Abstract

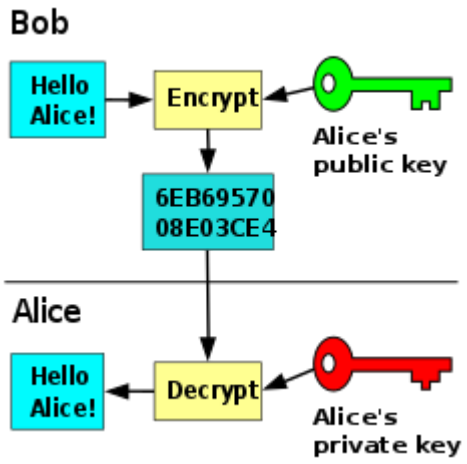
Data is any type of stored digital information. Security is about the protection of assets. Data security refers to protective digital privacy measures that are applied to prevent unauthorized access to computers, databases and websites. Cryptography is evergreen and developments. Cryptography protects users by providing functionality for the encryption of data and authentication of other users. Compression is the process of reducing the number of bits or bytes needed to represent a given set of data. It allows saving more data. The project aims to implement various cryptography algorithm for data security. The data will be first compressed using compression techniques and then encryption techniques will applied and then comparative analysis will be carried out for different combinations of compression and encryption techniques. If encryption and compression are done at the same time then it takes less processing time and more speed.

1. INTRODUCTION

Need of security is to ensuring that your information remains confidential and only access to authorized user and ensure that no one has been able to change your information, so it provide full accuracy. To secure the data, compression is used because it use less disk space (saves money), more data can be transfer via internet. It increase speed of data transfer from disk to memory. Security goals for data security are Confidential, Authentication, Integrity, and Non-repudiation. Data security delivers data protection across enterprise. 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 code word . 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 . Currently compression and encryption methods are done separately . The major problem existing with the current compression and encryption methods is the speed, the processing time required by a computer, more cost . To overcome this disadvantage, combine the two processes into one

2. CRYPTOGRAPHY

To hide any data two techniques are mainly used one is Cryptography other is Steganography. In this paper we use Cryptography. Cryptography is the science of protecting data, which provides methods of converting data into unreadable form, so that Valid User can access Information at the Destination . Cryptography is the science of using mathematics to encrypt and decrypt data .



3. COMPRESSION

Data compression implies sending or storing a smaller number of bits. Compression is the reduction in size of data in order to save space or transmission time [5]. Many methods are used for this purpose, in general these methods can be divided into two broad categories: Lossy and Lossless methods. Lossy Compression generally used for compress an Images. In this original data is not identical to compressed data that means there is some loss e.g. Block Truncation Coding, Transform Coding, etc... Lossless Compression used for compress any textual data. In this original data and compressed data are equal that means there is no loss e.g. Run Length Coding, Huffman Coding, LZW, Arithmetic Coding.

4. METHODOLOGIES

4.1 Compression Techniques

4.1.1 Run Length Encoding

Run-length encoding is the simplest method of compression. It can be used to compress data made of any combination of symbols [7]. The idea behind this method is to reducing the size of a repeating string of characters. If data have more than two consecutive characters then this method would give better result.

For example, consider a screen containing plain black text on a solid white background. There will be many long runs of white pixels in the blank space, and many short runs of black pixels within the text. A hypothetical scan line, with B representing a black pixel and W representing white, might read as follows:

WWWWWWWWWWWWWBWWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWW
WWWWWWWWWBWWWWWWWWWWWWWW
With a run-length encoding (RLE) data compression algorithm applied to the above

hypothetical scan line, it can be rendered as follows:

12W1B12W3B24W1B14W
This can be interpreted as a sequence of twelve Ws, one B, twelve Ws, three Bs, etc..,

The run-length code represents the original 67 characters in only 18. While the actual format used for the storage of images is generally binary rather than ASCII characters like this, the principle remains the same. Even binary data files can be compressed with this method; file format specifications often dictate repeated bytes in files as padding space. However, newer compression methods such as DEFLATE often use LZ77-based algorithms, a generalization of run-length encoding that can take advantage of runs of strings of characters (such as BWWBWWBWWBWW).

