0';
                    temp*=16;
                    if(st[k+1]-'0'>9)
                    {
                        temp+=st[k+1]-'a';
                        temp+=10;
                    }
                    else
                        temp+=st[k+1]-'0';
        return temp;
}
```

Module Encryption:

Data Structure: 2-D Arrays of type uint8_t: uint8_t input[4][4],uint8_t key[4][4],uint8_t sbox[16][16]

Algorithm encryption (uint8_t input[4][4],uint8_t key[4][4],uint8_t sbox[16][16]) {

```
extension_of_key(key,extended_key,sbox);
        Add_round_key(input,extended_key,0);
        printf("AFTER ADDING KEY :\t");
        print(input);
        for(k=1;k<10;k++)
        {
                PRINT ("ROUND %d \n\n",k)
                for(i=0;i<4;i++)
                        sub_bytes(input[i],sbox);
                PRINT ("AFTER SUBSTITUTION :\t")
                PRINT (input)
                shift_rows(input);
                PRINT ("AFTER SHIFTING ROWS :\t")
                PRINT (input)
                mix_cols(input);
                PRINT ("AFTER MIXING COLS :\t")
                PRINT (input)
                Add_round_key(input,extended_key,k);
                PRINT ("AFTER ADDING KEY :\t")
                PRINT (input)
        }
        PRINT ("\n ROUND %d \n",k)
        Execute Module Encryption Final Round
} // END OF Algorithm
```

Module Decryption:

Data Structure: 2-D Arrays of type uint8_t: uint8_t input[4][4], uint8_t key[4][4], uint8_t sbox[16][16], uint8_t Invsbox[16][16]

Algorithm decryption(uint8_t input[4][4],uint8_t key[4][4],uint8_t sbox[16][16],uint8_t Invsbox[16][16])

```
{
         k=0;
        PRINT ("INPUT IS :\t\t")
        PRINT (input)
        PRINT ("KEY IS :\t\t")
        PRINT (key)
        extension_of_key(key,extended_key,sbox);
        Add_round_key(input,extended_key,10);
        PRINT ("AFTER ADDING KEY :\t")
        PRINT (input)
        for(k=9;k>0;k--)
        {
            PRINT("\n ROUND %d \n\n",(10-k))
            Invshift_rows(input);
            PRINT ("AFTER SHIFTING ROWS :\t")
            PRINT (input)
            //substitution
            for(i=0;i<4;i++)
            {
                sub_bytes(input[i],Invsbox);
            }
            PRINT ("SUBSTITUTION OF BYTES :\t")
            PRINT (input)
            Add_round_key(input,extended_key,k);
            PRINT ("AFTER ADDING KEY :\t")
            PRINT (input)
            Invmix_cols(input);
            PRINT ("AFTER MIXING COLOUMNS :\t")
            PRINT (input)
        }
        PRINT ("\n ROUND 10 \n\n")
    Execute Module Decryption Final Round
    } // end of algorithm
```

We specify the design details of some of the subordinate modules that aid the modules encryption and decryption.

Module Expand Key:

Data Structure: 2-D Arrays of type uint8_t: uint8_t key[4][4],uint8_t extended_key[4][44],uint8_t sbox[16][16]

Algorithm:

Algorithm extension_of_key(uint8_t key[4][4],uint8_t extended_key[4][44],uint8_t sbox[16][16])

```
{
    Declare Array temp[4]: uint8_t, rcon
    i=0
    do {
        j=0
        do {
            extended_key[i][j]=key[i][j]
            j=j+1
        } while(j<4)
        i=i+1
    } while(i<4)
    i=4
    do {
        j=0
        do {
            temp[j]=extended_key[j][i-1]
            j=j+1
        } while(j<4)
        if (i MOD 4=0)
        {
            rcon=Rcon((i/4)-1)
            rotate_l(temp,1)
            sub_bytes(temp,sbox)
            temp[0]=temp[0] XOR rcon
        }
        j=0
        do {
            extended_key[j][i]=(extended_key[j][i-4] XOR temp[j])
        j=j+1
        } while(j<4)
    i=i+1
  } while(i<4)
} // end of algorithm
```

Module Add Round Key:

Data Structure: Primitive data type int and Arrays of type uint8_t: uint8_t inp[4][4],uint8_t key[4][44]

Algorithm:

```
Algorithm Add_round_key(uint8_t inp[4][4],uint8_t key[4][44],int r)
{
       LET i=0, j=0
      do {
            j=0
            do {
                  inp[j][i] XOR =key[j][(4*r)+i]
               j=j+1
            } while(j<4)
        i=i+1
       } while(i<4)
} //end of algorithm
```

Algorithm Rcon(int x)

```
{
  Declare temp: unsigned 8-bit integer (uint8_t in C)
  temp=01;
      if(x==0)
            return temp;
```

```
        for(i=1;i<=x;i++)
        {
                temp=multiply(temp,2);
        }
        return temp;
}
```

Module Substitute Bytes:

Data Structure: Arrays of type uint8_t: uint8_t inp[4], uint8_t box[16][16]

Algorithm:

```
Algorithm sub_bytes(uint8_t inp[4],uint8_t box[16][16])
{
     Declare i, x, y: integer
     i=0
        do {
                        y=inp[i] MOD 16
                        x=inp[i]/16
                        inp[i]=box[x][y]
                        i=i+1
        } while(i<4)
}
```

Module Inverse Substitute Bytes:

Data Structure: Arrays of type uint8_t: uint8_t inp[4], uint8_t box[16][16]

Algorithm:

```
Algorithm sub_bytes(uint8_t inp[4],uint8_t box[16][16])
{
     Declare i, x, y: unsigned 8-bit integer (uint8_t in C)
        i=0
        do {
                        y=inp[i] MOD 16
                        x=inp[i]/16
                        inp[i]=box[x][y]
```

