SELECTIVE ENCRYPTION WITH TEXT FILES

Abstract

Selective encryption is the technique of encrypting some parts of a compressed data file while leaving others unencrypted.

Selective encryption is not a new idea. It has been proposed in several applications, especially in the commercial multimedia industry. However, selective encryption of losslessly compressed text files has not been explored, and that is the focus of our project. Through the project, we will carefully study how selective encryption can achieve a high level of effectiveness. By this, I mean a strategy in which even a small fraction of encrypted bits can cause a high ratio of damage to a file if an attacker attempts to decode it without decrypting the secured portions.

In this project, we combined the encrypting and compressing processes to consider the choices of which types of bits are most effective in the selective encryption sense when they are changed. And so, instead of encrypting the whole file bit by bit, we changed only these highly sensitive bits. Moreover, by combining the compression and encryption tasks and reducing the total encryption work required, we can achieve a savings in system complexity.

We used Huffman coding as the compression scheme for the text files. There are other algorithms for data compression and we intend to examine them in future projects. To measure the damage inflicted on a text file when it is decoded without decryption, we are using the Levenshtein distance (D_{SID}) – the minimum number of substitution, insertion and deletion operations that are needed to make two files identical.

We first performed some experiments with nearly trivial codes in order to build a full understanding of the effect of different choices of which bits to encrypt as described by their location in the Huffman decoding tree. In particular, we measured the percentage of bits encrypted, observed the occurrence of the errors, and measured their contributions to the increase of D_{SID} of the decoded file compared to the original text file. Based on this, we are trying to find either optimal rules or good heuristics for effective selective encryption schemes, applied to text files.

Key words

Codeword, encryption, compression, alphabet, security, frequency, Huffman coding, Levenshtein distance.

Introduction

As security is an increasing public concern these days, encryption is becoming popular for any type of sensitive information. An effective encryption scheme that saves the cost

and time for data encrypting will be the need of the government, organizations, business companies or individuals. Selective encryption therefore has been proposed for this purpose.

Selective encryption suggests the technique that selectively encrypts just some parts of the compressed file while guaranteeing the security of the original data file. Our strategy for selective encryption here will nest the encrypting process into the encoding process while compressing a data file. With this arrangement, we are not only saving the time for encrypting the file, but also the cost of system complexity. Figure 1 and 2 explain the concept of selective encryption.

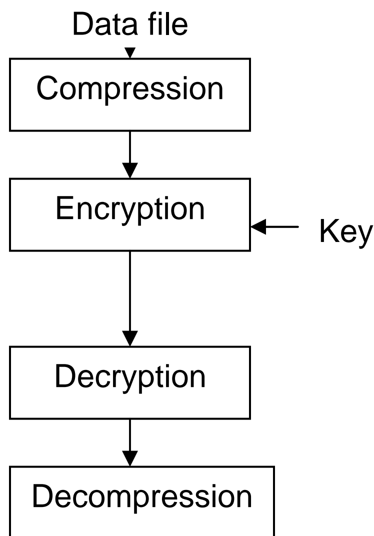

Figure 1

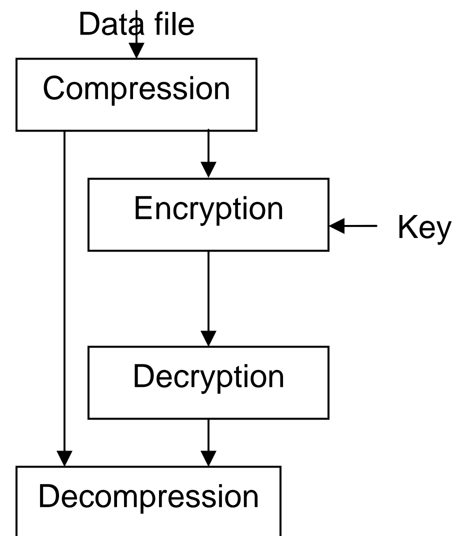

Figure 2

With text files, selective encryption can be applied in the same manner. In this project, we've used Huffman coding algorithm do the compression task. The encryption task was then added in when the compressing program does the encoding process. The key point of an effective selective encryption scheme was the selection of which type of bits that is encrypted. The effectiveness of the technique was evaluated by the value of D_{SID}.

Project background

1. Huffman coding algorithm

Huffman coding algorithm was introduced by David Huffman in 1952 and has been widely applied for data compression. The algorithm narrows the alphabet for the file based on the pattern of that particular data file and assigns the code for each character of the alphabet depending on the frequency of occurrence of that character. It assigns shorter code to the frequently used characters and longer code to the less-frequently used ones. Huffman coding therefore reduces the number of bits used for each high-frequency

character while may increase this number for low-frequency character. These assignments results in the compression of the file to about 20 to 40%. Huffman coding therefore is fix-to-variable compression scheme. Figure 3 and 4 below illustrates a simple example of how Huffman coding works.

