CHAPTER 3

RESEARCH METHODOLOGY
3.1 Methodology

The method that has been followed to carry out the current study is as follows:

1. Critical analysis of select existing lossless compression algorithm and to identify their salient features.
2. Critical analysis of select existing symmetric algorithm and to identify their salient features.
3. Designing a new compression and encryption algorithm for text data based on salient features identified.
4. Generating English text data set and testing the proposed algorithm with other type of data compression algorithm, e.g. RLE, Huffman, Adaptive Huffman and Shannon Fano algorithm, for comparison in terms of compression ratio, encoding time, and decoding time.
5. Generating English text data set and testing the proposed algorithm with other type of encryption algorithm, e.g. AES algorithm, for comparison in terms of encryption time, decryption time, and cryptanalysis.
6. Testing of proposed algorithm for English language test for total execution time, total memory consumption with other types of compression and encryption algorithms.
7. Generating other language text data set and testing the proposed algorithm for compression and encryption in terms of compression ratio, encoding time, decoding time, encryption time, decryption time, total execution time.

3.2 Study of Existing Lossless Compression Algorithms

The different types of lossless text data compression algorithms have been studied and analyzed. Lossless compression algorithms are widely depends upon the observation of the correlation between the parts of data (patterns) and replace the repetitions by a shorter one with a reference to a dictionary which containing the original.
3.2.1 Lempel Ziv Algorithms

Lempel Ziv Algorithm is a family of lossless data compression algorithms. It was proposed by Jacob Ziv and Abraham Lempel in 1977 and 1978.

Figure 3.1: Tree structure representing the whole family of Lempel Ziv Algorithms (Jain et. al. (2014))

3.2.1.1 Algorithm LZ77

Algorithm is known by its author and year of the implementation. Fact used in LZ77 is that, words and phrases within a text file are repeated frequently and when there is repetition, they can be encoded as a pointer to their earlier occurrence along with the number of characters to be matched. It does not require any prior information about the source characteristics. The dictionary is a segment that contains information about the encoded sequence in the LZ77 approach. The encoded sequence contains two parts: A search buffer that contains the information of the recent encoded sequence and Lookahead buffer that contain the successor of the information to be encoded. The scheme works on the basis of the sliding window for the search match and output pointer to that match. If no match is found, the output doesn’t contain any pointers.

In LZ77, the pointers are output as the triple <o,l,c>, where ‘o’ is an offset to the match, ‘l’ is length of the match, and ‘c’ is the next symbol after the match. According to Senthil et. al. (2005) algorithm output a null pointer when there is no match is found and represented in the lookahead buffer. The constant value is maximized and the offset value to the match and its length is limited. The performance of the LZ77 depends on these values. The offset
is encoded on 12-16 bits and limits from 0 to 65535 symbols. So, there is no need to remember more than 65535 last seen symbols in the sliding window. The match length is usually encoded on 8 bits, which gives maximum match length equal to 255.

The LZ77 algorithm is given below:

```plaintext
While (lookAheadBuffer not empty)
{
    get a reference (position, length) to longest match;
    if (length > 0)
    {
        output (position, length, next symbol);
        shift the window length+1 position along;
    }
    else {
        output (0, 0, first symbol in the lookahead buffer);
        shift the window 1 character along;
    }
}
```

The execution time of an algorithm is similar for compression and decompression. The encoding process of LZ77 is very fast due to the transmission of the several input symbols as one reference (a triple). Decoding is faster than the encoding process. In LZ77, compression time is higher due to searching for the longest match, whereas decompression time is faster as each reference is simply replaced by the string that it points to.

The scheme LZ77 can be solved in numerous ways. Wiseman (2007) stated that the best known LZ77 schemes are LZSS, LZH and LZB.

LZSS attempts to replace a string of symbols with a reference to a dictionary location for the same string. It is intended that the dictionary reference should be shorter than the string it replaces. LZSS uses one-bit flags to indicate whether the next chunk of data is a literal (byte) or a reference to an offset/length pair.
The combination of the Ziv-Lempel and Huffman is LZH. The coding process occurs in two steps. Firstly, it is same as LZSS and next it uses the statistics to code pointers and Huffman coding to explicit the characters.

LZB was published by Banikazemi (2009) used a scheme for encoding the references and lengths with the different sizes. LZB technique uses different coding for both segments of the pointer. LZB achieves a better compression than LZSS and has added value of being less sensitive to the parameters decision.

3.2.1.2 Algorithm LZ78

An explicit dictionary is maintained for the dictionary based compression algorithm. An identical dictionary is ensured in both encoding and decoding process. The output is the codeword that consists of <i,c>, where ‘i’ is an index, refer to the longest matching dictionary entry and ‘c’ refer to first non-matching symbol. In addition to output, the algorithms also add the index and symbol pair to the dictionary. When a symbol, which is not yet available in the dictionary, the codeword will get the index value 0 and symbols are added to the dictionary. The dictionary has build up gradually with this method.

The algorithm for LZ78 is given below:

```
  w := NIL;
  while (there is input )
  {
    K:= next symbol from input;
    if (wK exists in the dictionary)
    {
      w := wK;
    }
    else {
      output (index(w), K);
      add wK to the dictionary;
      w := NIL;
    }
  }
```
LZ78 algorithm captures the patterns and holds them indefinitely. Drawback with LZ78 is that the dictionary reference inclines without any limitation. Several methods are there to limit the dictionary size. The encoding and decoding speed of LZ78 is faster than LZ77.

3.2.1.3 Algorithm LZW

Welch (1984) presented LZW (Lempel–Ziv–Welch) algorithm based on LZ78 which applied the LZSS principle. The dictionary is initialized with all possible symbols from the input alphabet. It guarantees that a match will always be found. LZW would only send the index to the dictionary. The input to the encoder is accumulated in a pattern ‘w’ as long as ‘w’ is contained in the dictionary. If the addition of another letter ‘K’ results in a pattern ‘w*K’ that is not in the dictionary, then the index of ‘w’ is transmitted to the receiver, the pattern ‘w*K’ is added to the dictionary and another pattern is started with the letter ‘K’.