```
                i=i+1
        } while(i<4)
}
```

Module Shift Rows:

Data Structure: 2-D Array of type uint8_t: input[][]

Algorithm:

```
Algorithm shift_rows(uint8_t input[4][4])
{
        Declare i, j, temp, k: integer
        i=1
        do {
                rotate_l(input[i],i)
                i=i+1
    } while(i<4)
}
```

Algorithm rotate_l(uint8_t word[4],int shift)

```
{
      Declare j: integer
        for(j=1;j<=shift;j++)
        {
                temp=word[0];
                for(i=0;i<3;i++)
                {
                        word[i]=word[i+1];
                }
                word[i]=temp;
        }
}
```

Module Inverse Shift Rows:

Data Structure: 2-D Array of type uint8_t: input[][]

Algorithm:

```
Algorithm Invshift_rows(uint8_t input[4][4])
{
    Declare i, k, j: integer
    Declare temp: unsigned 8-bit integer (uint8_t in C)
      k=1
      do {
      j=1
      do {
            temp=input[k][3]
            i=3
            do {
                  input[k][i]=input[k][i-1]
                  i=i-1
            } while(i>=1)
            input[k][0]=temp
         j=j+1
      } while(j<=k)
       k=k+1
      } while(k<=3)
}
```

Module Mix Columns:

Data Structure: 2-D Array of type uint8_t: input[][]

Algorithm:

```
Algorithm mix_cols(uint8_t input[4][4])
{
      Declare Array word[4]: unsigned 8-bit integer (uint8_t in C)
      i=0
      do {
            word[0]=multiply(input[0][i],2) XOR multiply(input[1][i],3) XOR
input[2][i] XOR input[3][i]
```

```
                word[1]=multiply(input[1][i],2) XOR multiply(input[2][i],3) XOR
input[3][i] XOR input[0][i]
                word[2]=multiply(input[2][i],2) XOR multiply(input[3][i],3) XOR
input[0][i] XOR input[1][i]
word[3]=multiply(input[3][i],2) XOR multiply(input[0][i],3) XOR input[1][i] XOR
input[2][i]
                j=0
                  do {
                        input[j][i]=word[j]
                        j=j+1
                  } while(j<4)
        i=i+1
    } while(i<4)
}
```

Module Inverse Mix Columns:

Data Structure: 2-D Array of type uint8_t: input[][]

Algorithm:

```
Algorithm Invmix_cols(uint8_t input[4][4])
{
        Declare Array word[4]: unsigned 8-bit integer (uint8_t in C)
        i=0
        do {
                word[0]=multiply(input[0][i],14) XOR multiply(input[1][i],11) XOR
multiply(input[2][i],13) XOR multiply(input[3][i],9)
                word[1]=multiply(input[1][i],14) XOR multiply(input[2][i],11) XOR
multiply(input[3][i],13) XOR multiply(input[0][i],9)
                word[2]=multiply(input[2][i],14) XOR multiply(input[3][i],11) XOR
multiply(input[0][i],13) XOR multiply(input[1][i],9)
                word[3]=multiply(input[3][i],14) XOR multiply(input[0][i],11) XOR
multiply(input[1][i],13) XOR multiply(input[2][i],9)
                j=0
               do {
                        input[j][i]=word[j]
                         j=j+1
                } while(j<4)
        i=i+1
       } while(i<4)
} //end of algorithm
```

Module: Encryption Final Round

Data Structure: 2-D Arrays of type uint8_t: uint8_t input[4][4],uint8_t key[4][4],uint8_t sbox[16][16]

```
Algorithm Encryption Final Round {
    Declare i: integer
     for(i=0;i<4;i++)
            sub_bytes(input[i],sbox);
      PRINT ("AFTER SUBSTITUTION :\t")
      PRINT (input)
      shift_rows(input);
      PRINT ("AFTER SHIFTING ROWS :\t")
      PRINT (input)
      Add_round_key(input,extended_key,10);
      PRINT ("ENCRYPTED OUTPUT IS :\t")
      PRINT (input)
}
```

Module: Decryption Final Round

Data Structure: 2-D Arrays of type uint8_t: uint8_t input[4][4], uint8_t key[4][4], uint8_t sbox[16][16], uint8_t Invsbox[16][16]

```
Algorithm Decryption Final Round {
PRINT ("\n ROUND 10 \n\n")
      Invshift_rows(input);
      PRINT ("AFTER SHIFTING ROWS :\t");
      PRINT (input)
      for(i=0;i<4;i++)
```