Char	Codeword
A	0
C	100
B	101
F	1100
E	1101
D	111

Figure 3

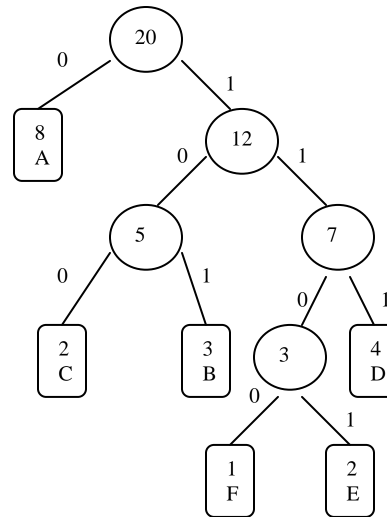

Figure 4

The number at each leaf nodes is the frequency of occurrence associated with the character in that node. By taking two lowest-frequency nodes to form an internal node who is assigned the frequency of the total frequencies of the two children, the Huffman tree is built up. The process continues until there is only one node. This node is the root of the tree.

To get the codeword for each character, we traverse the tree from root to leaf containing that character and assign 0 when going to the left or 1 when going to the right.

There are several algorithms that are used for data compression: Huffman coding, Lempel Ziv, Arithmetic coding. In this project, we used Huffman coding and studied Huffman tree - the tree to derive to the codeword for each character.

We started the project by first implementing the Huffman coding algorithm. For experimental purpose, our implementation separated the frequency table from the compressed file, so that the length of bits used for the table would not interfere with the length of the file being compressed. The encoder, being implemented this way, inputted a text file, compressed it and outputted the compressed file, as well as the frequency table in form of a text file separately. The decoder was implemented to read the frequency file and to build up the Huffman tree in the same manner as the encoder to transform the compressed file back to the original one.

The implementation for Huffman coding was first obtained from Ammeraal's code. It was then revised and added some more functions to perform the encryption and other tasks in our experiments. The general arrangement of the compressed file after being encoded, however, was kept as its origin.

2. Levenshtein distance (D_{SID})

Proposed by Vladimir Levenshtein in 1965, this algorithm is the measurement of the similarity between two strings (of which I will call the source string and the target string). The algorithm calculates the minimum number of substitution, insertion and deletion operations that are needed to transform the source string to the target string. For example, the two strings "aaab" and "aacba" have D_{SID} of 2 because we need to substitute the third 'a' and insert an 'a' at the end of the source string to transform it to the target string.

In our project, D_{SID} was used as the measurement of the damage of a text file after being encrypted. The encrypted file and the original file were considered as two long strings of characters of which the encrypted file was the source and the original file was the target. The value of D_{SID} reflexes the damage that would occur in the decoded file if an attacker tried to decode the encrypted file without decrypting it. (We assumed that attacker had the decoder which would create the same Huffman tree as the encoder would and could get back the original file if the compressed file wasn't encrypted.)

Strategy

By analyzing the Huffman tree, we noticed that an internal node at the high level of the Huffman tree could play more important role than the lower ones. Our hypothesis was encrypting bits for some internal node choices that were more effective (higher D_{SID} per encrypted bit) than others.

We term each of our strategies of flipping bits that are derived from each internal node as one type of system corresponding to one type of internal node. Since there is (n – 1) internal nodes if there is n leaf nodes, the number of different types of internal nodes is (n – 1). Our assignment of the type of systems is from the root node (type I) to leaf and from left to right increasingly. Type-I internal node is therefore on one level higher than type II (there is always one root which is of type I); while type II and type III or other types are not always in different levels.

We used the pseudorandom number as the cryptography key to manipulate the encryption task. In our experiments, the probability of flipping bits is 50%

Experiments

1. The simple cases

Our first experiments were with the simple case of 50% of 'A', 25% of 'B' and 25% of 'C'. The Huffman tree in this case would be as in figure 5 and figure 6 below.

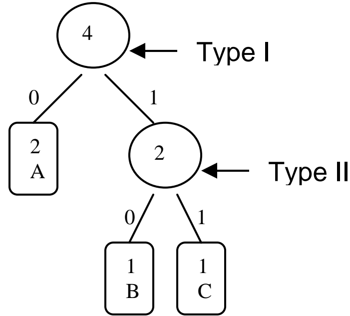

Figure 5

Char	Codeword
A	0
B	10
C	11

Figure 6

The string "AABBC", for example, would be encoded as "00101011".

Our expectation for an effective system was type I system because type I system would effect all kinds of characters, while type II system would not effect the highest-probability character. For this case, the Huffman tree for the flipped bits with type I system would be:

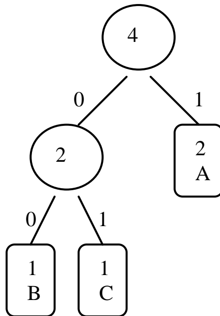

Figure 7

Char	Codeword
A	1
B	00
C	01

Figure 8

The string "AABBC" is now encoded and encrypted as "11000001". It is therefore decoded (without being encrypted) as "CAAAAAA". The last "1" is left over because there is no such code of only "1". D_{SID} in this case is 5.