The algorithm then proceeds as follows:

\[
\begin{align*}
\text{w} & := \text{NIL}; \\
\text{while (there is input)} & \\
\{ & \\
\quad \text{K} & := \text{next symbol from input}; \\
\quad \text{if (wK exists in the dictionary)} & \\
\quad \quad & \\
\quad \quad & \\
\quad \quad \text{w} & := \text{wK}; \\
\quad \quad & \\
\quad \text{else} & \\
\quad \quad & \\
\quad \quad & \\
\quad \quad & \\
\quad \quad \text{output (index (w))}; \\
\quad \quad & \\
\quad \quad \text{add wK to the dictionary}; \\
\quad \quad & \\
\quad \quad \text{w} & := \text{k}; \\
\quad & \\
\text{\}} & \\
\end{align*}
\]

In LZW, the pointer size is 12 bits which can grow up to 4096 dictionary entries. Once the limit is reached, the dictionary becomes static. LZFG which was developed by Fiala et. al. (1989) gives fast encoding and decoding and good compression without undue storage.
requirements. According to Kodituwakku et. al. (2008), LZFG algorithm uses the
dictionary building technique as LZ78, but the only difference is that it stores the elements
in a tree data structure. Here, the encoded characters are placed in a window (as in LZ77)
to remove the oldest phrases from the dictionary.

Prepressure.com (2013) defined Advantages and disadvantages of LZW algorithm:

- LZW compression works best for files containing lots of repetitive data. This is
  often the case with text and monochrome images.
- LZW compression is fast.
- LZW is a fairly old compression technique. All recent computer systems have the
  horsepower to use more efficient algorithms.
- Royalties have to be paid to use LZW compression algorithms within applications

### 3.2.2 Run Length Encoding Algorithm (RLE)

According to Suarjaya (2012), Run-Length Encoding (RLE) is one of the simplest
compression algorithm, which is created especially for data with strings of repeated
symbols (the length of the string is called a run). The idea is to encode repeated symbols as
a pair: the length of the string and the symbol. For example, the string ‘abbaaaaabaabbaa’
of length 16 bytes (characters) is represented as 7 integers plus 7 characters, which can be
easily encoded on 14 bytes (as for example ‘1a2b5a1b2a3b2a’). The biggest problem with
RLE is that in the worst case the size of compressed data can be two times more than the
size of input data. To eliminate this problem, each pair (the lengths and the strings
separately) can be later encoded with an algorithm like Huffman coding.

### 3.2.3 Huffman Coding

Huffman (1951) designed Huffman coding algorithm in 1950 which is used for text
compression. The idea is to replace fixed-length codes (such as ASCII) by variable-length
codes and assign shorter code to the more frequently occurred symbol and thus decrease
the overall length of the data. When using variable-length code, it is necessary to create a
(uniquestly decipherable) prefix-code, avoiding the need for a separator to determine code
boundaries. Huffman coding creates such a code. Huffman algorithm is almost same as
Shannon - Fano algorithm. Both the algorithms employ a variable bit probabilistic coding
method. Both the algorithms differ slightly in the manner in which the binary tree is built. Huffman uses a bottom-up approach and Shannon Fano uses a Top-down approach. The Huffman algorithm is simple and can be described in terms of creating a Huffman code tree.

The procedure for building this tree is:

1. Start with a list of free nodes, where each node corresponds to a symbol in the alphabet.
2. Select two free nodes with the lowest weight from the list.
3. Create a parent node of these two nodes selected and the weight is equal to the weight of the sum of two child nodes.
4. Remove the two child nodes from the list and the parent node is added to the list of free nodes.
5. Repeat the process starting from step 2 until only a single tree remains.

After building the Huffman tree, the algorithm created a prefix code for each symbol from the alphabet, simply by traversing the binary tree from the root to the node, which corresponds to the symbol. It assigns 0 for a left branch and 1 for a right branch. The algorithm presented above is called as a semi-adaptive or semi-static Huffman coding as it requires knowledge of frequencies for each symbol from alphabet. Chanhemo et. al. (2011) stated that along with the compressed output, the Huffman tree with the Huffman codes for symbols or just the frequencies of symbols which are used to create the Huffman tree must be stored. This information is needed during the decoding process and it is placed in the header of the compressed file.

### 3.2.4 Shannon Fano Coding

Shannon – Fano algorithm developed by Claude Shannon and R. M. Fano, which is used to encode messages depending upon their probabilities. According to Rueda et. al. (2004) Shannon-Fano algorithm allot less number of bits to highly probable messages and more number of bits to rarely occurring messages.

The algorithm is as follows:

1. From the given list of symbols, develop either frequency or probability table.
2. Sort the table according to the frequency, with the most frequently occurring symbol at the top.
3. Divide the table into two halves with the total frequency count of the upper half should be close to the total frequency count of the bottom half as possible.
4. Assign the upper half of the list a binary digit ‘0’ and the lower half a ‘1’.
5. Recursively apply the steps 3 and 4 on the two halves, subdividing groups and adding bits to the codes until each symbol has become a corresponding leaf on the tree.

Shannon-Fano coding does not guarantee that an optimal code is generated. Shannon – Fano algorithm is more efficient when the probabilities are closer to inverses of powers of 2.

3.2.5 Arithmetic Encoding
Rissane et. al. (1981) designed arithmetic encoding technique which provides extremely high coding efficiency and superior compression to the better-known Huffman algorithm. Arithmetic coding is a method to ensure lossless data compression. It is indeed a form of variable length entropy encoding. In the case of other entropy encoding techniques, the input message is separated into its component symbols and each symbol is replaced by a code word. But arithmetic coding encodes the entire message into a single number, a fraction n where (0.0_n< 1.0). The coding algorithm is symbol wise recursive; i.e., it operates upon and encodes (decodes) one data symbol per iteration or recursion. On each recursion, the algorithm successively partitions an interval of the number line between 0 and 1, and retains one of the partitions as the new interval. Thus, the algorithm successively deals with smaller intervals, and the code string, viewed as a magnitude, lies in each of the nested intervals. The data string is recovered by using magnitude comparisons on the code string to recreate how the encoder must have successively partitioned and retained each nested subinterval.

3.2.6 Adaptive Huffman Coding
The basic Huffman algorithm suffers from the drawback that to generate Huffman codes probability distribution of the input set is required, which is often not available. Moreover,
it is not suitable for cases when probabilities of the input symbols are changing. Shanmugasundaram et al. (2011) said that Adaptive Huffman coding technique was developed based on Huffman coding first by Newton Faller and Robert G. Gallager and then improved by Donald Knuth and Jefferey S. Vitter.