```
                sub_bytes(input[i],Invsbox);
        PRINT ("AFTER SUBSTITUTION :\t")
        PRINT (input)
        Add_round_key(input,extended_key,0);
        PRINT ("AFTER ADDING KEY :\t")
        PRINT (input)
        PRINT ("DECRYPTED OUTPUT IS :\t")
        PRINT (input)
} // end of algorithm
```

11.3 Objectives of Software Design

- **Correctness:** The design satisfies the requirements specified in SRS.
- **Verifiable:** The correctness is verifiable.
- **Traceable:** The design is traceable to the requirements.
- **Completeness:** All specifications have been implemented.
- **Consistency:** Design is inherently consistent.
- **Understandable:** Design has interconnection and modifying dependencies.
- **Efficiency:** There is a proper utilisation of resources.
- **Maintainable:** The design has been implemented by modular approach. Hence there is scope for easy incorporation of changes. It is the most important aspect of software design.

12. Testing

Software testing can be stated as the process of verifying and validating that a software or application is bug free, meets the technical requirements as guided by its design and development and meets the user requirements effectively and efficiently with handling all the exceptional and boundary cases.

The process of software testing aims not only at finding faults in the existing software but also at finding measures to improve the software in terms of efficiency, accuracy and usability. It mainly aims at measuring specification, functionality and performance of a software program or application.

Software level testing can be majorly classified into 4 levels:

1. **Unit Testing:** A level of the software testing process where individual units/components of a software/system are tested. The purpose is to validate that each unit of the software performs as designed.
2. **Integration Testing:** A level of the software testing process where individual units are combined and tested as a group. The purpose of this level of testing is to expose faults in the interaction between integrated units.

3. **System Testing:** A level of the software testing process where a complete, integrated system/software is tested. The purpose of this test is to evaluate the system's compliance with the specified requirements.
4. **Acceptance Testing:** A level of the software testing process where a system is tested for acceptability. The purpose of this test is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

There are two testing techniques: Black box testing and glass-box testing.

1. **Black Box Testing:** The technique of testing in which the tester doesn't have access to the source code of the software and is conducted at the software interface without concerning with the internal logical structure of the software is known as black box testing.
2. **Glass-Box Testing:** The technique of testing in which the tester is aware of the internal workings of the product, have access to its source code and is conducted by making sure that all internal operations are performed according to the specifications is known as glass box testing.

Software testing can also be classified into two steps:

Verification and **Validation**

1. **Verification:** It refers to the set of tasks that ensure that software correctly implements a specific function.
2. **Validation:** it refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Verification: "Are we building the product right?"
Validation: "Are we building the right product?"

12.1 Structural Testing

Structural testing comprises three parts namely UNIT TSTING, INTEGRATED TESTING AND SYSTEM TESTING that are explained below:

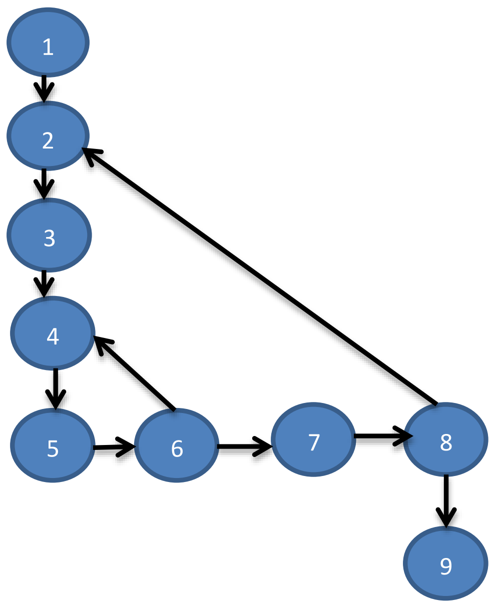
12.1.1 Unit Testing

Algorithm get_input (inp[][])

```
{
   Declare Array  st[100]: character
1  {    Input st
       k=0
2       do{
3      j=0
```

```
4     do{

              5 { inp[j][i]=hex_to_deci(st,k);
                { k=k+2; j=j+1;

              }while(j<4)  6

7   i=i+1

}while(i<4)  8

} // end of Algorithm  9
```

Control flow graph



No of bounded regions: 2

No of edges: 10

No of vertices: 9

Cyclomatic complexity: V(G)= E– V+2 = 3

V(G)= 2+1 = 3

Basis set of linearly independent path:

Path 1: 1->2->3->4->5->6->7->8->9

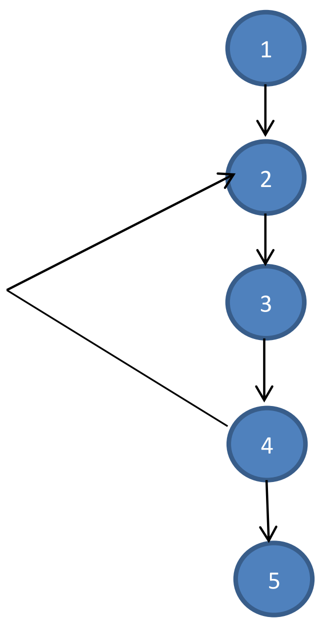
Path 2: 1->2->3->4->5->6->7->8->2

Path 3: 1->2->3->4->5->6->4

Module: Substitute Bytes

```
Algorithm sub_bytes(uint8_t inp[4],uint8_t box[16][16])
{
1   Declare i, x, y: integer
    i=0
2       do {
                    y=inp[i] MOD 16
            3       x=inp[i]/16
                    inp[i]=box[x][y]
                    i=i+1
        }while(i<4)     4
}// end of algorithm    5
```

Control flow graph



No. of bounded regions: 1

Cyclomatic complexity:

V(G) = 1+1= 2

Basis set of linearly independent path:

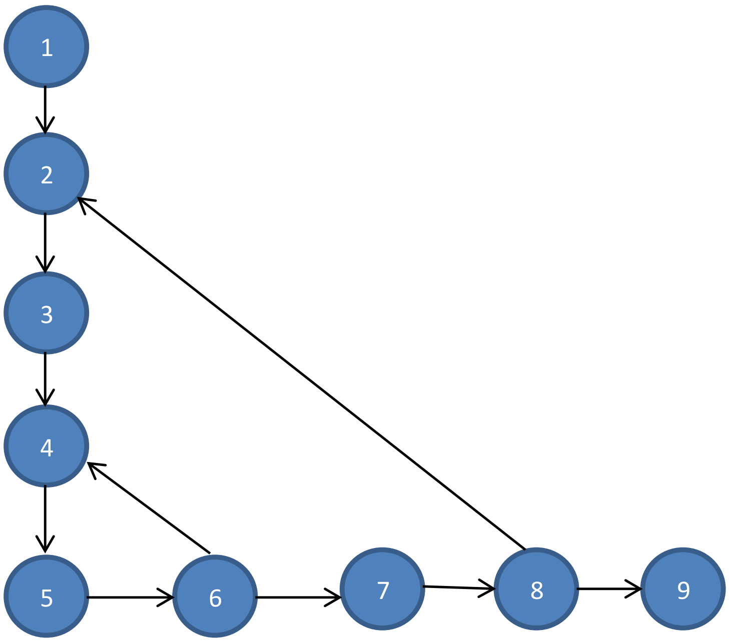
Path 1: 1->2->3->4->5

Path 2: 1->2 ->3 ->4->2

Module: Add Round Key