Run-length encoding can be expressed in multiple ways to accommodate data properties as well as additional compression algorithms. For instance, one popular method encodes run lengths for runs of two or more characters only, using an "escape" symbol to identify runs, or using the character itself as the escape, so that any time a character appears twice it denotes a run. On the previous example, this would give the following:

WW12BWW12BB3WW24BWW14
This would be interpreted as a run of twelve Ws, a B, a run of twelve Ws, a run of three Bs, etc. In data where runs are less frequent, this can significantly improve the compression rate.

One other matter is the application of additional compression algorithms. Even with the runs extracted, the frequencies of different characters may be large, allowing for further compression; however, if the run lengths are written in the file in the locations where the runs occurred, the presence of these numbers interrupts the normal flow and makes it harder to compress. To overcome this, some run-length encoders separate the data and escape symbols from the run lengths, so that the two can be handled independently. For the example data, this would result in two outputs, the string "WWBWWBBWWBWW" and the numbers (12,12,3,24,14).

4.1.2 Huffman Coding

A Commonly used method for data compression is Huffman coding. The Huffman algorithm is based on statistical coding, which means that the more probable the occurrence of a symbol is, the shorter will be its bit-size representation [8]. In any file, certain characters are used more than others. Using binary representation, the number of bits required to represent each character depends upon the number of characters that have to be represented [8].

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the

frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bit stream.

Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

There are mainly two major parts in Huffman Coding

1) Build a Huffman Tree from input characters.

2) Traverse the Huffman Tree and assign codes to characters.

4.1.3 LZW

LZW compression is a lossless compression. It compress a file into a smaller file using a table-based lookup algorithm invented by Abraham Lempel, Jacob Ziv, and Terry Welch. It is a ‘dictionary based’ compression algorithm that scan a file for sequences of data that occur more than once [6]. These sequences are then stored in a dictionary and references are put where-ever repetitive data occurred [6].

Encoding

A high level view of the encoding algorithm is shown here:

1. Initialize the dictionary to contain all strings of length one.
2. Find the longest string W in the dictionary that matches the current input.
3. Emit the dictionary index for W to output and remove W from the input.
4. Add W followed by the next symbol in the input to the dictionary.
5. Go to Step 2.

A dictionary is initialized to contain the single-character strings corresponding to all the possible input characters (and nothing else except the clear and stop codes if they're being used). The algorithm works by scanning through the input string for successively longer substrings until it

finds one that is not in the dictionary. When such a string is found, the index for the string without the last character (i.e., the longest substring that *is* in the dictionary) is retrieved from the dictionary and sent to output, and the new string (including the last character) is added to the dictionary with the next available code. The last input character is then used as the next starting point to scan for substrings.

In this way, successively longer strings are registered in the dictionary and made available for subsequent encoding as single output values. The algorithm works best on data with repeated patterns, so the initial parts of a message will see little compression. As the message grows, however, the compression ratio tends asymptotically to the maximum (i.e., the compression factor or ratio improves on an increasing curve, and not linearly, approaching a theoretical maximum inside a limited time period rather than over infinite time).

Decoding

The decoding algorithm works by reading a value from the encoded input and outputting the corresponding string from the initialized dictionary. In order to rebuild the dictionary in the same way as it was built during encoding, it also obtains the next value from the input and adds to the dictionary the concatenation of the current string and the first character of the string obtained by decoding the next input value, or the first character of the string just output if the next value can not be decoded (If the next value is unknown to the decoder, then it must be the value that will be added to the dictionary this iteration, and so its first character must be the same as the first character of the current string being sent to decoded output). The decoder then proceeds to the next input value (which was already read in as the "next value" in the previous pass) and repeats the process until there is no more input, at which point the final input value is decoded without any more additions to the dictionary.

In this way the decoder builds up a dictionary which is identical to that used by the encoder, and uses it to decode subsequent input values. Thus the full dictionary does not need to be sent with the encoded data; just the initial dictionary containing the single-character strings is sufficient (and is typically defined beforehand within the encoder and decoder rather than being explicitly sent with the encoded data.)

