

# **DATA REDUCTION BY HUFFMAN CODING AND ENCRYPTION BY INSERTION OF SHUFFLED CYCLIC REDUNDANCY CODE**

**A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF**

**Bachelor of Technology  
in  
Electronics & Instrumentation Engineering**

**By  
NILKESH PATRA  
And  
SILA SIBA SANKAR**



**Department of Electronics & Communication Engineering  
National Institute of Technology  
Rourkela  
2007**

# **DATA REDUCTION BY HUFFMAN CODING AND ENCRYPTION BY INSERTION OF SHUFFLED CYCLIC REDUNDANCY CODE**

**A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS  
FOR THE DEGREE OF  
Bachelor of Technology**

**in**

**Electronics & Instrumentation Engineering**

**By**

**NILKESH PATRA**

**And**

**SILA SIBA SANKAR**

**Under the Guidance of**

**Prof. G.S.Rath**



**Department of Electronics & Communication Engineering  
National Institute of Technology**

**Rourkela**

**2007**



**National Institute of Technology  
Rourkela**

**CERTIFICATE**

This is to certify that the thesis entitled, “Data reduction by huffman coding and encryption by insertion of shuffled cyclic redundancy code” submitted by Sri Nilkesh Patra and Sri Sila Siba Sankar in partial fulfillments for the requirements for the award of Bachelor of Technology Degree in Electronics & Instrumentation Engineering at National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by him under my supervision and guidance.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University / Institute for the award of any Degree or Diploma.

Date:

Prof. G. S. Rath  
Dept. of Electronics & Instrumentation Engg  
National Institute of Technology  
Rourkela - 769008

## **ACKNOWLEDGEMENT**

We would like to articulate our deep gratitude to our project guide Prof. G.S.Rath who has always been our motivation for carrying out the project. It is our pleasure to refer Microsoft Word exclusive of which the compilation of this report would have been impossible. An assemblage of this nature could never have been attempted with out reference to and inspiration from the works of others whose details are mentioned in reference section. We acknowledge our indebtedness to all of them. Last but not the least , our sincere thanks to all of our friends who have patiently extended all sorts of help for accomplishing this undertaking.

NILKESH PATRA

SILA SIBA SANKAR

# CONTENTS

	Page No	
<i>Abstract</i>	ii	
<b>Chapter 1</b>	<b>GENERAL INTRODUCTION</b>	1
<b>Chapter 2</b>	<b>CONVENTIONAL DATA COMPRESSION</b>	3
2.1	Introduction	4
2.2	Lossless vs. lossy compression	5
2.3	Compression algorithms	6
<b>Chapter 3</b>	<b>CONVENTIONAL DATA ENCRYPTION</b>	10
3.1	Introduction	11
3.2	Why encrypting?	12
3.3	How encryption works?	13
3.4	Private & public key encryption	13
3.5	Encryption algorithms	16
<b>Chapter 4</b>	<b>CODING THEORY &amp; CRC CODE</b>	19
4.1	Introduction	20
4.2	Source coding	20
4.3	Channel encoding	21
4.4	CRC code	22
<b>Chapter 5</b>	<b>ENCRYPTION BY SHUFFLED CRC CODE</b>	25
5.1	Introduction	26
5.2	Data compression by Huffman coding	26
5.3	Insertion of shuffled CRC code	28
<b>Chapter 6</b>	<b>EXPERIMENTATION AND RESULTS</b>	29
6.1	Source code	30
6.2	Results	39
6.3	Conclusion	41
	<b>REFERENCES</b>	42

## ABSTRACT

### Introduction

Cryptography today is assumed as the study of techniques and applications of securing the integrity and authenticity of transfer of information under difficult circumstances. It uses mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication, and data origin authentication. The encryption here mainly of two types. They are private and public key cryptography.

### Steps involved

#### 1. Huffman compression

In information theory, Huffman coding is an entropy encoding algorithm used for lossless data compression. The term refers to the use of a variable length code table for encoding a source symbol (such as a character in a file) where the variable-length code table has been derived in a particular way based on the estimated probability of occurrence for each possible value of the source symbol

#### 2. Encryption by shuffled CRC

After the text was compressed, CRC is used to detect the error of the code word. A word is generated using the generator polynomial, which is used to encrypt the compressed code. Then the CRC will be inserted into the code in a random fashion.

### Experimental work

The above compression and encryption methods have been employed by using high level language MATLAB. The Huffman compression and shuffled CRC encryption operations have been implemented using this language.

# Chapter 1

## GENERAL INTRODUCTION

Data compression is known for reducing storage and communication costs. It involves transforming data of a given format, called source message, to data of a smaller sized format, called codeword. Data encryption is known for protecting information from eavesdropping. It transforms data of a given format, called plaintext, to another format, called cipher text, using an encryption key. The major problem existing with the current compression and encryption methods is the large amount of processing time required by the computer to perform the tasks. To lessen the problem, I combine the two processes into one. To combine the two processes, I introduce the idea of adding a pseudo random shuffle into a data compression process. The method of using a pseudo random number generator to create a pseudo random shuffle is well known. A simple algorithm as below can do the trick. Assume that we have a list  $(x_1, x_2, \dots, x_n)$  and that we want to shuffle it randomly.

```
for i = n downto 2
```

```
{  
    k = random(1,i);  
    swap  $x_i$  and  $x_k$ ;  
}
```

Since we are adding pseudo random shuffles into data compression processes, understanding all three compression algorithms used is critical in the understanding of our algorithms. Even though our algorithms are based on random shuffles, our algorithms don't merely re-ordering data. Unlike substitution ciphers, our algorithms don't encrypt plaintext by simply replacing a piece of data with another equal sized data. Unlike transposition cipher, our algorithms don't encrypt plaintext by just playing anagrams. As I will explain in detail, simultaneous data compression and encryption offers an effective remedy for the execution time problem in data security. These methods can easily be utilized in multimedia applications, which are lacking in security and speed.

# Chapter 2

## CONVENTIONAL DATA COMPRESSION

## 2.1 Introduction

Compression is used just about everywhere. All the images you get on the web are compressed, typically in the JPEG or GIF formats, most modems use compression, HDTV will be compressed using MPEG-2, and several file systems automatically compress files when stored, and the rest of us do it by hand. The neat thing about compression, as with the other topics we will cover in this course, is that the algorithms used in the real world make heavy use of a wide set of algorithmic tools, including sorting, hash tables, tries, and FFTs. Furthermore, algorithms with strong theoretical foundations play a critical role in real-world applications.

In this chapter we will use the generic term message for the objects we want to compress, which could be either files or messages. The task of compression consists of two components, an encoding algorithm that takes a message and generates a “compressed” representation (hopefully with fewer bits), and a decoding algorithm that reconstructs the original message or some approximation of it from the compressed representation. These two components are typically intricately tied together since they both have to understand the shared compressed representation.