```
Algorithm Add_round_key(uint8_t inp[4][4],uint8_t key[4][44],int r)
{
1       LET i=0, j=0
2       do {
            3       j=0
            4       do {
                        5 { inp[j][i] XOR =key[j][(4*r)+i]
                          { j=j+1
                    } while(j<4)        6
                i=i+1    7
        } while(i<4)      8
}end of function      9
```

Control flow Graph



No of bounded regions: 2

No of edges: 10

No of vertices: 9

Cyclomatic complexity: V(G)= E– V+2 = 3

V(G)= 2+1 = 3

Basis set of linearly independent path:

Path 1: 1->2->3->4->5->6->7->8->9

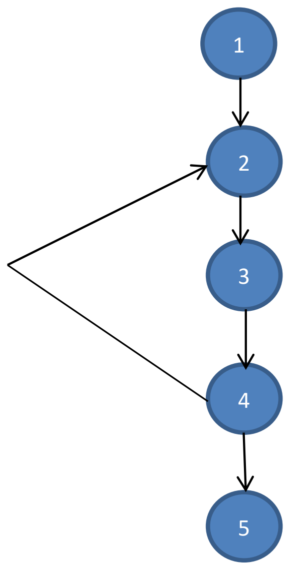
Path 2: 1->2->3->4->5->6->7->8->2

Path 3: 1->2->3->4->5->6->4

Module: Inverse Substitution Bytes

```
Algorithm sub_bytes(uint8_t inp[4],uint8_t box[16][16])
{
1   Declare i, x, y: integer
    i=0
2       do {
                    y=inp[i] MOD 16
          3         x=inp[i]/16
                    inp[i]=box[x][y]
                    i=i+1
        }while(i<4)      4
}// end of algorithm   5
```

Control flow graph



No. of bounded regions: 1

Cyclomatic complexity:

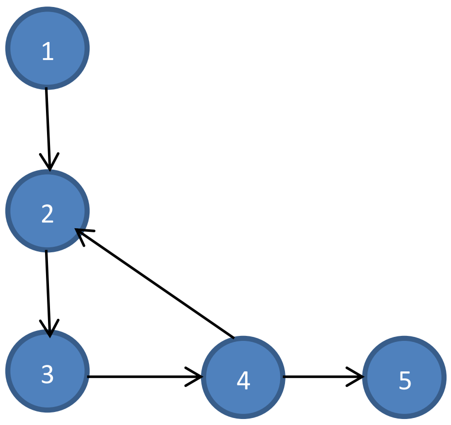
$$V(G) = 1+1= \ 2$$

Module: Shift Rows

```
Algorithm shift_rows(input[][])
{
1       Declare i, j, temp, k: integer
2       i=1
3         do {
        4    rotate_l(input[i],i)
             i=i+1
     } while(i<4)        5
```

}//end of function

Control flow graph



Number of bounded regions: 1

No. of edges: 5

No. of vertices: 5

Cyclomatic complexity: E- V + 2 = V(G)= 2

Basis set of linearly independent paths:

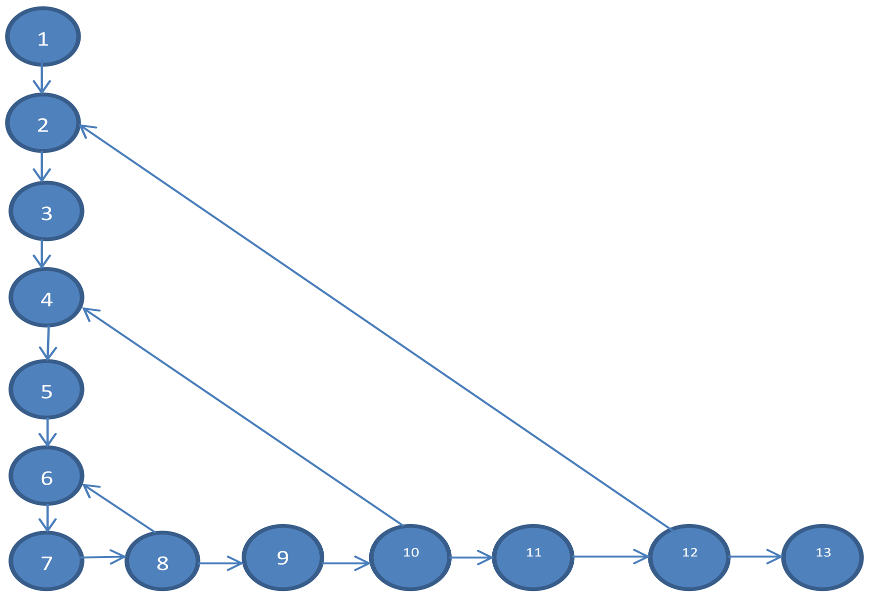
Path 1: 1->2->3->4->5

Path 2: 1->2->3->4->2

Module: Inverse Shift Rows