In this method, a different approach known as sibling property is followed to build a Huffman tree. Kunuth (1985) said that both sender and receiver maintain dynamically changing Huffman code trees whose leaves represent characters seen so far. Initially the tree has only the 0-node which represents a message that have yet to be seen. Here, the Huffman tree includes a counter for each symbol and the counter is updated every time when a corresponding input symbol is coded. Vitter (1987) said that Huffman tree under construction is still a Huffman tree if it is ensured by checking whether the sibling property is retained. If the sibling property is violated, the tree has to be restructured to ensure this property. Usually this algorithm generates codes that are more effective than static Huffman coding. According to Brickell (1985) storing Huffman tree along with Huffman codes for symbols with the Huffman tree is not needed here. It is superior to Static Huffman coding in two aspects: It requires only one pass through the input and it adds little or no overhead to the output. But this algorithm has to rebuild the entire Huffman tree after encoding each symbol which becomes slower than the static Huffman coding.

3.2.7 Salient Features of Lossless Compression Algorithms –

1. Lossless compression algorithms work on the principle of statistical modeling techniques. Using this technique any non-random file which contains duplicated information can be condensed. Statistical modeling determines the probability of a character or phrase which is used to generate code for specific characters or phrases based and assign the shortest codes to the most common data. Using these techniques 8-bit character or a string of such characters could be represented with just a few bits resulting in a large amount of redundant data being removed.

2. RLE works by reducing the physical size of a repeating string of characters. RLE compression is only efficient with files that contain lots of repetitive data.

3. In Huffman encoding the characters that occur most often, such the space and period, may be assigned as few as one or two bits. Infrequently used characters,
such as: !, @, #, $ and %, may require a dozen or more bits. Huffman uses a bottom-up approach. Huffman code is a minimum-length code in the sense that no other encoding has a shorter average length. Huffman produces the best variable-length codes when the probabilities of the symbols are negative powers of 2.

4. Shannon-Fano algorithm allots less number of bits to highly probable messages and more number of bits to rarely occurring messages. Shannon Fano uses a top-down approach. Shannon-Fano provides a similar result compared with Huffman coding.

5. Adaptive Huffman coding is dynamic coding technique which builds code as the symbols are being transmitted, without having initial knowledge of source distribution. It allows one-pass encoding and adaptation to work according to changing conditions in data.

6. Arithmetical coding evaluates the probability of each symbol and optimizes the length of the required code. Arithmetic coding works by representing a number by an interval of real numbers between 0 and 1. As the message becomes longer, the interval needed to represent it becomes smaller and smaller, and the number of bits needed to specify that interval increases.

3.3 Study of Existing Symmetric Key Encryption Algorithms

The different types of encryption algorithms have been studied and analyzed which is particularly based on the symmetric key encryption algorithm. Symmetric-key algorithms use the same cryptographic keys for both encryption of plaintext and decryption of ciphertext. The keys may be identical or there may be a simple transformation to go between the two keys. The performance of the different Algorithm has been analyzed.

3.3.1 DES

DES is a Data Encryption Standard algorithm, developed by IBM in 1972. Mandal (2012) stated that DES is based on Lucifer, which was developed by Horst Feistel. After the assessment of DES strength and modifications the National Bureau of Standards and Technology (NIST) has approved this algorithm and became a federal standard in 1977. DES uses a 64-bit key, but eight of those bits are used for parity checks, effectively
limiting the key to 56-bits. Hence, it would take a maximum of $2^{56}$ or 72,057,594,037,927,936, attempts to find the correct key. DES use 56-bit keys to encrypt 64-bit blocks. The algorithm converts a 64-bit input into a 64-bit output using a series of steps. The same steps, with same key are used in reverse to decrypt the message. A key is usually chosen randomly from the available possible keys. According to Stalling (2010) DES uses eight predefined S-boxes which have been determined by the U.S National Security Agency (NSA). Each of the S-boxes accepts 6-bits as input and produces 4-bits as output. These S-boxes are resistant against differential cryptanalysis, which was earlier known in the 1990s. The encryption process of DES is done in 16 rounds.

Kak (2015) described DES encryption algorithm as shown in Figure 3.2 as follows:

1. DES consists of 16 rounds to process plain text, and each round consists of “substitution” followed by “permutation”.
2. The input block of each round is divided into two halves that denoted by L for left half and R for right half respectively.
3. In each round, R half goes through unchanged, but the L half goes through an operation that depends on R and the encryption key.
4. The permutation step at the end of each round consists of swapping the modified L and R. Therefore, the L for the next round would be R of the current round and R for the next round is the output L of the current round.
3.3.2 3DES

3DES is also known as Triple Data Encryption Algorithm (TDEA or triple DEA), which applies Data Encryption Standard (DES) three times on each data block. Karthik et. al. (2014) stated that Triple DES was designed to solve many of the security vulnerabilities of DES. It is very easy to design and modify the existing software to use 3DES, as it is based on the DES algorithm. Singh et. al. (2013) said that the larger key length of 3DES eliminates many of the attacks that were possible in DES due to shorter key length. 3DES uses three 56-bit DES keys which creates a key with the total length of 168 bits. According to Alanazi et. al. (2010) the process of encryption using 3DES is exactly the same as DES, except that instead of single key three keys is used in the following order:
1. Encryption using DES with the first 56-bit key
2. Decryption using DES with the second 56-bit key
3. Encryption using DES with the third 56-bit key

Because Triple-DES applies the DES algorithm three times (hence the name), Triple-DES takes three times as long as standard DES. Decryption using Triple-DES is the same as the encryption, except it is executed in reverse.