We distinguish between lossless algorithms, which can reconstruct the original message exactly from the compressed message, and lossy algorithms, which can only reconstruct an approximation of the original message. Lossless algorithms are typically used for text, and lossy for images and sound where a little bit of loss in resolution is often undetectable, or at least acceptable. Lossy is used in an abstract sense, however, and does not mean random lost pixels, but instead means loss of a quantity such as a frequency component, or perhaps loss of noise. For example, one might think that lossy text compression would be unacceptable because they are imagining missing or switched characters. Consider instead a system that reworded sentences into a more standard form, or replaced words with synonyms so that the file can be better compressed. Technically the compression would be lossy since the text has changed, but the “meaning” and clarity of the message might be fully maintained, or even improved. In fact Strunk and White might argue that good writing is the art of lossy text compression.

Is there a lossless algorithm that can compress all messages? There has been at least one patent application that claimed to be able to compress all files (messages)—Patent 5,533,051 titled “Methods for Data Compression”. The patent application claimed that if it was applied recursively, a file could be reduced to almost nothing. With a little thought you should convince yourself that this is not possible, at least if the source messages can contain any bit-sequence. We can see this by a simple counting argument. Lets consider all 1000 bit messages, as an example.

There are different messages we can send, each which needs to be distinctly identified by the decoder. It should be clear we can't represent that many different messages by sending 999 or fewer bits for all the messages — 999 bits would only allow us to send distinct messages. The truth is that if any one message is shortened by an algorithm, then some other message needs to be lengthened. You can verify this in practice by running GZIP on a GIF file. It is, in fact, possible to go further and show that for a set of input messages of fixed length, if one message is compressed, then the average length of the compressed messages over all possible inputs is always going to be longer than the original input messages. Consider, for example, the 8 possible 3 bit messages. If one is compressed to two bits, it is not hard to convince yourself that two messages will have to expand to 4 bits, giving an average of  $31/8$  bits. Unfortunately, the patent was granted.

Because one can't hope to compress everything, all compression algorithms must assume that there is some bias on the input messages so that some inputs are more likely than others, i.e. that there is some unbalanced probability distribution over the possible messages. Most compression algorithms base this “bias” on the structure of the messages – i.e., an assumption that repeated characters are more likely than random characters, or that large white patches occur in “typical” images. Compression is therefore all about probability.

## **2.2 Lossless vs. lossy compression**

Lossless compression algorithms usually exploit statistical redundancy in such a way as to represent the sender's data more concisely, but nevertheless perfectly. Lossless compression is possible because most real-world data has statistical redundancy. For example, in English text, the letter 'e' is much more common than the letter 'z', and the probability that the letter 'q' will be followed by the letter 'z' is very small.

Another kind of compression, called lossy data compression, is possible if some loss of fidelity is acceptable. For example, a person viewing a picture or television video scene might not notice if some of its finest details are removed or not represented perfectly (i.e. may not even notice compression artifacts). Similarly, two clips of audio may be perceived as the same to a listener even though one is missing details found in the other. Lossy data compression algorithms introduce relatively minor differences and represent the picture, video, or audio using fewer bits.

Lossless compression schemes are reversible so that the original data can be reconstructed, while lossy schemes accept some loss of data in order to achieve higher compression.

However, lossless data compression algorithms will always fail to compress some files; indeed, any compression algorithm will necessarily fail to compress any data containing no discernible patterns. Attempts to compress data that has been compressed already will therefore usually result in an expansion, as will attempts to compress encrypted data.

In practice, lossy data compression will also come to a point where compressing again does not work, although an extremely lossy algorithm, which for example always removes the last byte of a file, will always compress a file up to the point where it is empty.

A good example of lossless vs. lossy compression is the following string -- 888883333333. What you just saw was the string written in an uncompressed form. However, you could save space by writing it 8[5]3[7]. By saying "5 eights, 7 threes", you still have the original string, just written in a smaller form. In a lossy system, using 83 instead, you cannot get the original data back (at the benefit of a smaller file size).

## **2.3 Compression algorithms**

As mentioned in the introduction, coding is the job of taking probabilities for messages and generating bit strings based on these probabilities. In practice we typically use probabilities for parts of a larger message rather than for the complete message, e.g., each character or word in a text. To be consistent with the terminology in the previous section, we will consider each of these components a message on its own, and we will use the term message sequence for the larger message made up of these components. In general each little message can be of a different type and come from its own probability distribution. For example, when sending an image we might send a message specifying a color followed by messages specifying a frequency component of that color. Even the messages specifying the color might come from different probability distributions since the probability of particular colors might depend on the context. We distinguish between algorithms that assign a unique code (bit-string) for each message, and ones that "blend" the codes together from more than one message in a row. In the first class we will consider Huffman codes, which are a type of prefix code. In the later category we consider arithmetic codes. The arithmetic codes can achieve better compression, but can require the

encoder to delay sending messages since the messages need to be combined before they can be sent.

### 2.3.1 prefix codes

A code  $C$  for a message set is a mapping from each message to a bit string. Each bit string is called codeword. A code called a codeword, and we will denote codes using the syntax  $C = \{(S_1, W_1), (S_2, W_2), \dots, (S_M, W_M)\}$ . Typically in computer science we deal with fixed length codes, such as the ASCII code which maps every printable character and some control characters into 7 bits. For compression, however, we would like code words that can vary in length based on the probability of the message. Such variable length codes have the potential problem that if we are sending one codeword after the other it can be hard or impossible to tell where one codeword finishes and the next starts. For example- given the code  $\{(a,1), (b,01), (c,101), (d,011)\}$ , the bit-sequence 1011 could either be decoded as aba, ca, or ad. To avoid this ambiguity we could add a special stop symbol to the end of each codeword (e.g., a 2 in a 3-valued alphabet), or send a length before each symbol. These solutions, however, require sending extra data. A more efficient solution is to design codes in which we can always uniquely decipher a bit sequence into its code words. We will call such uniquely decodable codes **uniquely decodable codes**. A prefix code is a special kind of uniquely decodable code in which no bit-string is a prefix of another one, for example  $\{(a,1), (b,01), (c,101), (d,011)\}$  1 01 000 001 . All prefix codes are uniquely decodable since once we get a match, there is no longer code that can also match.

### 2.3.2 Huffman Codes

Huffman codes are optimal prefix codes generated from a set of probabilities by a particular algorithm, the Huffman Coding Algorithm. David Huffman developed the algorithm as a student in a 12 class on information theory at MIT in 1950. The algorithm is now probably the most prevalently used component of compression algorithms, used as the back end of GZIP, JPEG and many other utilities.

The Huffman algorithm is very simple and is most easily described in terms of how it generates the prefix-code tree.

1. Start with a forest of trees, one for each message. Each tree contains a single vertex with weight  $W_i = P_i$

2. Repeat until only a single tree remains

- Select two trees with the lowest weight roots ( $W_1$  and  $W_2$ ).
- Combine them into a single tree by adding a new root with weight  $W_1 + W_2 = C$ , and making the two trees its children. It does not matter which is the left or right child, but our convention will be to put the lower weight root on the left if  $W_1 \neq W_2$ .