4.1.4 Arithmetic Coding

Arithmetic coding is a form of entropy encoding used in lossless data compression. Arithmetic coding, which is a method of generating variable-length codes, is useful when dealing with sources with small alphabets such as binary sources [9]. It encodes data (the data string) by creating a code string which represents a fractional value on the number line between 0 and 1. Replace the entire input with a single floating-point number.

Encoding and decoding: overview

In general, each step of the encoding process, except for the very last, is the same; the encoder has basically just three pieces of data to consider:

- The next symbol that needs to be encoded
- The current interval (at the very start of the encoding process, the interval is set to [0,1], but that will change)
- The probabilities the model assigns to each of the various symbols that are possible at this stage (as mentioned earlier, higher-order or adaptive models mean that these probabilities are not necessarily the same in each step.)

The encoder divides the current interval into sub-intervals, each representing a fraction of the current interval proportional to the probability of that symbol in the current context. Whichever interval corresponds to the actual symbol that is next to be encoded becomes the interval used in the next step.

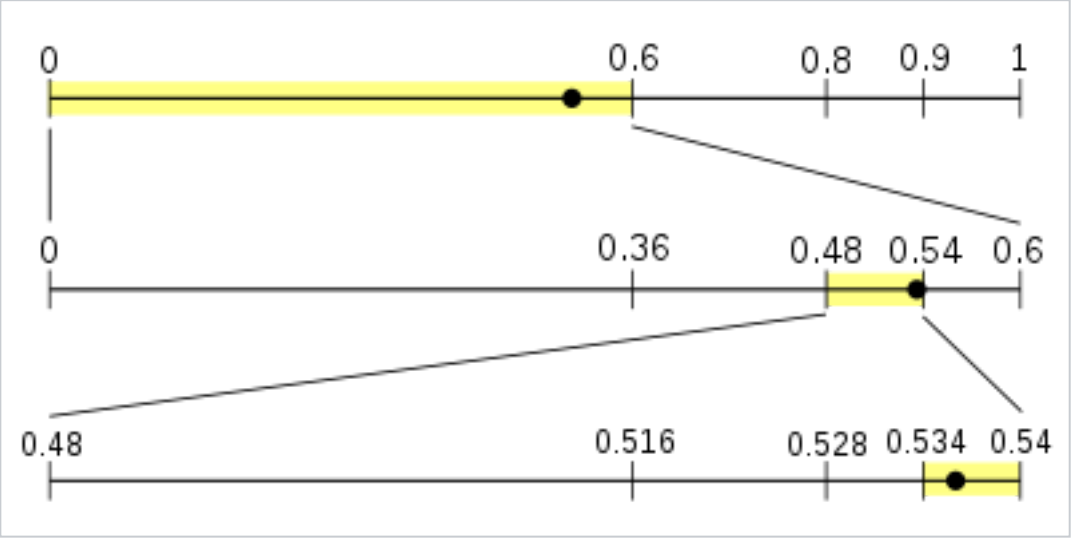
Example: for the four-symbol model above:

- the interval for NEUTRAL would be [0, 0.6)
- the interval for POSITIVE would be [0.6, 0.8)
- the interval for NEGATIVE would be [0.8, 0.9)
- the interval for END-OF-DATA would be [0.9, 1).

When all symbols have been encoded, the resulting interval unambiguously identifies the sequence of symbols that produced it. Anyone who has the same final interval and model that is being used can reconstruct the symbol sequence that must have entered the encoder to result in that final interval.

It is not necessary to transmit the final interval, however; it is only necessary to transmit *one fraction* that lies within that interval. In particular, it is only necessary to transmit enough digits (in whatever base) of the fraction so that all fractions that begin with those digits fall into the final interval; this will guarantee that the resulting code is a prefix code.

Encoding and decoding: example



A diagram showing decoding of 0.538 (the circular point) in the example model. The region is divided into subregions proportional to symbol frequencies, then the subregion containing the point is successively subdivided in the same way.