3.3.3 AES

DES is no longer adequate for security due to increase computing power. Alanazi et. al. (2010) stated that in 1998 DES was cracked in less than three days by a cracker created by electronic frontier foundation. Kaufman (2002) mentioned that NIST wanted an algorithm that would provide good security and should be efficient and flexible. In 2001, federal information processing standards 197 (FIPS 197) announced a standard form of the Rijndael algorithm. This standard was called Advanced Encryption Standard. Alanazi et. al. (2010) stated that AES allows for block sizes of 128, 168, 192, 224 and 256 bits. Paar (2010) stated that the key size varies between 128, 192 and 256 bits; but only the key size of 128 bits was approved as the AES standard. At the basic level the Rijndael algorithm uses a number of rounds to transform the data for each block. The number of rounds used is 10, 12 or 14. The initial block (also known as state) is added to an expanded key derived from the initial cipher key. Then the round processing occurs consisting of operations of the S-box, shifts, and a Mix Column. The result state is then added to the next expanded key. This is done for all rounds, with the exception of the Mix Column operation of the final round. The final result is the encrypted cipher block.

Wikipedia.org (2015) states that each round of AES is governed by the following transformations:

- Substitute Byte transformation
  AES contains 128 bit data block, which means each of the data blocks has 16 bytes. In sub-byte transformation, each byte (8-bit) of a data block is transformed into another block using an 8-bit substitution box which is known as Rijndael S-box.
- **Shift Rows transformation**
  It is a simple byte transposition, the bytes in the last three rows of the state, depending upon the row location, are cyclically shifted. For 2nd row, 1 byte circular left shift is performed. For the 3rd and 4th row 2-byte and 3-byte left circular left shifts are performed respectively.

- **Mix columns transformation**
  This round is equivalent to a matrix multiplication of each Column of the states. A fix matrix is multiplied to each column vector. In this operation the bytes are taken as polynomials rather than numbers.

- **Add round key transformation**
  It is a bitwise XOR between the 128 bits of present state and 128 bits of the round key. This transformation is its own inverse.

### 3.3.4 RC6

RC6 is a block cipher which uses 128 bit block size and supports key sizes of 128, 192 and 256 bits. It was designed in order to meet the requirements of the AES. It is an improvement of the RC5 Algorithm. Charbathia et. al. (2014) stated that RC6 is very similar to RC5 in structure, using data-dependent rotations, and modular addition and XOR operations; in fact, RC6 could be viewed as interweaving two parallel RC5 encryption processes. However, RC6 does use an extra multiplication operation not present in RC5 in order to make the rotation depends on every bit in a word, and not just the least significant few bits. RC6 provides better security against attacks. It uses of 4 registers, each of 32 bit, and is more secure than the RC5. It is also protected from various other possible security attacks. It uses fewer rounds and offers a higher throughput. Figure 3.3 shows RC6 encryption algorithm.
RC6 with 15 rounds or less, running on input blocks of 128 bits, it has been shown that the resulting ciphertext could be distinguished from a random series of bits. One of the conditions for an encryption algorithm to be secure is that its output resembles a completely random series of bits. Several applications check for randomness of bit streams to indicate strong encryption. Moreover, the writers of the algorithm have shown an attack against RC6 running with up to 15 rounds that is faster than an exhaustive key search. For one class of weak keys, it was shown that full randomness is not accomplished for up to 17 rounds of the algorithm.

For RC6 with 16 rounds, a linear cryptanalysis attack is possible, but requires $2^{119}$ known plaintexts, which makes this attack quite infeasible. The RC6 algorithm is robust against differential cryptanalysis, provided that it applies more than 12 rounds.
3.3.5 RC2

RC2 is a symmetric-key block cipher algorithm designed by Ronald Rivest in 1987. RC stands for "Rivest Cipher" or "Ron's Code". According to Charbathia et. al. (2014) RC2 works on 64-bit blocks which are divided into four words of each sixteen bits. It is an iterated block cipher where the ciphertext is computed as a function of the plaintext and the secret key in a number of rounds. There are two kinds of rounds in RC2, the mixing rounds and the mashing rounds. There are in total 16 mixing rounds and 2 mashing rounds. In each round each of the four sixteen-bit words in an intermediate ciphertext is updated as a function of the other words. Each of the mixing rounds takes a 16-bit subkey. The 64 subkeys are derived from the user selected key which can be of length from one to 128 bytes.

The 18 rounds are performed using the following interleaved sequence:

- Perform 5 mixing rounds.
- Perform 1 mashing round.
- Perform 6 mixing rounds.
- Perform 1 mashing round.
- Perform 5 mixing rounds.

RC2 uses key-expansion algorithm by which an expanded key consisting of 64 (16-bit words) is produced depending in a complicated way on every bit of the supplied "variable-length" input key. A mixing round consists of four applications of the "mix-up" transformation, as shown in the diagram. A round is "mashed" by adding to it one of the 16-bit words of the expanded key.

3.3.6 Salient Features of Symmetric Key Encryption Algorithms –

- AES is considered as most secure symmetric key algorithm. It uses 256 bit key size which is considered as most secure. AES design is based on a substitution-permutation network. Its symmetric and parallel structure provides great flexibility for implementers, with effective resistance against cryptanalytic attacks.
- RC6 cipher uses four 32-bit registers for operations instead of two, as in RC5, which makes it possible to do two rotations per round. RC6 included Integer
multiplication and increased the diffusion achieved, this leads to higher security, fewer rounds and an increased throughput.

- RC2 is a 64-bit block cipher with a variable size key. Its 18 rounds consist with 16 rounds of mixing punctuated by 2 rounds of mashing. RC2 is vulnerable to a related-key attack using $2^{34}$ chosen plaintexts.

Symmetric Key Encryption Algorithms are relatively Fast - Major drawbacks with public key encryption algorithm is that they have complicated mathematics, which makes them very computationally exhaustive. Performing encryption and decryption using symmetric key is relatively fast and give very good reading and writing performance. In fact, many solid state drives, which are typically extremely fast, use symmetric key encryption algorithm internally for storing the data and they are still faster than unencrypted traditional hard drives.

### 3.3.7 Limitation and Drawback of Symmetric Key Encryption Algorithm

Mushtaque et. al. (2014) mentioned that due to small key length of 56 bits, DES does not provide strong security. DES can easily cracked by $2^{56}$ imaginations. Initially DES was considered as an algorithm with strong security. But due to Brute force attack, DES is not considered as a secure encryption algorithm. Major problem with DES is its weak key. It doesn’t protect the data against linear and differential attacks. DES runs slowly and not supports any kind of modification.

3DES remove the security problem of DES. 3DES have 3 different keys (3 x 56) of 168 bits size. In 3DES algorithm, DES process is performed 3 times with 3 different keys to provide better security. 3DES provides high level security in compare to DES. According to Alanazi et. al. (2010) due to security, Disadvantage of 3DES is that it requires 3 times more space than DES.