For a code of size  $n$  this algorithm will require  $n-1$  steps since every complete binary tree with  $n$  leaves has  $n-1$  internal nodes, and each step creates one internal node. If we use a priority queue with  $O(\log n)$  time insertions and find-mins (*e.g.*, a heap) the algorithm will run in  $O(n \log n)$  time.

The key property of Huffman codes is that they generate optimal prefix codes. We show this in the following theorem, originally given by Huffman.

### 2.3.3 Arithmetic Coding

Arithmetic coding is a technique for coding that allows the information from the messages in a message sequence to be combined to share the same bits. The technique allows the total number of bits sent to asymptotically approach the sum of the self information of the individual messages (recall that the self information of a message is defined as  $\log_2(1/p_i)$ )

To see the significance of this, consider sending a thousand messages each having probability 0.999. Using a Huffman code, each message has to take at least 1 bit, requiring 1000 bits to be sent.

On the other hand the self information of each message is  $\log_2(1/p_i) = 0.00144$  bits, so the sum of this self-information over 1000 messages is only 1.4 bits. It turns out that arithmetic coding will send all the messages using only 3 bits, a factor of hundreds fewer than a Huffman coder. Of course this is an extreme case, and when all the probabilities are small, the gain will be less significant. Arithmetic coders are therefore most useful when there are large probabilities in the probability distribution.

The main idea of arithmetic coding is to represent each possible sequence of messages by a separate interval on the number line between 0 and 1, *e.g.* the interval from .2 to .5. For a sequence of messages with probabilities  $p_1, p_2, \dots, p_n$ , the algorithm will assign the sequence to an interval of size  $\prod_{i=1}^n p_i$ , by starting with an interval of size 1 (from 0 to 1) and narrowing the interval by a factor of  $p_i$  on each message  $i$ . We can bound the number of bits required to

uniquely identify an interval of size  $s$ , and use this to relate the length of the representation to the self information of the messages.

In the following discussion we assume the decoder knows when a message sequence is complete either by knowing the length of the message sequence or by including a special end-of-file message. This was also implicitly assumed when sending a sequence of messages with Huffman codes since the decoder still needs to know when a message sequence is over.

We will denote the probability distributions of a message set as  $\{p(1), p(2), \dots, p(m)\}$ , and we define the *accumulated probability* for the probability distribution as

$$f(j) = \sum_{i=1}^{j-1} p(i) \quad (j= 1, 2, \dots, m)$$

Arithmetic coding assigns an interval to a sequence of messages using the following recurrences

$$l_i = \begin{cases} f_i & i = 1 \\ l_{i-1} + f_i * s_{i-1} & 1 < i \leq n \end{cases}$$

$$s_i = \begin{cases} p_i & i = 1 \\ s_{i-1} * p_i & 1 < i \leq n \end{cases}$$

Where  $l_n$  is the lower bound of the interval and  $s_n$  is the size of the interval, *i.e.* the interval is given by  $[l_n, l_n + s_n]$ . We assume the interval is inclusive of the lower bound, but exclusive of the upper bound. The recurrence narrows the interval on each step to some part of the previous interval. Since the interval starts in the range  $[0, 1)$ , it always stays within this range. An important property of the intervals generated by the above Equation is that all unique message sequences of length  $n$  have non overlapping intervals. Specifying an interval therefore uniquely determines the message sequence. In fact, any number within an interval uniquely determines the message sequence. The job of decoding is basically the same as encoding but instead of using the message value to narrow the interval, we use the interval to select the message value, and then narrow it. We can therefore “send” a message sequence by specifying a number within the corresponding interval.

# Chapter 3

## CONVENTIONAL DATA ENCRYPTION

### **3.1 Introduction**

Security and privacy have long been important issues forming the basis of numerous democracies around the world. In the digital age, securing personal information and ensuring privacy pose to be issues of paramount concern. At first glance, one might find it gratifying that an online website greets the person by their first name, sends them emails when goods of their taste are added, or recommends goods services based on their demographic profile, previous visits, etc. An astute surfer though will also see the privacy drawbacks in such services. Who else is being provided this information? Is there a way to ensure the security of this information? What happens with the information if the company meets financial difficulties and has to liquidate its assets? Where does all that "private information" go?

Many studies over the last few years have suggested that a majority of consumers are concerned about when, what and how their personal information is being collected, how this information is being used and whether it is being protected. They want to know whether the information is being sold or shared with others, and if so with whom and for what purposes. They also want to have control over their privacy in today's digital age where strides in telecommunication, storage and software technologies have made monitoring a person's activities effortless.

The Internet, once a research tool has grown into a mammoth educational, entertainment and commercial implementation. The advent of commerce on the Internet exposed the lack of security over this public network. The incorporation of encryption (especially strong 128 bit encryption) into Internet browsers and web servers quelled this concern to a certain extent. There was still the matter of storing the information sent over the Internet in a safe manner. Firewalls and encryption software evolved to ensure that the computers and data on the Internet were safer.

What can be done regarding these important issues? Part of the solution is to secure important data - more specifically, using strong encryption. Educating end users and corporations on the use of email and file encryption software, data encryption during transmission using VPNs, password encryption on public interfaces and use of encryption software like PGP, F-Secure and 128 bit version of IE/NS will lead us closer to the end goal of a safer Internet.

The growth of the worldwide Internet user base and with Internet based transactions believed to reach well over a trillion dollars in the next three years, it makes sense for the parties involved to secure the Internet. Haphazard handling of financial and personal information can

lead to the Internet being constantly associated with fraud and privacy abuses instead of being a viable commerce medium.

### **3.2 Why encrypting?**

As organizations and individuals have connected to the Internet in droves, many have begun eyeing its infrastructure as an inexpensive medium for wide-area and remote connections. The Internet is an international network consisting of individual computers and computer networks that are all interconnected by many paths. Unlike Local Area Networks where access is physically restricted to authorized users, the Internet is a public network and can be accessed by anyone. Now more than ever, moving vast amounts of information quickly and safely across great distances is one of our most pressing needs. The basic idea of cryptography is to hide information from prying eyes. On the Internet this can be your credit card numbers, bank account information, health/social security information, or personal correspondence with someone else.

#### **History of Encryption**

Encryption pre-dates the Internet by thousands of years. Looking back in history we find that Julius Caesar was an early user of cryptography. He sent messages to his troops in a simple but ingenious method. A letter in the alphabet was replaced by one say 5 positions to the right. So, an "A" would be replaced by an "E", "B" by "F" and so on. Hence **RETURN** would become **VJYZVS**. But as it can be seen, this cipher can be easily broken by either figuring out a pattern, by brute force or by getting ones hands on a plaintext and cipher text combination to deduce the pattern.

#### **Users of Encryption**

A few decades ago, only governments and diplomats used encryption to secure sensitive information. Today, secure encryption on the Internet is the key to confidence for people wanting to protect their privacy, or doing business online. E-Commerce, secure messaging, and virtual private networks are just some of the applications that rely on encryption to ensure the safety of data. In many companies that have proprietary or sensitive information, field personnel are required to encrypt their entire laptops fearing that in the wrong hands this information could cause millions of dollars in damage.