```
Algorithm Invshift_rows(input[][])
{
1     Declare i, k, j: integer
      Declare temp: unsigned 8-bit integer (uint8_t in C)
      k=1
2     do {
3     j=1
4       do {
      5   temp=input[k][3]
          i=3
      6       do {
              7   input[k][i]=input[k][i-1]
                  i=i-1
              } while(i>=1)       8
              input[k][0]=temp
      9        j=j+1
      } while(j<=k)       10
       k=k+1       11
      } while(k<=3)       12
}//end of function  13
```

control flow graph



No. of bounded regions: 3

No. of edges: 15

No. of vertices: 13

Cyclomatic complexity: E-V+2 = 15-13+2= 4

Basis of linearly independent paths:

Path 1: 1->2->3->4->5->6->7->6

Path 2: 1->2->3->4->5->6->7->8->9->10->4

Path 3: 1->2->3->4->5->6->7->8->9->10->11->12->2

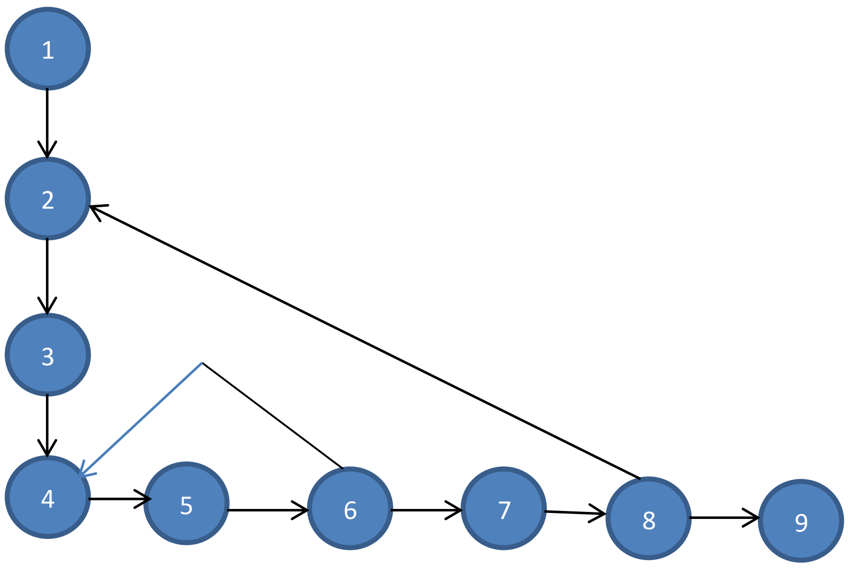
Path 4: 1->2->3->4->5->6->7->8->9->10->11->12->13

Module: Mix Columns

```
Algorithm mix_cols(input[][])
{
1     Declare Array word[4]: unsigned 8-bit integer (uint8_t in C)
      i=0
2     do {
          word[0]=multiply(input[0][i],2) XOR multiply(input[1][i],3) XOR
          input[2][i] XOR input[3][i]
          word[1]=multiply(input[1][i],2) XOR multiply(input[2][i],3) XOR
3         input[3][i] XOR input[0][i]
          word[2]=multiply(input[2][i],2) XOR multiply(input[3][i],3) XOR
          input[0][i] XOR input[1][i]
          word[3]=multiply(input[3][i],2) XOR multiply(input[0][i],3) XOR
          input[1][i] XOR input[2][i]
          j=0
      4   do {
              input[j][i]=word[j]
      5       j=j+1
          } while(j<4)     6
      i=i+1 7
    } while(i<4)    8
}//end of function    9
```

NO

Control flow graph



No. of bounded regions: 2

No. of edges: 10

No. of vertices: 9

Cyclomatic complexity is: E – V+2 = 10-9+2 = 3

Basis of linearly independent paths:

Path 1: 1->2->3->4->5->6->4

Path 2: 1->2->3->4->5->6->7->8->2

Path 3: 1->2->3->4->5->6->7->8->9

Module: Inverse Mix Columns

Algorithm Invmix_cols(uint8_t input[4][4])

{

1 $\left\{\begin{array}{l}\text{Declare Array word[4]: unsigned 8-bit integer (uint8\_t in C)}\\ \text{i=0}\end{array}\right.$