RC2 is a block cipher with a 64-bits block cipher with a variable key size that range from 8 to 128 bits. RC2 is vulnerable to a related-key attack using $2^{34}$ chosen plaintexts.
RC6 algorithm is more secure against differential attacks as it follows the parameter of random series of output. Linear attack can apply for 16 rounds but RC6 needs $2^{119}$ imagination of plaintext which is impossible. RC6 uses the key length in the multiple of 32 which can be extended up to 2048 bits. Singh et. al. (2014) stated that for a class of weak keys, RC6 is not able to achieve randomness up to 17 rounds; otherwise RC6 is observed as a very secure algorithm.

Compare to above algorithms AES provides high security level because of having variable length key i.e. 128, 192 or 256 bits. Mandal et. al. (2012) mentioned that AES is highly secured encryption algorithm as it cannot be cracked by attacks like Key attack, Differential attack, Square attack and improved square attack. Singh et. al. (2014) stated that AES can also protect data against future attack (collision attack). AES structure is flexible to the multiple of 64. AES has not any kind of weakness in it. Some initial rounds of AES were observed as unprotected as they can be break by square method.

So, from the above discussion it can be proved that among the selected symmetric key encryption algorithm, AES is the only algorithm which is fast and secure among all the existing algorithm with no serious weakness.

In the course of this study, a dictionary based compression algorithm was designed and developed for textual data, but that algorithm did not fulfill the objective of the proposed study. The reason being

1. It was only compression algorithm but the need was to design a joint compression and encryption algorithm.
2. The compression performance was poor as compare to the existing compression algorithms.

Due to these aforementioned reasons that compression algorithm was discarded and a new approach was proposed which combines both compression and encryption algorithm together.
3.4 Proposed Integrated approach for Compression and Encryption on Textual data

The basic principle behind this approach is that encoding of data would reduce the required storage and the encryption of the data would provide secured transmission. Further due to this integrated approach the attacks on storage can be easily avoided because the data is fully encoded and the storage space required will be less due to good compression ratio.

In this algorithm, firstly an intelligent dictionary is created from the input text data based on the unique strings and frequency, and then encoding of the input text data is done using intelligent dictionary. Finally the dictionary is encrypted in an effective way so that no one can obtain the original data back without decrypting the dictionary.

The encryption of dictionary is done based on private key (128 bit size) which is essential at the receiver to retrieve original text data back. This approach is more secure against all the attacks except the case of stolen key. This stolen key attack can be avoided with permutation-based encryption of the encoded text data.

In this approach four new techniques have been designed, namely:

1. Condition Based Huffman Encoding (CHE)
2. Complex Shuffle Encryption (CSE)
3. Complex Shuffle Decryption (CSD)
4. Condition Based Huffman Decoding (CHD)

All these techniques are discussed in detail in the next section.

The approach can be summarized as follows:

The original text data is encoded using CHE to bring more compression ratio. Dictionary which is generated from original text data is encrypted using CSE technique which includes several operations like permutation, sorting and reordering. CSE technique uses private key to encrypt the dictionary. So, at the receiver end one has - private key, encrypted dictionary and encoded text data. This private key is utilized for decryption of encrypted dictionary.
using CSD technique. Finally, encoded data is decoded using CHD technique and decrypted dictionary to obtain the original text data back. The same is illustrated in Figure 3.4 and Figure 3.5.

![Diagram](image1)

**Figure 3.4: Process at transmitter side**

![Diagram](image2)

**Figure 3.5: Process at receiver side**

**3.4.1 The Proposed CHE Technique**

This section shows the proposed Condition based Huffman Encoding (CHE) technique. The original data is encoded using condition based Huffman encoding technique. The condition based Huffman encoding technique consist of three stages –

a. Huffman code generation of original data
b. Code conversion of condition based sequence

c. Encoding.

All the above stages are explained as follows:

### 3.4.1.1 Huffman code generation

The Huffman code generation is as follows: initially the repeated letters from the document are taken and assigned the frequency value to them. Thereafter, the generation of code is done by combining the letters of least two frequencies and assigning zero’s and one’s values to it. Zero is assigned to left child and One is assigned to right child of tree. Assigning zero’s and one’s values based on two least frequency letters is done step by step until last combination. After generating zeroes and ones for every combination, the Huffman code for each letter is formed by traversing from the bottom end to the root of the tree formed.

### 3.4.1.2 Code conversion of condition based sequence

The code conversion of condition based sequence is done after generating Huffman code for each letter in the document. The process of code conversion of condition based sequence is as follows: initially the letters used to generate the Huffman code are arranged in ascending order based on the length of the code. Thereafter, the code conversion process is done by taking the letter that has highest length of Huffman code i.e. the last letter after the arrangement as constant and check with the least length letters and verifying whether the least length letter is the preceding letter of the constant letter in the original data. If yes, assign the Huffman code of the least length letter to the constant letter. If no, check with next least length letter and if no letters in the series forms the combination, there would not be any change. This process is repeated on the prior highest length Huffman coded letter.

### 3.4.1.3 Encoding

The encoding process is done based on the combination of letters used in the code conversion of condition based sequence and the preceding letter in the original data. The encoding process is as follows: initially the combination of the letters used for the code conversion process and the preceding letter of the combination of the letters used for the code conversion process is checked to decide whether the code formed using code conversion process is to be considered or not. If the Huffman code of the preceding letter of
the combination of the letters used for code conversion ends with zero, then the code formed in the code conversion process is considered and if it does not end with zero, it will not be considered. After this verification, a code is formed for the original data. The final code is the encoded data based on condition based Huffman encoding technique.

### 3.4.2 Example of CHE Technique

The whole process of condition based Huffman encoding is explained by an example as follows: consider ‘optoipoa’ is an original data. From this original data the Huffman code is generated for each letter of the original data. The formation of code is shown in Figure 3.6.