### 3.3 How encryption works?

The concept behind encryption is quite simple - make the data illegible for everyone else except those specified. This is done using cryptography - the study of sending 'messages' in a secret form so that only those authorized to receive the 'message' be able to read it.

The easy part of encryption is applying a mathematical function to the plaintext and converting it to an encrypted cipher. The harder part is to ensure that the people who are supposed to decipher this message can do so with ease, yet only those authorized are able to decipher it. We of-course also have to establish the legitimacy of the mathematical function used to make sure that it is sufficiently complex and mathematically sound to give us a high degree of safety.

The essential concept underlying all automated and computer security application is cryptography. The two ways of going about this process are conventional (or symmetric) encryption and public key (or asymmetric) encryption.

#### 3.4.1 Private key encryption

Private Key encryption also referred to as conventional, single-key or *symmetric encryption* was the only available option prior to the advent of Public Key encryption in 1976. This form of encryption has been used throughout history by Julius Caesar, the Navaho Indians, and German U-Boat commanders to present day military, government and private sector applications. It enquires all parties that are communicating to share a common key.

A conventional encryption scheme has five major parts:

*Plaintext* - this is the text message to which an algorithm is applied.

*Encryption Algorithm* - it performs mathematical operations to conduct substitutions and transformations to the plaintext.

*Secret Key* - This is the input for the algorithm as the key dictates the encrypted outcome.

*Cipher text* - This is the encrypted or scrambled message produced by applying the algorithm to the plaintext message using the secret key.

*Decryption Algorithm* - This is the encryption algorithm in reverse. It uses the ciphertext, and the secret key to derive the plaintext message.

When using this form of encryption, it is essential that the sender and receiver have a way to exchange secret keys in a secure manner. If someone knows the secret key and can figure out the algorithm, communications will be insecure. There is also the need for a strong encryption

algorithm. What this means is that if someone were to have a ciphertext and a corresponding plaintext message, they would be unable to determine the encryption algorithm.

There are two methods of breaking conventional/symmetric encryption - brute force and cryptanalysis. Brute force is just as it sounds; using a method (computer) to find all possible combinations and eventually determine the plaintext message. Cryptanalysis is a form of attack that attacks the characteristics of the algorithm to deduce a specific plaintext or the key used. One would then be able to figure out the plaintext for all past and future messages that continue to use this compromised setup.

### **3.4.2 Public key encryption**

1976 saw the introduction of a radical new idea into the field of cryptography. This idea centered around the premise of making the encryption and decryption keys different - where the knowledge of one key would not allow a person to find out the other. Public key encryption algorithms are based on the premise that each sender and recipient has a private key, known only to him/her and a public key, which can be known by anyone. Each encryption/decryption process requires at least one public key and one private key. A key is a randomly generated set of numbers/ characters that is used to encrypt/decrypt information.

A public key encryption scheme has six major parts:

*Plaintext* - this is the text message to which an algorithm is applied.

*Encryption Algorithm* - it performs mathematical operations to conduct substitutions and transformations to the plaintext.

*Public and Private Keys* - these are a pair of keys where one is used for encryption and the other for decryption.

*Ciphertext* - this is the encrypted or scrambled message produced by applying the algorithm to the plaintext message using key.

*Decryption Algorithm* - This algorithm generates the ciphertext and the matching key to produce the plaintext.

## Selecting the Public and Private Keys

1. Select large prime numbers  $p$  and  $q$  and form  $n = pq$ .
2. Select an integer  $e > 1$  such that  $\text{GCD}(e, (p - 1)(q - 1)) = 1$ .
3. Solve the congruence,  $ed \equiv 1 \pmod{(p - 1)(q - 1)}$  for an integer  $d$  where  $1 < d < (p - 1)(q - 1)$ .
4. The public encryption key is  $(e, n)$ .
5. The private encryption key is  $(d, n)$ .

## The Encryption Process

- The process of encryption begins by converting the text to a pre hash code. This code is generated using a mathematical formula.
- This pre hash code is encrypted by the software using the senders private key. The private key would be generated using the algorithm used by the software.
- The encrypted pre hash code and the message are encrypted again using the sender's private key.
- The next step is for the sender of the message to retrieve the public key of the person this information is intended for.
- The sender encrypts the secret key with the recipient's public key, so only the recipient can decrypt it with his/her private key, thus concluding the encryption process.

1. Lookup the user's public key  $(e, n)$ .
2. Make sure that the message  $M$  is an integer such that  $0 \leq M \leq n$ .
3. Compute,  $M^e \pmod n$  where  $0 \leq C \leq n$ .
4. Transmit the integer  $C$ .

## The Decryption Process

- The recipient uses his/her private key to decrypt the secret key.

- The recipient uses their private key along with the secret key to decipher the encrypted pre hash code and the encrypted message.
- The recipient then retrieves the sender's public key. This public key is used to decrypt the pre hash code and to verify the sender's identity.
- The recipient generates a post hash code from the message. If the post hash code equals the pre hash code, then this verifies that the message has not been changed en-route.

Use your private key  $(d, n)$ .

1. Receive the integer  $C$ , where  $0 \leq C \leq n$ .
2. Compute,  $C^d \pmod n$  where  $0 \leq R \leq n$ .
3.  $R$  is the original message.

### 3.5 Encryption algorithms

Different encryption algorithms use proprietary methods of generating these keys and are therefore useful for different applications. Here are some nitty gritty details about some of these encryption algorithms. Strong encryption is often discerned by the key length used by the algorithm.

#### RSA

In 1977, shortly after the idea of a public key system was proposed, three mathematicians, Ron Rivest, Adi Shamir and Len Adleman gave a concrete example of how such a method could be implemented. To honour them, the method was referred to as the RSA Scheme. The system uses a private and a public key. To start two large prime numbers are selected and then multiplied together;  $n=p*q$ .

If we let  $f(n) = (p-1)(q-1)$ , and  $e > 1$  such that  $GCD(e, f(n))=1$ . Here  $e$  will have a fairly large probability of being co-prime to  $f(n)$ , if  $n$  is large enough and  $e$  will be part of the encryption key. If we solve the Linear Diophantine equation;  $ed \text{ congruent } 1 \pmod{f(n)}$ , for  $d$ . The pair of integers  $(e, n)$  are the public key and  $(d, n)$  form the private key. Encryption of  $M$  can be accomplished by the following expression;  $Me = qn + C$  where  $0 \leq C < n$ . Decryption

would be the inverse of the encryption and could be expressed as;  $Cd \text{ congruent } R \pmod{n}$  where  $0 \leq R < n$ . RSA is the most popular method for public key encryption and digital signatures today.