```
2    do {
3        word[0]=multiply(input[0][i],14) XOR multiply(input[1][i],11) XOR
         multiply(input[2][i],13) XOR multiply(input[3][i],9)
         word[1]=multiply(input[1][i],14) XOR multiply(input[2][i],11) XOR
         multiply(input[3][i],13) XOR multiply(input[0][i],9)
         word[2]=multiply(input[2][i],14) XOR multiply(input[3][i],11) XOR
         multiply(input[0][i],13) XOR multiply(input[1][i],9)
         word[3]=multiply(input[3][i],14) XOR multiply(input[0][i],11) XOR
         multiply(input[1][i],13) XOR multiply(input[2][i],9)
         j=0
4            do {
5                input[j][i]=word[j]
                 j=j+1
             } while(j<4)        6
         i=i+1        7
     } while(i<4)        8
} //end of algorithm        9
```

No. of bounded regions: 2

No. of edges = 10

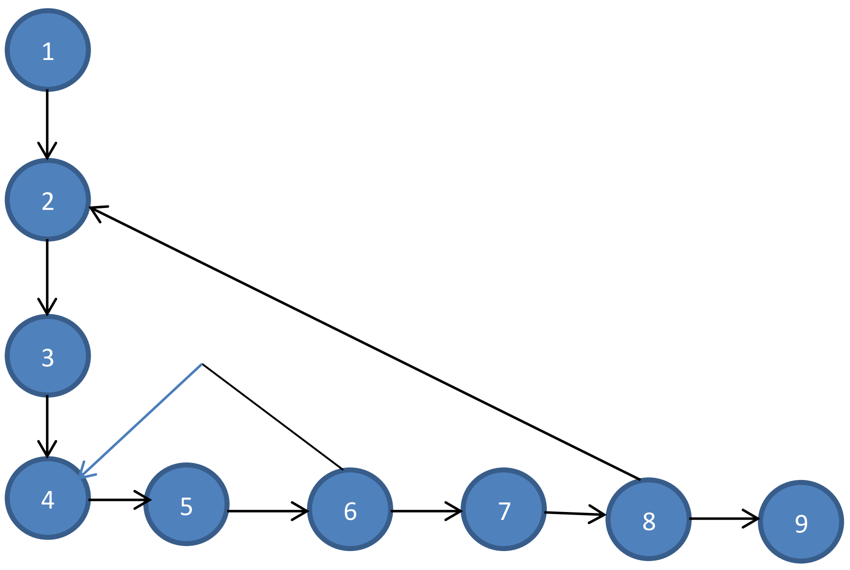
No. of vertices = 9

Cyclomatic complexity= E – V + 2

= 10- 9+2

= 3

Control flow graph



Basis set of linearly independent path:

Path 1: 1->2->3->4->5->6->4

Path 2: 1->2->3->4->5->6->7->8->2

Path 3: 1->2->3->4->5->6->7->8->9

12.1.2 Integration Testing

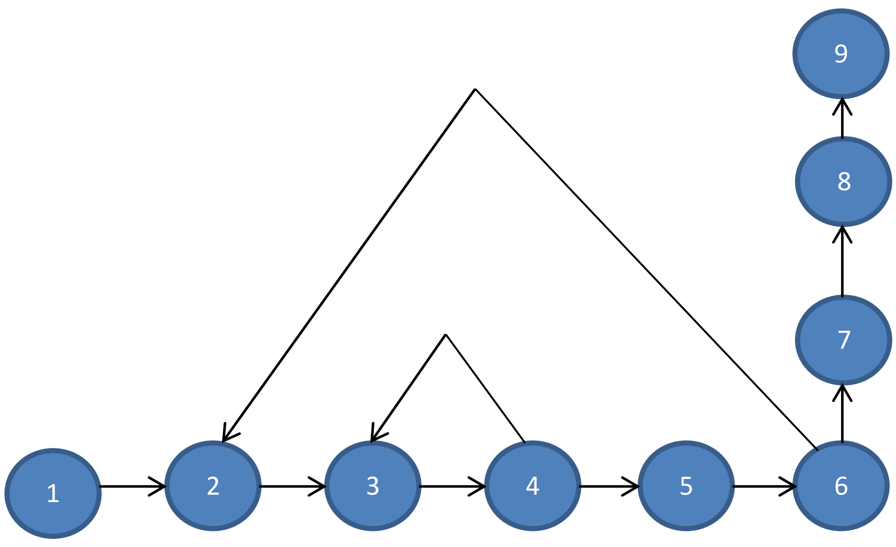
Module Encryption:

Algorithm encryption (input[][], key[][], sbox[][]) {

```
     extension_of_key(key,extended_key,sbox);
1    Add_round_key(input,extended_key,0);
     printf("AFTER ADDING KEY :\t");
     print(input);

2    for(k=1;k<10;k++)
     {
          PRINT ("ROUND %d \n\n",k)
     3    for(i=0;i<4;i++)
```

```
        4  sub_bytes(input[i],sbox);
           } end of for  5
           PRINT ("AFTER SUBSTITUTION :\t")
           PRINT (input)
           shift_rows(input);
           PRINT ("AFTER SHIFTING ROWS :\t")
    6      PRINT (input)
           mix_cols(input);
           PRINT ("AFTER MIXING COLS :\t")
           PRINT (input)
           Add_round_key(input,extended_key,k);
           PRINT ("AFTER ADDING KEY :\t")
           PRINT (input)
       }end of for   7
    8  PRINT ("\n ROUND %d \n",k)
       Execute Module Encryption Final Round
    } // END OF Algorithm     9
```

Control flow graph



No. of bounded regions: 2

No. of edges = 9

No. of vertices = 8

Cyclomatic complexity= E – V + 2

= 9- 8+2

= 3

Basis of linearly independent Path:

Path 1: 1->2->3->4->3

Path 2: 1->2->3->4->5->6->2

Path 3: 1->2->3->4->6->7->8->9

Test cases are:

1: cipher text 50 67 246 168 136 90 48 141 49 49 152 162 224 55 7 52

Key: 43 126 21 22 40 174 210 166 171 247 21 136 9 207 79 60

Module: Decryption

```
Algorithm decryption(input[][],  key[][], sbox[][], Invsbox[][])
{
        k=0;
        PRINT ("INPUT IS :\t\t")
        PRINT (input)
1       PRINT ("KEY IS :\t\t")
        PRINT (key)
        extension_of_key(key,extended_key,sbox);
        Add_round_key(input,extended_key,10);
        PRINT ("AFTER ADDING KEY :\t")
        PRINT (input)

2       for(k=9;k>0;k--)
        {
                PRINT("\n ROUND %d \n\n",(10-k))
3               Invshift_rows(input);
                PRINT ("AFTER SHIFTING ROWS :\t")
                PRINT (input)
                //substitution
        4       for(i=0;i<4;i++)
                {
                5       sub_bytes(input[i],Invsbox);
                }end of for   6
                PRINT ("SUBSTITUTION OF BYTES :\t")
        7       PRINT (input)
                Add_round_key(input,extended_key,k);
                PRINT ("AFTER ADDING KEY :\t")
                PRINT (input)
                Invmix_cols(input);
                PRINT ("AFTER MIXING COLOUMNS :\t")
                PRINT (input)
            }end of for   8
9           PRINT ("\n ROUND 10 \n\n")
            Execute Module Decryption Final Round
    } // end of algorithm       10
```

No. of bounded regions: 2

No. of edges = 10

No. of vertices = 9

Cyclomatic complexity= E – V + 2

= 10- 9+2

= 3

Basis of linearly independent Path:

Path 1: 1->2->3->4->3

Path 2: 1->2->3->4->5->6->2

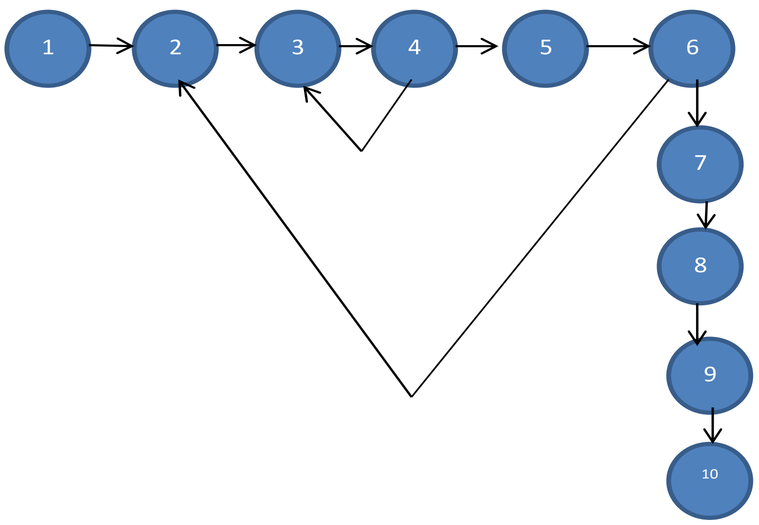
Path 3: 1->2->3->4->6->7->8->9->10

Test cases are:

1.plain text: 0 17 34 51 68 85 102 119 136 153 170 187 204 221 238 255

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

Control flow graph:



12.1.3 System testing:

Module: Main module