![Figure 3.6: Formation of Huffman code](image)

The Figure 3.6 is explained as follows: initially letters occurring in the text are selected and further the frequency of letters occurring in the original text is mentioned i.e. the numeric term 3 below the letter ‘o’ represents that the letter ‘o’ is repeated three times in the original data. Thereafter, two letters with least frequency (1) are taken and assigned zero and one respectively. The total frequency of two letters is represented under them in the tree. Similarly, this process is repeated by taking the next two letters and assign zeros and ones until last step. The Huffman code is then formed for each letter by considering the corresponding branches of zeros and ones from the last step to the first. From the Figure 3.6, the Huffman code formed for the letter ‘o’ is 10; the Huffman code formed for the letter ‘p’ is 00; the Huffman code formed for the letter ‘t’ is 01; the Huffman code formed for the letter ‘i’ is 110; the Huffman code formed for the letter ‘a’ is 111. Eventually, the Huffman code for the original data ‘optoipoa’ is ‘10000110110010111’. The Figure 3.7 shows the direction of Huffman code formed for each letter.
After generating the Huffman code for each letter, the code conversion of condition based sequence is done by arranging the least length letters first. The code conversion of condition based sequence process is used to compress the data.

The least length letter is obtained first from the Figure 3.7 and its code remains unchanged. Thereafter, the highest length letters are compared with the least length letters by checking whether the least length letter is the preceding letter of the highest length letter in the original data. Therefore, the highest length letter ‘a’ would be compared with the least length letter ‘o’ to check whether ‘o’ is the previous letter of ‘a’ in the original data ‘optoipoa’. From the original data it can be seen that the letter ‘o’ which has two length code comes before the letter ‘a’ which has three length code and therefore the Huffman code of ‘o’ is assigned to the letter ‘a’. Figure 3.8 shows the letters with its code after the code conversion process of the letters ‘o’ and ‘a’.

Similarly the code conversion process is done between the prior highest length letter and the least length letters. When comparing all the other letters in the original code, the letters ‘i’ and ‘p’ comes sequent in the original data but the three length coded letter ‘i’ comes before
the two length coded letter ‘p’ and therefore it has not satisfied the condition. So the code conversion will not work between the letters ‘i’ and ‘p’ of the original code.

After the code conversion process, the encoding is done based on the Huffman code of the preceding letter of the letters which are used for code conversion. The encoding is the process that fixes the code converted in the code conversion process. The code conversion is done between the sequence of letter ‘o’ and ‘a’ in the original code ‘optoipoa’. Therefore, the Huffman code of the previous letter ‘p’ is considered to fix the converted code between the letters ‘o’ and ‘a’. To fix the converted code, the value of last length of the letter ‘p’ should be zero i.e. the Huffman code of ‘p’ should ends with zero. Here, the Huffman code of ‘p’ is 00. Therefore, the code converted between ‘o’ and ‘a’ after the code conversion process is considered for encoding the data. If the Huffman code of ‘p’ ends with 1, then the code converted between ‘o’ and ‘a’ after the code conversion process would not be considered and the Huffman code obtained which is before the code conversion process is considered as encoded data. Finally, the CHE based encoded data for the original data ‘optoipoa’ is ‘1000110110001010’.

3.4.3. Dictionary formation

A dictionary is generated to retrieve the encoded data and it is sent to the receiver after encrypting it. The dictionary contains the letters considered for generating the Huffman code with its frequency in the original data. The original data taken in the illustrated example is ‘optoipoa’. Figure 3.9 shows the dictionary generated for the original data taken.

<table>
<thead>
<tr>
<th>Letters</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>3</td>
</tr>
<tr>
<td>p</td>
<td>2</td>
</tr>
<tr>
<td>t</td>
<td>1</td>
</tr>
<tr>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 3.9: Dictionary generated for the original data

The dictionary word generated from the data given is ‘optia’ that would have the frequency of each letter placed after it i.e. ‘o3p2t1i1a1’.
3.4.4 The Proposed CSE Technique

The dictionary is then encrypted using complex shuffle encryption (CSE) technique and it is transmitted to the receiver side. The complex shuffle encryption is done based on a 128 bit key. The key is split into three different divisions to do different jobs. The Figure 3.10 shows the structure of 128 bit key.

![Figure 3.10: Structure of 128 bit key](image)

In this 128 bit key, the first bit to ninety sixth bit are used for shuffling; ninety seventh bit to one hundred twelfth bit are used to convert the letter based on ASCII code; one hundred thirteenth bit to one twenty eighth bit contains the information about position of the dictionary data after retrieval process. The encryption of the dictionary word ‘o3p2t1i1a1’ with its frequency is as follows:

Initially, the dictionary word is rotated clockwise based on the length of the word i.e. if the dictionary word is of four character length, then it would be rotated clockwise four times. Thereafter, the rotated words are sorted based on the alphabet and the last letter of the sorted words is chosen to form a different word. This process is called reorder process and it is shown in Figure 3.11.

![Figure 3.11: Reorder Process of dictionary word](image)
After a different word is formed from the dictionary word, the 128 bit key is used to complete the encryption process. The three different jobs of the key are interchange process, rotation process and location identification.

### 3.4.4.1 Interchange Process (1 to 96 bits)

The zeros and ones are generated randomly in the first ninety six bits of the 128 bits key. The 96 bits are then split into twelve groups of subsequent eight bits each. For each group the eight digit binary value is converted into corresponding decimal value. That is twelve integer values are obtained. These twelve integer values are grouped into six adjoining pairs of values. These integer values in each pair are interchanged. For instance if the first integer value is 1 and the second integer value is 4, then the corresponding letters in the first and fourth position of the reordered dictionary word are interchanged i.e. the ‘itapo11132’ is formed as ‘ptaio11132’. Similarly, the letters are interchanged for all subsequent five pairs of values.