## **DES/3DES**

The Data Encryption Standard (DES) was developed and endorsed by the U.S. government in 1977 as an official standard and forms the basis not only for the Automatic Teller Machines (ATM) PIN authentication but a variant is also utilized in UNIX password encryption. DES is a block cipher with 64-bit block size that uses 56-bit keys. Due to recent advances in computer technology, some experts no longer consider DES secure against all attacks; since then Triple-DES (3DES) has emerged as a stronger method. Using standard DES encryption, Triple-DES encrypts data three times and uses a different key for at least one of the three passes giving it a cumulative key size of 112-168 bits.

## **BLOWFISH**

Blowfish is a symmetric block cipher just like DES or IDEA. It takes a variable-length key, from 32 to 448 bits, making it ideal for both domestic and exportable use. Bruce Schneier designed Blowfish in 1993 as a fast, free alternative to the then existing encryption algorithms. Since then Blowfish has been analyzed considerably, and is gaining acceptance as a strong encryption algorithm.

## **IDEA**

International Data Encryption Algorithm (IDEA) is an algorithm that was developed by Dr. X. Lai and Prof. J. Massey in Switzerland in the early 1990s to replace the DES standard. It uses the same key for encryption and decryption, like DES operating on 8 bytes at a time. Unlike DES though it uses a 128 bit key. This key length makes it impossible to break by simply trying every key, and no other means of attack is known. It is a fast algorithm, and has also been implemented in hardware chipsets, making it even faster.

## **SEAL**

Rogaway and Coppersmith designed the Software-optimized Encryption Algorithm (SEAL) in 1993. It is a Stream-Cipher, i.e., data to be encrypted is continuously encrypted. Stream Ciphers are much faster than block ciphers (Blowfish, IDEA, DES) but have a longer initialization phase during which a large set of tables is done using the Secure Hash Algorithm. SEAL uses a 160 bit key for encryption and is considered very safe.

## **RC4**

RC4 is a cipher invented by Ron Rivest, co-inventor of the RSA Scheme. It is used in a number of commercial systems like Lotus Notes and Netscape. It is a cipher with a key size of up to 2048 bits (256 bytes), which on the brief examination given it over the past year or so seems to be a relatively fast and strong cypher. It creates a stream of random bytes and 'XORing' those bytes with the text. It is useful in situations in which a new key can be chosen for each message.

# Chapter 4

**CODING THEORY & CRC CODE**

## 4.1 Introduction

Coding theory is a branch of mathematics and computer science dealing with the error-prone process of transmitting data across noisy channels, via clever means, so that a large number of errors that occur can be corrected. It also deals with the properties of codes, and thus with their fitness for a specific application.

There are two classes of codes.

1. Source coding (Data compression).
2. Channel coding (Forward error correction).

The first, source encoding, attempts to compress the data from a source in order to transmit it more efficiently. We see this practice every day on the Internet where the common "Zip" data compression is used to reduce the network load and make files smaller. The second, channel encoding adds extra data bits, commonly called redundancy bits, to make the transmission of data more robust to disturbances present on the transmission channel. The ordinary user may not be aware of many applications using channel coding. A typical music CD uses a powerful Reed-Solomon code to correct for scratches and dust. In this application the transmission channel is the CD itself. Cell phones also use powerful coding techniques to correct for the fading and noise of high frequency radio transmission. Data modems, telephone transmissions, and of course NASA all employ powerful channel coding to get the bits through.

A cyclic redundancy check (CRC) is a type of hash function, which is used to produce a small, fixed-size checksum of a larger block of data, such as a packet of network traffic or a computer file. The checksum is used to detect errors after transmission or storage. A CRC is computed and appended before transmission or storage, and verified afterwards by the recipient to confirm that no changes occurred in transit. CRCs are popular because they are simple to implement in binary hardware, are easy to analyze mathematically, and are particularly good at detecting common errors caused by noise in transmission channels.

## 4.2 Source coding

Entropy of a source is the measure of information. Basically source codes try to reduce the redundancy present in the source, and represent the source with a fewer bits that carry more information.

Data compression which explicitly tries to minimize the entropy of messages according to a particular probability model is called entropy encoding.

Various techniques used by source coding schemes try to achieve the limit of Entropy of the source.  $C(x) \geq H(x)$ , where  $H(x)$  is entropy of source (bit rate), and  $C(x)$  is the bit rate after compression. In particular, no source coding scheme can be better than the entropy limit of the symbol. Example: Facsimile transmission uses a simple run length code.

### **4.3 Channel encoding**

The aim of channel encoding theory is to find codes which transmit quickly, contain many valid code words and can correct or at least detect many errors. These aims are mutually exclusive however, so different codes are optimal for different applications. The needed properties of this code mainly depend on the probability of errors happening during transmission. In a typical CD, the impairment is mainly dust or scratches. Thus codes are used in an interleaved manner. The data is spread out over the disk. Although not a very good code, a simple repeat code can serve as an understandable example. Suppose we take a block of data bits (representing sound) and send it three times. At the receiver we will examine the three repetitions bit by bit and take a majority vote. The twist on this is that we don't merely send the bits in order. We interleave them. The block of data bits is first divided into 4 smaller blocks. Then we cycle through the block and send one bit from the first, then the second, etc. This is done three times to spread the data out over the surface of the disk. In the context of the simple repeat code, this may not appear effective. However, there are more powerful codes known which are very effective at correcting the "burst" error of a scratch or a dust spot when this interleaving technique is used.

Other codes are more appropriate for different applications. Deep space communications are limited by the thermal noise of the receiver which is more of a continuous nature than a bursty nature. Likewise, narrowband modems are limited by the noise present in the telephone network and is also modeled better as a continuous disturbance. Cell phones are troubled by rapid fading. The high frequencies used can cause rapid fading of the signal even if the receiver is moved a few inches. Again there are a class of channel codes that are designed to combat fading.

The term algebraic coding theory denotes the sub-field of coding theory where the properties of codes are expressed in algebraic terms and then further researched. Algebraic Coding theory, is basically divided into two major types of codes

1. Linear block codes
2. Convolutional codes

It analyzes the following three properties of a code -- mainly:

- code word length
- total number of valid code words
- the minimum Hamming distance between two valid code words

#### 4.4 CRC Code

A CRC "checksum" is the remainder of a binary division with no bit carry (XOR used instead of subtraction), of the message bit stream, by a predefined (short) bit stream of length  $n + 1$ , which represents the coefficients of a polynomial with degree  $n$ . Before the division,  $n$  zeros are appended to the message stream.

CRCs are based on division in the ring of polynomials over the finite field GF (2) (the integers modulo 2). In simpler terms, this is the set of polynomials where each coefficient is either zero or one (a single binary bit), and arithmetic operations wrap around. For example:

$$(x^3 + x) + (x + 1) = x^3 + 2x + 1 \equiv x^3 + 1 \pmod{2}$$

Note that  $2x$  becomes zero in the above equation because addition of coefficients is performed modulo 2:

$$2x = x + x = x \times (1 + 1) = x \times 0 \equiv 0 \pmod{2}$$

Multiplication is similar:

$$(x^2 + x)(x + 1) = x^3 + 2x^2 + x \equiv x^3 + x \pmod{2}$$

We can also divide polynomials mod 2 and find the quotient and remainder. For example, suppose we're dividing  $x^3 + x^2 + x$  by  $x + 1$ . We would find that

$$\frac{(x^3 + x^2 + x)}{(x + 1)} = (x^2 + 1) - \frac{1}{(x + 1)}$$

In other words,

$$(x^3 + x^2 + x) = (x^2 + 1)(x + 1) - 1 \equiv (x^2 + 1)(x + 1) + 1 \pmod{2}$$

The division yields a quotient of  $x^2 + 1$  with a remainder of -1, which, since it is odd, has a last bit of 1.