With type II system, the Huffman tree for the flipped bits with type I system would be:

Char	Codeword

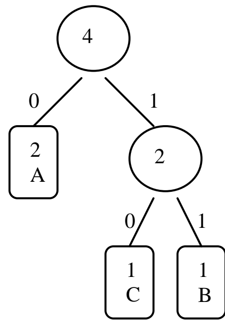

Figure 9

A	0
B	11
C	10

Figure 10

The above string is now encoded and encrypted as “00111110”. Note that code word for characters “B” and “C” are swapped with each other and so “B” and “C” are swapped when they are decoded (without being encrypted). That is “AACCB” and has D_{SID} of 3 which is 2 smaller than this value in type-I case.

But when we experimented with large number of characters (1000 characters, as we did), the complexity in the alignments of characters made D_{SID} different from our anticipations.

2. The real text cases

In the real text cases, the results were encouraging. With the ratio of encryption of about 10%, we could achieve the ratio of damage to the file of nearly 80% in some cases. Although we could not give a conclusion base upon those cases, they promised the potential of selective encryption with the ratio of encryption as low as 10%.

Conclusion

We have not yet reached a conclusion; the project is still going on. However, we’ve finished the implementations of the project’s base algorithms and started carrying out some experiments with some simple cases and some particular text files. The results diverged from our expectations in some simple cases due to the complexities in the alignments of characters when calculating D_{SID}. We realized that the samples for these cases were too short for the results to approach our calculations. By using a sparser experimental set-up, we believe we will be able to validate a match between our theory and experimental results, and will then be able to confidently proceed to evaluate more complicated sources and codes.

The results from our experiments with some real text files, however, were very encouraging. With a ratio of encryption of only 10 to 20 %, we observed damage to an attacker’s unauthorized decoding on the order of 70 – 85 % in 10 different text files that were picked randomly.

Appendixes

1. Attached here are some results from some of our experiments.
2. The C++ implementation of Huffman coding below will show the way we nest the compressing and encrypting process together.

```
#include<limits.h>
#include"huffman.h"

struct TableItem {
    unsigned code, len;
};

void traverse( HuffmanTree &Tr, TableItem *t, int p, unsigned code, int len ) {
    if( Tr.tree[p].left == nil ) {
            uchar ch = Tr.tree[p].right;
            t[ch].code = code;
            t[ch].len = len;
    }
    else {
            code <<= 1; len ++;
            traverse( Tr, t, Tr.tree[p].left, code, len );
            traverse( Tr, t, Tr.tree[p].right, code | 1, len );
    }
}

int main() {

    ifstream ifile;
    ofstream ofile;
    ofstream fout;
    fout.open( "frequency.txt", ios::out );
    cout << "Huffman file compression.\n";
    GetStreams( ifile, ofile );
    ulong freq[sizeFreq], count = 0;
    uchar ch;
    int j, k, b;
    for( j = 0; j < sizeFreq; j++ ) freq[j] = 0;
    while( ifile.get( ch ), !ifile.fail() )     {
            freq[ch]++;
            count++;
    }
    cout << "The length of the original file is: " << count << " bytes" <<endl;
    ifile.clear(); ifile.seekg( 0, ios::beg );

    HuffmanTree Tr( freq );
    TableItem t[sizeFreq];
    for( int i = 0; i < sizeFreq; i++ ) {     t[i].code = t[i].len = 0; fout << i << " "     <<
freq[i] << endl;     }
    traverse( Tr, t, Tr.root, 0, 0 );
    uchar buf = 0;
    k = 0;
    while( ifile.get( ch ), !ifile.fail() ) {
            for( int i = t[ch].len - 1; i >= 0; i-- ) {
                    b = ( t[ch].code >> i ) & 1;
                    if( k < 8 ) { buf = ( buf << 1 ) | b; k++; }
                     else {
                            ofile.put( buf );          buf = b;          k = 1;
                    }
            }
    }
    ofile.seekp( 0, ios::end );
    unsigned long LengthFile = ofile.tellp();
    if( k != 0 ) {
            ofile.put( char( buf << ( 8 - k ) ) );
```

```
            ofile.put( char( k ) );
    }
    cout << "The length of the compressed file is: " << LengthFile * 8 + k << " bits"
<< endl;
    return 0;
}
```

References:

1. http://www.cs.duke.edu/csed/poop/huff/info/.
2. Adam Drozdek. Data Structure and Algorithms in C++. PWS Publishing Co.: Boston, MA. 1996.
3. Leendert Ammeraal. Algorithm and Data Structures in C++. John Wiley & Sons Ltd.: NY. 1996
4. Lelewer, Debra A. and Daniel S. Hirschberg. "Data Compression". AMC Computing Survey. Vol. 19, No. 3. September 1987.
5. Nelson, Mark. The Data Compression Book. M & T Publishing Inc.: NY. 1992.