Consider the process for decoding a message encoded with the given four-symbol model. The message is encoded in the fraction 0.538 (using decimal for clarity, instead of binary; also assuming that there are only as many digits as needed to decode the message.)

The process starts with the same interval used by the encoder: [0,1), and using the same model, dividing it into the same four sub-intervals that the encoder must have. The fraction 0.538 falls into the sub-interval for NEUTRAL, [0, 0.6); this indicates that the first symbol the encoder read must have been NEUTRAL, so this is the first symbol of the message.

Next divide the interval [0, 0.6) into sub-intervals:

- the interval for NEUTRAL would be [0, 0.36), *60% of [0, 0.6)*.
- the interval for POSITIVE would be [0.36, 0.48), *20% of [0, 0.6)*.
- the interval for NEGATIVE would be [0.48, 0.54), *10% of [0, 0.6)*.
- the interval for END-OF-DATA would be [0.54, 0.6), *10% of [0, 0.6)*.

Since .538 is within the interval [0.48, 0.54), the second symbol of the message must have been NEGATIVE.

Again divide our current interval into sub-intervals:

- the interval for NEUTRAL would be [0.48, 0.516).
- the interval for POSITIVE would be [0.516, 0.528).
- the interval for NEGATIVE would be [0.528, 0.534).
- the interval for END-OF-DATA would be [0.534, 0.540).

Now 0.538 falls within the interval of the END-OF-DATA symbol; therefore, this must be the next symbol. Since it is also the internal termination symbol, it means the decoding is complete. If the stream is not internally terminated, there needs to be some other way to indicate where the stream stops. Otherwise, the decoding process could continue forever, mistakenly reading more symbols from the fraction than were in fact encoded into it.

4.2 Cryptographic Techniques

4.2.1 RC4 (Rivest Cipher)

RC4 design by Ron Rivest of RSA Security in 1987. It is stream cipher, Symmetric key encryption. RC4 kept as a trade secret by RSA Security earlier. But someone anonymously posted RC4 code on internet it was a big loss. After that RSA security tell that it is still a trade secret but it was too late. The algorithm is used for both encryption and decryption as the data stream is simply XORed with the generated key sequence . It uses a variable length key from 1 to 256 bit to initialize a 256-bit state table .

RC4 generates a pseudorandom stream of bits (a keystream). As with any stream cipher, these can be used for encryption by combining it with the plaintext using bit-wise exclusive-or; decryption is performed the same way (since exclusive-or with given data is an involution). This is similar to the one-time pad except that generated *pseudorandom bits*, rather than a prepared stream, are used.

To generate the keystream, the cipher makes use of a secret internal state which consists of two parts:

1. A permutation of all 256 possible bytes (denoted "S" below).
2. Two 8-bit index-pointers (denoted "i" and "j").

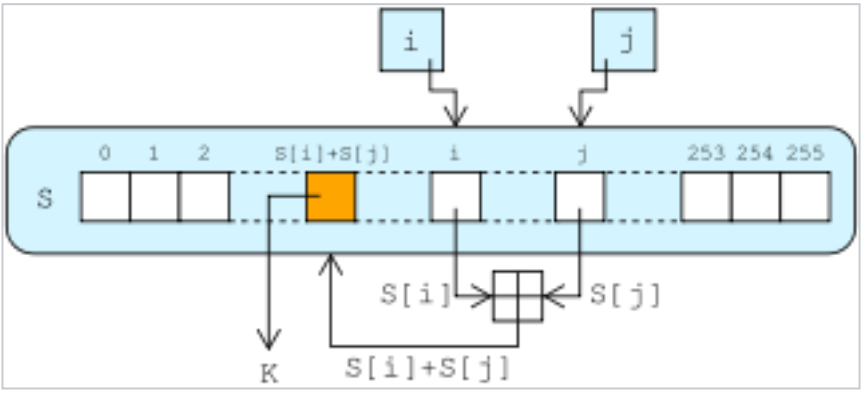
The permutation is initialized with a variable length key, typically between 40 and 2048 bits, using the *key-scheduling* algorithm (KSA). Once this has been completed, the stream of bits is generated using the *pseudo-random generation algorithm* (PRGA).