Any string of bits can be interpreted as the coefficients of a message polynomial of this sort, and to find the CRC, we multiply the message polynomial by  $x^n$  and then find the remainder when dividing by the degree- $n$  generator polynomial. The coefficients of the remainder polynomial are the bits of the CRC.

In the above equations,  $x^2 + x + 1$  represents the original message bits 111,  $x + 1$  is the generator polynomial, and the remainder 1 (equivalently,  $x^0$ ) is the CRC. The degree of the generator polynomial is 1, so we first multiplied the message by  $x^1$  to get  $x^3 + x^2 + x$ .

In general form:

$$M(x) \cdot x^n = Q(x) \cdot G(x) + R(x)$$

Here  $M(x)$  is the original message polynomial and  $G(x)$  is the degree- $n$  generator polynomial. The bits of  $M(x) \cdot x^n$  are the original message with  $n$  zeros added at the end.  $R(x)$  is the remainder polynomial, which is the CRC 'checksum'. The quotient polynomial  $Q(x)$  is uninteresting. In communication, the sender attaches the  $n$  bits of  $R$  after the original message bits of  $M$  and sends them out (in place of the zeros). The receiver takes  $M$  and  $R$  and checks whether  $M(x) \cdot x^n - R(x)$  is divisible by  $G(x)$ . If it is, then the receiver assumes the received message bits are correct. Note that  $M(x) \cdot x^n - R(x)$  is exactly the string of bits the sender sent; this string is called the *codeword*.

A CRC is a checksum in a strict mathematical sense, as it can be expressed as the weighted modulo-2 sum of per-bit syndromes, but that word is generally reserved more specifically for sums computed using larger moduli, such as 10, 256, or 65535.

CRCs can also be used as part of error-correcting codes, which allow not only the detection of transmission errors, but the reconstruction of the correct message. These codes are based on closely related mathematical principles.

# Chapter 5

## ENCRYPTION BY SHUFFLED CRC CODE

## 5.1 Introduction

In cryptography, **encryption** is the process of obscuring information to make it unreadable without special knowledge, sometimes referred to as scrambling. Encryption has been used to protect communications for centuries, but only organizations and individuals with extraordinary privacy and/or secrecy requirements had bothered to exert the effort required to implement it. In the mid-1970s, strong encryption emerged from the preserve of secretive government agencies into the public domain, and is now used in protecting many kinds of systems, such as the Internet e-commerce, mobile telephone networks and bank automatic teller machines.

Encryption can be used to ensure secrecy and/or privacy, but other techniques are still needed to make communications secure, particularly to verify the integrity and authenticity of a message; for example, a message authentication code (MAC) or digital signatures. Another consideration is protection against traffic analysis.

Encryption or software code obfuscation is also used in software copy protection against reverse engineering, unauthorized application analysis, cracks and software piracy used in different encryption or obfuscating software.

## 5.2 Data compression by Huffman coding

The technique works by creating a binary tree of nodes. These can be stored in a regular array, the size of which depends on the number of symbols ( $N$ ). A node can be either a leaf node or an internal node. Initially, all nodes are leaf nodes, which contain the symbol itself, the weight (frequency of appearance) of the symbol and optionally, a link to a parent node which makes it easy to read the code (in reverse) starting from a leaf node. Internal nodes contain symbol weight, links to two child nodes and the optional link to a parent node. As a common convention, bit '0' represents following the left child and bit '1' represents following the right child. A finished tree has  $N$  leaf nodes and  $N-1$  internal nodes.

A linear-time method to create a Huffman tree is to use two queues, the first one containing the initial weights (along with pointers to the associated leaves), and combined weights (along with pointers to the trees) being put in the back of the second queue. This assures that the lowest weight is always kept at the front of one of the two queues.

Creating the tree:

1. Start with as many leaves as there are symbols.
2. Enqueue all leaf nodes into the first queue (by probability in increasing order so that the least likely item is in the head of the queue).
3. While there is more than one node in the queues:
  - Dequeue the two nodes with the lowest weight.
  - Create a new internal node, with the two just-removed nodes as children (either node can be either child) and the sum of their weights as the new weight.
  - Enqueue the new node into the rear of the second queue.
4. The remaining node is the root node; the tree has now been generated.

It is generally beneficial to minimize the variance of codeword length. For example, a communication buffer receiving Huffman-encoded data may need to be larger to deal with especially long symbols if the tree is especially unbalanced. To minimize variance, simply break ties between queues by choosing the item in the first queue. This modification will retain the mathematical optimality of the Huffman coding while both minimizing variance and minimizing the length of the longest character code.

After creating the tree next step is to find the codeword of each character of the text. This is obtained by taking the bits from the leaf node of that character to the root node of the tree. By using this code words the full text can be compressed to a bit pattern or final code word.

### 5.3 Insertion of shuffled CRC code

Here comes the most important step of the project, which is nothing but encryption. The encryption is done by taking the same CRC code or the generator polynomial used for the error detection. So, this method is quite advantageous as the error detection and the encryption is done with the same CRC code.

After the text was compressed using Huffman coding, we will get the code word. This code word is taken for error detection using CRC code. So, a generator polynomial of length  $P$  is used. Let the code word length is  $N$ . Then we will get the remainder of length  $P-1$ . This remainder will be attached with the code word and it will be sent to the receiver. Before that, encryption will be done.

In this method the length of the code word ( $N$ ) will be divided with the length of the generator polynomial ( $P$ ). The nearest prime number of the quotient will be taken, which will be multiplied with the generator polynomial. The resulted bit pattern will be used for encryption. Let this pattern is of length  $m$ . These  $m$  bits will be XORed with the last  $m$  bits of the code word. The same operation will be done again with the previous  $m$  bits. The process is going on until the last bit pattern of the code word, whose length is less than  $m$ . They will be left unoperated.

The encryption process will be much more strengthened by using the CRC code again. These CRC bits will be shuffled with the  $N$  bit code word using a random number generation. A series of random numbers are generated, which will be more than zero and less than  $N$ . These numbers are the positions, where the bits of the CRC code will be introduced into the code word. The number of random numbers is same as the number of bits of the CRC code.

In this way the text will be coded and encrypted with the shuffled CRC code. The encrypted word will be sent to the receiver, where decryption will be done in the reverse manner of the encryption process. So, we will get the same code word again, XOR of XOR of a number is the same number. Then the code word will be checked for error. For that, the code word with the CRC code will be divided with the same generator polynomial. If the remainder will be zero, then no error. Otherwise error is present. Then some error correction method will be used to remove the error.