### 3.4.4.2 Rotation Process (97 to 112 bits)

The process in 97 to 112 bits is as follows: alike 1 to 96 bits, the zeros and ones are generated randomly in 97 to 112 bits and these sixteen bits are split into two groups of first eight bits and second eight bits. The binary data in both cells are converted into corresponding decimal value and using these decimal values the letters in corresponding positions are changed based on ASCII code. For instance the word obtained after 1 to 96 bits process is ‘ptaio11132’. Assuming that the integer value of the first group from 97 to 112 bits is 3, then the ASCII code of the letter in the third position is taken and added with the integer value and the solution is then converted into character. This character replaces the original character at this place. Therefore, the letter in the third position would get changed. In ‘ptaio11132’, the third letter is ‘a’ and the decimal value of ASCII is 97 which is added with 3 giving the solution as 100. The character ‘d’ has the ASCII value 100 and hence we replace ‘a’ with ‘d’. The word is now changed to ‘ptdio11132’. Similarly we assumed that the decimal value for the second group is 7. The ASCII code of the letter in the seventh position is subtracted by 7 and the character corresponding to that resultant ASCII value replaced the original character in the same position. For example the seventh letter of ‘ptdio11132’ is ‘1’ and the ASCII value of ‘1’ is 49 which is subtracted with 7
resulting in 42. Character ‘*’ corresponds to ASCII value 42 hence the character at seventh position i.e. ‘1’ gets replaced by ‘*’. Eventually the encrypted dictionary is ‘ptdio1*132’.

3.4.4.3 Location Identification (113 to 128 bits)
The 113 to 128 bits contains the information about position of the dictionary word after retrieval process. Therefore, the retrieval process is also done simultaneously with the encryption process to identify the position of the dictionary word. Table 3.1 shows the retrieval process of the dictionary word ‘o3p2t1i1a1’.

<table>
<thead>
<tr>
<th>C1</th>
<th>i</th>
<th>t</th>
<th>a</th>
<th>p</th>
<th>o</th>
<th>l</th>
<th>1</th>
<th>1</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>i</td>
<td>l</td>
<td>l</td>
<td>2</td>
<td>3</td>
<td>a</td>
<td>i</td>
<td>o</td>
<td>p</td>
</tr>
<tr>
<td>C2</td>
<td>ii</td>
<td>ti</td>
<td>ai</td>
<td>p2</td>
<td>o3</td>
<td>la</td>
<td>li</td>
<td>lo</td>
<td>lp</td>
</tr>
<tr>
<td>S2</td>
<td>ia</td>
<td>li</td>
<td>lo</td>
<td>2i</td>
<td>3p</td>
<td>al</td>
<td>il</td>
<td>o3</td>
<td>p2</td>
</tr>
<tr>
<td>C3</td>
<td>ila</td>
<td>tli</td>
<td>alo</td>
<td>p2t</td>
<td>o3p</td>
<td>la1</td>
<td>lil</td>
<td>lo3</td>
<td>lp2</td>
</tr>
<tr>
<td>S3</td>
<td>ial</td>
<td>tli</td>
<td>lo3</td>
<td>2t</td>
<td>3p2</td>
<td>alo</td>
<td>ila</td>
<td>o3p</td>
<td>p2t</td>
</tr>
<tr>
<td>C4</td>
<td>i1a1</td>
<td>tlli</td>
<td>alo3</td>
<td>p2t1</td>
<td>o3p2</td>
<td>lalo</td>
<td>lila</td>
<td>lop3</td>
<td>3p2t</td>
</tr>
<tr>
<td>S4</td>
<td>i1o3</td>
<td>lilla</td>
<td>lop3</td>
<td>2ti</td>
<td>3p2t</td>
<td>la03</td>
<td>ila1</td>
<td>o3p2</td>
<td>p2h1</td>
</tr>
<tr>
<td>C5</td>
<td>i1o3</td>
<td>lila1</td>
<td>alo3p</td>
<td>p2t1i</td>
<td>o3p2l</td>
<td>lialo3</td>
<td>lila1</td>
<td>lop32</td>
<td>3p3t1</td>
</tr>
<tr>
<td>S5</td>
<td>i1o3</td>
<td>lila1</td>
<td>alo3p</td>
<td>2t1i</td>
<td>3p2t1</td>
<td>lialo32</td>
<td>lila1o</td>
<td>lop32t</td>
<td>3p2t1i</td>
</tr>
<tr>
<td>C6</td>
<td>i1o3</td>
<td>lila1</td>
<td>alo3p</td>
<td>p2t1i</td>
<td>o3p2l</td>
<td>lialo32</td>
<td>lila1o</td>
<td>lop32t</td>
<td>3p2t1i</td>
</tr>
<tr>
<td>S6</td>
<td>i1o3</td>
<td>lila1</td>
<td>alo3p</td>
<td>2t1i</td>
<td>3p2t1i</td>
<td>lialo32</td>
<td>lila1o</td>
<td>lop32t</td>
<td>3p2t1i</td>
</tr>
<tr>
<td>C7</td>
<td>i1o3</td>
<td>lila1</td>
<td>alo3p</td>
<td>2t1i</td>
<td>3p2t1i</td>
<td>lialo32</td>
<td>lila1o3</td>
<td>lop32t</td>
<td>3p2t1i</td>
</tr>
<tr>
<td>S7</td>
<td>i1o3</td>
<td>lila1</td>
<td>alo3p</td>
<td>2t1i</td>
<td>3p2t1i</td>
<td>lialo32</td>
<td>lila1o3</td>
<td>lop32t</td>
<td>3p2t1i</td>
</tr>
<tr>
<td>C8</td>
<td>i1o3</td>
<td>lila1</td>
<td>alo3p</td>
<td>2t1i</td>
<td>3p2t1i</td>
<td>lialo32</td>
<td>lila1o3</td>
<td>lop32t</td>
<td>3p2t1i</td>
</tr>
<tr>
<td>S8</td>
<td>i1o3</td>
<td>lila1</td>
<td>alo3p</td>
<td>2t1i</td>
<td>3p2t1i</td>
<td>lialo32</td>
<td>lila1o3</td>
<td>lop32t</td>
<td>3p2t1i</td>
</tr>
<tr>
<td>C9</td>
<td>i1o3</td>
<td>lila1</td>
<td>alo3p</td>
<td>2t1i</td>
<td>3p2t1i</td>
<td>lialo32</td>
<td>lila1o3</td>
<td>lop32t</td>
<td>3p2t1i</td>
</tr>
<tr>
<td>S9</td>
<td>i1o3</td>
<td>lila1</td>
<td>alo3p</td>
<td>2t1i</td>
<td>3p2t1i</td>
<td>lialo32</td>
<td>lila1o3</td>
<td>lop32t</td>
<td>3p2t1i</td>
</tr>
<tr>
<td>C10</td>
<td>i1o3</td>
<td>lila1</td>
<td>alo3p</td>
<td>2t1i</td>
<td>3p2t1i</td>
<td>lialo32</td>
<td>lila1o3</td>
<td>lop32t</td>
<td>3p2t1i</td>
</tr>
<tr>
<td>S10</td>
<td>i1o3</td>
<td>lila1</td>
<td>alo3p</td>
<td>2t1i</td>
<td>3p2t1i</td>
<td>lialo32</td>
<td>lila1o3</td>
<td>lop32t</td>
<td>3p2t1i</td>
</tr>
</tbody>
</table>