Key-scheduling algorithm (KSA)

The key-scheduling algorithm is used to initialize the permutation in the array "S". "keylength" is defined as the number of bytes in the key and can be in the range 1 ≤ keylength ≤ 256, typically between 5 and 16, corresponding to a key length of 40 – 128 bits. First, the array "S" is initialized to the identity permutation. S is then processed for 256 iterations in a similar way to the main PRGA, but also mixes in bytes of the key at the same time.

```
for i from 0 to 255
    S[i] := i
endfor
j := 0
for i from 0 to 255
    j := (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

Pseudo-random generation algorithm (PRGA)



The lookup stage of RC4. The output byte is selected by looking up the values of S[i] and S[j], adding them together modulo 256, and then using the sum as an index into S; S(S[i] + S[j]) is used as a byte of the key stream, K.

For as many iterations as are needed, the PRGA modifies the state and outputs a byte of the keystream. In each iteration, the PRGA:

- increments *i*
- looks up the *i*th element of S S[*i*], and adds that to *j*
- exchanges the values of S[*i*] and S[*j*] then uses the sum S[*i*] + S[*j*] (modulo 256) as an index to fetch a third element of S (the keystream value K below)
- then bitwise exclusive ORed (XORed) with the next byte of the message to produce the next byte of either ciphertext or plaintext.

Each element of S is swapped with another element at least once every 256 iterations.

```
i := 0
j := 0
while GeneratingOutput:
   i := (i + 1) mod 256
   j := (j + S[i]) mod 256
   swap values of S[i] and S[j]
   K := S[(S[i] + S[j]) mod 256]
   output K
endwhile
```

4.2.2 Caesar Cipher

Caesar Cipher is named after Julius Caesar. It is one of the simplest and most widely known encryption techniques. It shifts the alphabet. The key is the number of letters you shift. It is a Substitution Cipher that involves replacing each letter of the secret message with a different letter of the alphabet which is a fixed number of positions further in the alphabet .

Example

The transformation can be represented by aligning two alphabets; the cipher alphabet is the plain alphabet rotated left or right by some number of positions. For instance, here is a Caesar cipher using a left rotation of three places, equivalent to a right shift of 23 (the shift parameter is used as the key):

```
Plain:   ABCDEFGHIJKLMNOPQRSTUVWXYZ
Cipher:  XYZABCDEFGHIJKLMNOPQRSTUVW
```

When encrypting, a person looks up each letter of the message in the "plain" line and writes down the corresponding letter in the "cipher" line.

```
Plaintext:  THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG
Ciphertext: QEB NRFZH YOLTK CLU GRJMP LSBO QEB IXWV ALD
```

Deciphering is done in reverse, with a right shift of 3.

The encryption can also be represented using modular arithmetic by first transforming the letters into numbers, according to the scheme, A → 0, B → 1, ..., Z → 25. Encryption of a letter *x* by a shift *n* can be described mathematically as,

$$E_n(x) = (x + n) \mod 26.$$

Decryption is performed similarly,

$$D_n(x) = (x - n) \mod 26.$$

(There are different definitions for the modulo operation. In the above, the result is in the range 0 to 25; i.e., if $x + n$ or $x - n$ are not in the range 0 to 25, we have to subtract or add 26.)

4.2.3 DES

The Data Encryption Standard (DES) is based on a symmetric-key algorithm that uses a 56-bit key. DES is a block cipher, which means that during the encryption process, the plaintext is broken into fixed length blocks and each block is encrypted at the same time [13]. DES consists of 16 steps, each of which called as a Round.

Result

Input size		
280bytes	huffman(90)	rc4(63)
280bytes	lwz(145)	rc4(146)
280bytes	runlength(354)	rc4(355)
28000	huffman(9037)	rc4(9038)
28000	lwz(2939)	rc4(2941)
28000	runlength(35400)	rc4(35401)

Conclusion

LWZ is good in large files become we can find patterns which are repeating over time whereas huffmann is performing well when file size is small because there are very less repeating patterns.