# Chapter 6

## **EXPERIMENTATION & RESULTS**

## 6.1 Source code

The code is written in MATLAB. It consists of seven phases. These are

- Starting of the code with inputting the text file
- Formation of Tree
- Encoding
- Error detection with CRC
- Encryption with shuffled CRC
- Decryption
- Decoding

### Starting phase

```
clc;
clear all;
k=input('Enter the file name :','s');
fid = fopen(k,'r');
F = fread(fid);
img = char(F);
mx=255;
[x y z]=size(img);
h(1:mx)=0;
for i=1:y
    iy=img(i);
    val=double(iy);
    h(val)=h(val)+1;
end
end
%Probability calculating phase started
i=1:mx;
p(i)=h(i)/(x*y);
j=1;
for i=1:mx
    if(p(i)~=0)
        lst(j)=i;
        lst(j+1)=p(i);
        j=j+2;
    end
end
end
[tt,mx]=size(lst);
%sorting phase started
for i=2:2:mx
    for j=i:2:mx
        if (lst(i)>lst(j))
```

```

        temp1=lst(i-1);
        temp2=lst(i);
        lst(i-1)=lst(j-1);
        lst(i)=lst(j);
        lst(j-1)=temp1;
        lst(j)=temp2;
    end
end
end
fhtree1(lst,img);

```

### Formation of Tree

```

function HT=fhtree1(lst,img)
[p,q]=size(lst);
[tt,mx]=size(lst);
sz1=q;
xx=1;
k1=0;
prt=0;
while (k1<1)
    k1=lst(2)+lst(4);
    prt=prt-1;
    lstn(xx)=lst(1);
    lstn(xx+1)=0;
    lstn(xx+2)=prt;
    xx=xx+3;
    lstn(xx)=lst(3);
    lstn(xx+1)=1;
    lstn(xx+2)=prt;
    xx=xx+3;
    lst(1)=prt;
    lst(2)=k1;
    lst(3)=99;
    lst(4)=99;

    for i=2:2:mx
        for j=i:2:mx
            if (lst(i)>lst(j))
                temp1=lst(i-1);
                temp2=lst(i);
                lst(i-1)=lst(j-1);
                lst(i)=lst(j);
                lst(j-1)=temp1;
                lst(j)=temp2;
            end
        end
    end
end
end
end

```

```

lstrn(xx)=lstr(1);
lstrn(xx+1)=lstr(2);
lstrn(xx+2)=lstr(3);
fhcode1(lstrn,img);

```

### Encoding

```

function HC=fhcode1(lstrn,img)
[lm,ln]=size(lstrn);
ntt=ln-1;
[im,in]=size(img);
t=0;
idd=input('Enter destination huffman code file name : ','s');
tab='table.m';
tb = fopen(tab,'w+');
idd=fopen(idd,'w+');
fst1="";
fst2="";
ed=0;
din=0;
for i=1:in
    k=img(i);
    ftemp=img(i);
    a=0;
    for j=1:3:ln
        if(lstrn(j+2)==99)
            break;
        end
        if(lstrn(j)==k)
            a=a+1;
            ary(a)=lstrn(j+1);
            k=lstrn(j+2);
        end
    end
    % Reversing the reverse Huffman Code%
    for b=a:-1:1
        t=t+1;
        hc(t)=ary(b);
        fprintf(idd,'%d',ary(b));
        fst1=int2str(ary(b));
        fst2=strcat(fst2,fst1);
    end
    %Building Huffman Table for Decoding%
    din=0;
    for z=1:ed
        if dict(z)==ftemp
            din=1;
        end
    end
end

```

```

if din==0
    ed=ed+1;
    dict(ed)=ftemp;
    fprintf(tb,'%c',' ');
    fprintf(tb,'%c',ftemp);
    fprintf(tb,'%s',fst2);
end
fst1="";
fst2="";
end
fclose(tb);
fclose(idd);
disp('Generated Compressed file');
error1(hc);

```

### **Error detection using CRC**

```

function ERR=error1(hc)
hc1=hc;
g=input('enter the generator polynomial g=');
[m,n]=size(hc1);
[o,p]=size(g);
for i=n+1:n+p-1
    hc1(i)=0;
end
a=zeros(1,p);
b=zeros(1,p);
r=zeros(1,p-1);
for i=1:p
    a(i)=hc1(i);
end
b=xor(a,g);
j=p+1;
d=zeros(1,p);
while(j<=n)
    a=zeros(1,p);
    for i=1:p-1
        a(i)=b(i+1);
    end
    a(p)=hc1(j);
    j=j+1;
    if(a(1)==1)
        b=xor(a,g);
    else
        b=xor(a,d);
    end
end
end
for i=1:p-1
    r(i)=b(i+1);

```

```

end
encrypt1(hc,g,r);

```

### Encryption with shuffled CRC

```

function ENC=encrypt1(hc,g,r)
[m,n]=size(hc);
hc2=hc;
hc3=zeros(1,n);
hc3=hc;
n1=n;
[o,p]=size(g);
a=rem(n,p);
n=n-a;
a=n/p;
a=a+1;
t=0;
%finding a nearest prime number
while(a>0)
for i=2:a-1
    if(rem(a,i)~=0)
        t=0;
    else
        t=1;
        break;
    end
end
if(t==0)
    break;
end
a=a+1;
end
%finding the encrypting word
[o,p]=size(g);
s=0;
for j=1:p
    s=s+g(j)*(2^(p-j));
end
b=a*s;
i=1;
while(b>0)
    c(i)=rem(b,2);
    b=b-c(i);
    b=b/2;
    i=i+1;
end
[e,f]=size(c);
for i=1:f
    d(i)=c(f+1-i);

```

```

end
%encryption phase
h=n-rem(n,f);
h=h/f;
for i=1:h
    k=1;
    for j=n+1-i*f:n-(i-1)*f
        hc(j)=xor(hc(j),d(k));
        k=k+1;
    end
end
%encryption of remainder
[u,v]=size(r);
w(1)=1;
w(2)=2;
for i=3:v
    if(rem(i,2)~=0)
        w(i)=w(i-1)+2;
    else
        w(i)=w(i-1)+1;
    end
end
end
w
[m,n]=size(hc);
for i=1:v
    k=w(i);
    if(k==1)
        hc=[r(i) hc];
    elseif(k==2)
        hc=[hc(1) r(i) hc(2:n)];
    else
        hc=[hc(1:k-1) r(i) hc(k:n)];
    end
end
n=n+1;
end
idd=input('Enter destination file name for encrypted text : ','s');
id=fopen(idd,'w+');
for i=1:n
    fprintf(id,'%d',hc(i));
end
fclose(id);
disp('encrypted file generated')
decrypt1(hc,hc3,g);