In Table 3.1 C1 represented the reordered dictionary word ‘itapo11132’ which is sorted on ASCII value resulting in ‘11123aiotp’ represented by S1. Letters in corresponding cells of C1 and S1 are merged resulting in C2. Next the letters in C2 are sorted resulting in S2. C3 is obtained by merging the corresponding cells of C1 and S2. The process is repeated for times equal to the length of the dictionary word, ten times for the dictionary word in the example i.e. until S10. In S10, the dictionary word ‘o3p2t1i1a1’ is obtained in the eighth position, represented in italics. The position information is converted into 16 bit representation and placed from 113 to 128 bit position in the key. The encrypted data with the key and encoded data is sent to the receiver.
3.4.5 The Proposed CSD Technique

At receiver side the received encrypted data is decrypted using Complex Shuffle Decryption (CSD) technique. Decryption process is the reverse of encryption process, which is performed with the help of the key received from transmitter side. The decryption process is explained as follows: initially the two groups of eight bit each are taken from 97 to 112 bits and the reverse process of encryption is performed. During the rotation operation in encryption process decimal value of first group of bits were added with the ASCII code of the letter in the corresponding position but in decryption process same decimal value of first group of bits will be subtracted from the ASCII value of letter in the corresponding position. Similarly decimal value of second group of bits is subtracted from the ASCII code of the letter in the corresponding position during the encryption process, but in the decryption same decimal value of second group of bits will be added to the ASCII value of letter in the corresponding position. Thereafter, the twelve groups of eight bits each are taken from 1 to 96 bits to implement the reverse process. For the encryption process groups were taken from left to right (1st to 12th group) but for decryption groups were selected from right to left (12th to 1st group) in sequential order. For each group the eight digit binary value is converted into corresponding decimal value. That is twelve integer values are obtained. These twelve integer values are grouped into six adjoining pairs of values. These integer values in each pair are interchanged. For instance if the integer value of 12th group is 3 and the integer value of 11th group is 7, then the corresponding letters in the third and seventh position of the reordered dictionary word are interchanged. Similarly, the letters are interchanged for all subsequent five pairs of values. This process generates a new word for the dictionary word. To obtain the original word, location identification operation is performed as done during encryption process. Bits 113-128 of the key are used to retrieve the original dictionary word.

3.4.6 The Proposed CHD Technique

The decrypted dictionary is then used to decode the original data based on Condition Based Huffman decoding (CHD) technique. The CHD would do the reverse process of CHE. The proposed algorithm is depicted as follows:
**Input: original data**

**Output: Original data**

**Transmitted side:**

1. Start
2. Get the original data
3. For each letter in the original data
4. Check for repeated letters
5. If repeated letter exists
6. Convert the similar repeated letters as one and assign frequency value
7. Else
8. Keep the letter as it is and assign in the frequency as one
9. End if
10. End for
11. Take the letters with its frequency
12. For each two least frequency letters
13. Assign 0’s and 1’s until last combination
14. End for
15. For each letter
16. Generate Huffman code based on 0’s and 1’s from last branch to first of corresponding letter
17. End for
18. Arrange the letters in ascending order based on the length of the code
19. For each highest length letter
20. Check with each least length letter
21. If any of the least length letter comes previous to the position of the highest length letter of original data
22. Assign the code of least length letter to the highest length letter to compress the data
23. Else
24. Ignore
25. End if
26. If the code of previous letter of the least and highest length letter used for code conversion ends with zero
27. Fix the converted code for encoding
28. Else
29. Don’t consider the converted code for encoding
30. End if
31. End for
32. Compressed encoded data would be formed
33. Generate a dictionary which is used to decode the encoded data
34. Generate 128 bit key
35. Encrypt the dictionary based on CSE technique
36. Send the compressed encoded data, encrypted dictionary and key to the receiver.

The CSD algorithm is as follows:

**Input:** Encrypted dictionary  
**Output:** Dictionary word

1. Start  
2. Get the encrypted dictionary  
3. Take the two decimal values of 97 to 112 bits  
4. Take the ASCII code of the letter of the corresponding position to the first decimal  
5. Subtract the decimal value with the ASCII code and convert it to character  
6. Take the ASCII code of the letter of the corresponding position to the second decimal.  
7. Add the decimal value with the ASCII code and convert it to character  
8. Take the twelve decimal values of the first 96 bits  
9. Take two after two cells from right to left  
10. For each two cells  
11. Interchange the letters in the corresponding positions of the decimal value  
12. End for  
13. Do retrieval process  
14. Use 113 to 128 bits of the key to identify the correct position  
15. Original dictionary word would be obtained  
16. Stop
The CSE algorithm is as follows:

**Input:** Dictionary word

**Output:** Encrypted dictionary

1. Start
2. Get the dictionary word
3. Generate 128 bit key
4. Take the first 96 bits from the 128 bit key
5. For each bit
6. Generate 0’s and 1’s randomly
7. End for
8. Split 8 bits as one cell and therefore we would get 12 cells
9. For each cell
10. Convert binary to decimal
11. End for
12. Take two after two cells from left to right
13. For each two cells
14. Interchange the letters in the corresponding positions of the decimal value
15. End for
16. Take 97 to 112 bits
17. For each bit
18. Generate 0’s and 1’s
19. End for
20. Split 8 bits as one cell and therefore 2 cells would be obtained
21. Convert binary to decimal
22. Take the ASCII code of the letter of the corresponding position to the first decimal value
23. Add the decimal value with ASCII code and convert it to character and place the new character on the same position
24. Take the ASCII code of the letter of the corresponding position to the second decimal value
25. Subtract the decimal value with ASCII code and convert it to character and place the new character on the same position
26. The 113 to 128 bits contain the information about the position of the original dictionary based on the retrieval process.

27. Encrypted dictionary would be obtained based on the 128 bit key.

28. Stop