```
Algorithm Main {
Declare Array input[][], key[][]: unsigned 8-bit integer (uint8_t in C)      1
Invoke Method s_box(sbox)
do {          2
Print (“ENTER E OR e TO ENCRYPT AND D or d TO DECRYPT”)          3
Input dec

        4            5
If (dec=='E' OR dec=='e')
{
            PRINT ("Enter input of 128 bits \n")
            Input input[][]      // taking input                  6
            PRINT ("Enter the key of 128 bits \n")
            Input key        //taking input of key
            encryption(input,key,sbox)
        }   end if
        7                  8
else if (dec=='D' OR dec=='d')
{
        PRINT ("Enter input of 128 bits \n");
        Input input[][]                                           9
        PRINT ("Enter the key of 128 bits \n")
        Input key
        decryption(input,key,sbox,Invsbox)
}
else
        PRINT ("\n PLEASE! ENTER THE CORRECT INPUT \n")    10
            PRINT ("\n ENTER 0 TO CONITNUE \n")
            Input con                                          11
        } while(con==0)    12
} // end of main           13


No. of bounded regions:   6
```

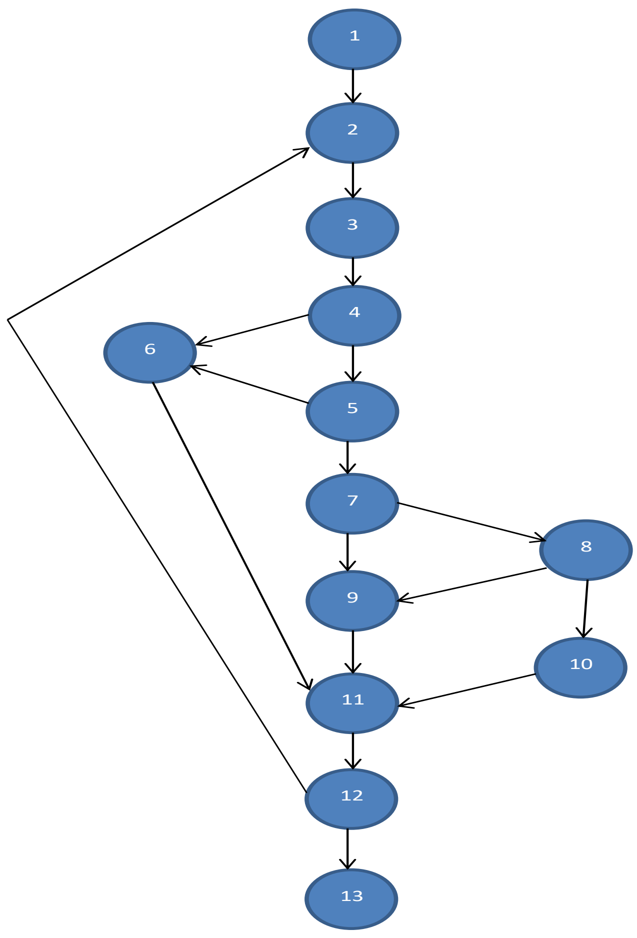
No. of edges: 17

No. of vertices: 13

Cyclomatic complexity: E- V+ 2

17-13+2 = 6

Control flow graph



Basis set of linearly independent paths:

Path 1: 1->2->3->4->6

Path 2: 1->2->3->4->5->6

Path 3: 1->2->3->4->5->6->11->12->13

Path 4:	1->2->3->4->5->7->9->11->2
Path 5:	1->2->3->4->5->7->8->9->11->12->13
Path 6:	1->2->3->4->5->7->8->10->11->12->13

Test cases are:

1. <inp = E >
2. <inp=e>
3. <inp = D>
4. <inp = d>
5. <inp p or any alphabet other than above>

12.2 Functional Testing

12.2.1 Cause Effect Graphic Technique

Module: Encryption

Causes:

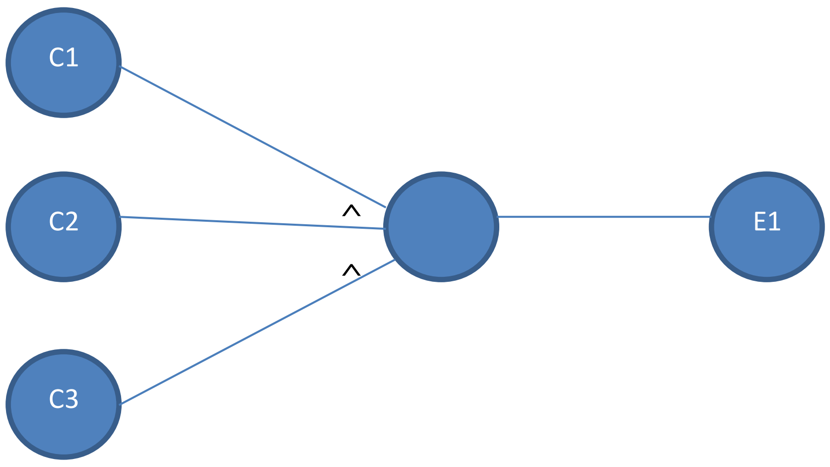
C1: Input plain text

C2: Extended key

C3: Substitution box

Effects:

E1: Cipher text



Design Table:

C1 **C2** **C3**	**1** **1** **1**
E1	**1**

Module: Decryption

Causes:

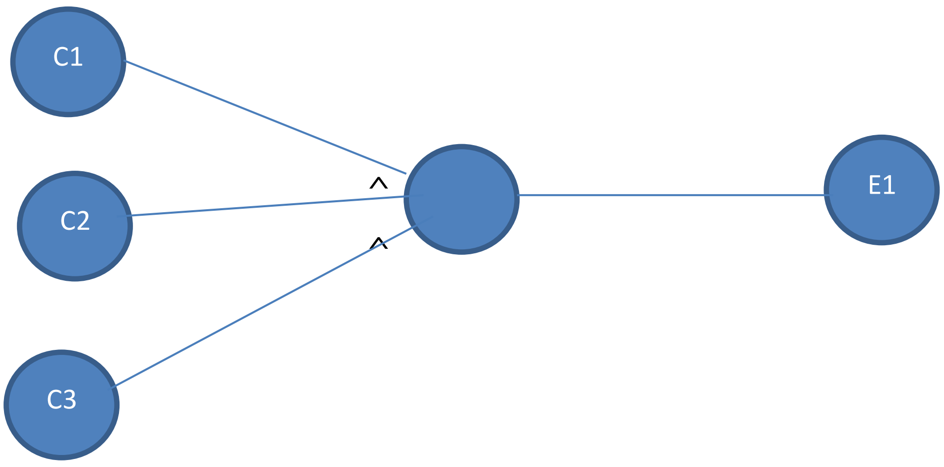
C1: Input cypher text

C2: Extended key

C3: Substitution box

Effects:

E1: decipher text or plain text



Design Table:

C1 **C2** **C3**	**1** **1** **1**
E1	**1**

Module: Main Module

Causes:

C1: Input E

C2: Input e

C3: Input D

C4: Input d

C5: Input Other than above

Effects:

E1: Encryption

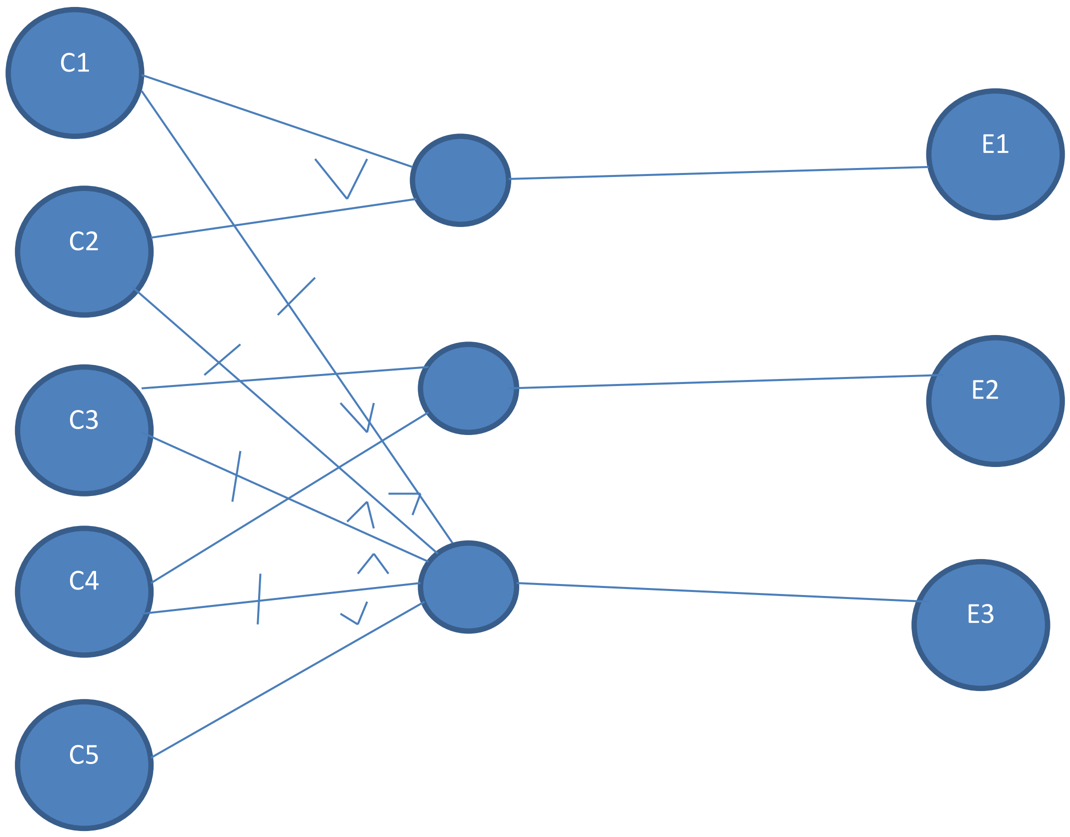
E2: Decryption

E3: Error Message

Design Table:

	1	**2**	**3**	**4**	**5**

C1	1	0	0	0	0
C2	0	1	0	0	0
C3	0	0	1	0	0
C4	0	0	0	1	0
C5	0	0	0	0	1
E1	**1**	**1**			
E2			**1**	**1**	
E3					**1**



---THE END-------------------------------------