```

### Decryption

```

function DEC=decrypt1(hc,hc3,g)
w=input('enter the random number matrix w=');
[m,n]=size(hc);

```

```

[u,v]=size(w);
for i=v:-1:1
    k=w(i);
    if(k==2)
        hc=[hc(1) hc(3:n)];
    elseif(k==1)
        hc=[hc(2:n)];
    else
        hc=[hc(1:k-1) hc(k+1:n)];
    end
    n=n-1;
end
[m,n]=size(hc);
hc2=hc;
[o,p]=size(g);
a=rem(n,p);
n=n-a;
a=n/p;
a=a+1;
t=0;
%finding a nearest prime number
while(a>0)
    for i=2:a-1
        if(rem(a,i)~=0)
            t=0;
        else
            t=1;
            break;
        end
    end
    if(t==0)
        break;
    end
    a=a+1;
end
%finding the encrypting word
[o,p]=size(g);
s=0;
for j=1:p
    s=s+g(j)*(2^(p-j));
end
b=a*s;
i=1;
while(b>0)
    c(i)=rem(b,2);
    b=b-c(i);
    b=b/2;
    i=i+1;
end

```

```

end
[e,f]=size(c);
for i=1:f
    d(i)=c(f+1-i);
end
%decryption phase
[m,n]=size(hc);
h=n-rem(n,f);
h=h/f;
for i=1:h
    k=1;
    for j=n+1-i*f:n-(i-1)*f
        hc(j)=xor(hc(j),d(k));
        k=k+1;
    end
end
hc=hc3;
[s,t]=size(hc);
nme=input('Enter the destination file name :','s');
id = fopen(nme,'w+');
for i=1:8:t
    ck=t-i+1;
    if(ck>8)
        tp=(hc(i:i+7));
        num=8;
    else
        tp=(hc(i:t));
        num=ck;
    end
    temp1=b2d(tp,num);
    temp2=char(temp1);
    fprintf(id,'%c',temp2);
end
fclose(id);
return

```

### **Decoding**

```

clc;
clear all;
nme=input('Enter the file name :','s');
id = fopen(nme,'r');
a = fscanf(id,'%c',inf);
fclose(id);
[m,n]=size(a);
k=1;
for i=1:n
    b(i)=double(a(i));
end

```

```

for i=1:n
    c = dec2bin(b(i),8);
    for j=1:8
        d(k)=c(j);
        k=k+1;
    end
end
nme2='table.m';
id2 = fopen(nme2,'r');
a2=fscanf(id2,'%c',inf);
fclose(id2);
[m1,n1]=size(a2);
chk=0;
cnt=1;
str="";
temp=0;
for j=1:n1
    if chk==1 & a2(j)~=' '
        str=strcat(str,a2(j));
        cd2{cnt-1}=str;
    end
    if temp==1
        cd1(cnt)=a2(j);
        cnt=cnt+1;
        chk=1;
        temp=0;
        str="";
    end

    if a2(j)==' '
        temp=1;
        chk=0;
        if j>1
            if a2(j-1)==' '
                chk=1;
                temp=0;
                str="";
            end
        end
    end

end
[m2,n2]=size(d);
[m3,n3]=size(cd2);
% Enter a file name to deliver the decoded output
nme=input('Enter the file name (to produce output) :','s');
id = fopen(nme,'w+');
comp="";

```

```

tap=0;
disp('Decompression starts.....');
for i=1:n2
    cnt=0;
    z1=d(i);
    m=num2str(z1);
    for j=1:n3
        k=strcmp(m,cd2(j));
        if(k==1 & tap==0)
            fprintf(id,'%c',cd1(j));
            comp="";
            cnt=1;
        end
    end
end

if(cnt==0)
    comp=strcat(comp,num2str(z1));
    tap=1;
    for j=1:n3
        m=cd2(j);
        k=strcmp(comp,cd2(j));
        if(k==1)
            cd1(j);
            fprintf(id,'%c',cd1(j));
            comp="";
            tap=0;
        end
    end
end
end
disp('Decompression Over');
fclose(id);
return

```

## 6.2 Results

### Input text :-

We have completed the project

"data compression by huffman coding & encryption by shuffled crc code"

under the guidance of prof. G. S. Rath

Dept of Electronics & Communication Engg

NIT Rourkela

Compressed output:-

11100101000101100001101001010100001010111110000101011011111100000110001000110  
100111000000010101101110111100010101100000111001110111111000100111010110001110  
100011110101010111110000101011011101100011110011110010011110001001011110111010  
111101100000110010010100100010111010010010101111100100011001101001111011010101  
001010000100011111011010111011010011100111100010010111101110101111011111001000  
001100100101001011111100010001101011111011011110101111100100010001110101111110  
001000110001001000100011011101001110000000101111101011001001110001110100100011  
100010111001001010101101110111100100100101101011110000001011010111100111010110  
101111010011010001110000111110001001110001100001101001110111001001010111101101  
11111000011100111101111000100100110111111001010101001011110001011000010100101  
011000100100110111110100011100111100010010111101100100111101111101111110001001  
110011011100001111001001011110100110001100110110101011100011111111010

Generator Polynomial used:-

[1 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 1 0 0 0 1 1 1 0 1 1 0 1 1 0 1 1 1]

Encrypted output:-

101001011110000101010111100010100111000001001110000110100101010000100101001001  
011010101111010110111011001110110001010111100101001100110100101010110100011101  
101101110101100100000010010101100001000101010000001010110001001100011100000000  
111110011100111111101010001100110001011001010001001011110110011111101011101011  
010111110001000101000110010010110010110110011100011001000010100111001001110011  
001000001100011101111110001010000110111111101001000100000000100110111100001100  
110110000100100110000011011110111111100010101100111001111000010011001110111100  
100101101010111011111101011111100011000001111110111000101101110010100100101101  
001010001011001101111100000011010000001111110000100000001101011001000001100101  
111000000011110011001000010101100001010000111100110111001100100100000000101010  
001101110111110101101011000110111010011010111101110110101010111000110010100110  
100001001010101001100100011001011000001011000001001001001010010100000001010011  
10101110010101000100010

Decoded output:-

We have completed the project

"data compression by huffman coding & encryption by shuffled crc code"

under the guidance of prof. G. S. Rath

Dept of Electronics & Communication Engg

NIT Rourkela

**6.3 Conclusion**

We implemented successfully the data reduction by huffman coding and encryption by insertion of shuffled cyclic redundancy code.

## REFERENCES

1. R. Merkle, "Secure communication over an insecure channel," submitted to *Communications of the ACM*.
2. D. Kahn, *The Codebreakers, The Story of Secret Writing*. New York: Macmillan, 1967.
3. C. E. Shannon, "Communication theory of secrecy systems," *Bell Syst. Tech. J.*, vol. 28, pp. 656–715, Oct. 1949.
4. M. E. Hellman, "An extension of the Shannon theory approach to cryptography," submitted to *IEEE Trans*.
5. Helen Fouché Gaines, "Cryptanalysis", 1939, Dover. ISBN 0-486-20097-3 .
6. Abraham Sinkov, *Elementary Cryptanalysis: A Mathematical Approach*, Mathematical Association of America, 1966. ISBN 0-88385-622-0