Simultaneous Data Compression and Encryption

Chung-E Wang

Department of Computer Science
California State University, Sacramento
Sacramento, CA 95819-6021

Abstract. This paper describes cryptographic methods for concealing information during data compression processes. These include novel approaches of adding pseudo random shuffles into the processes of dictionary coding (Lampel-Ziv compression), arithmetic coding, and Huffman coding.

Keywords. data compression, encryption.

1. Introduction

Data compression is known for reducing storage and communication costs. It involves transforming data of a given format, called source message, to data of a smaller sized format, called codeword.

Data encryption is known for protecting information from eavesdropping. It transforms data of a given format, called plaintext, to another format, called cipher text, using an encryption key.

The major problem existing with the current compression and encryption methods is the speed, i.e. the processing time required by a computer. To lessen the problem, I combine the two processes into one.

The basic idea of combining the two processes is to add a pseudo random shuffle into a data compression. The method of using a pseudo random number generator to create a pseudo random shuffle is well known. A simple algorithm as below can do the trick. Assume that we have a list (x_1, x_2, ... x_n) and we want to shuffle it randomly.

```
for i = n downto 2 {
    k = random(1,i);
    swap $x_i$ and $x_k$;
}
```

2. Adding a Pseudo Random Shuffle to a Dictionary Coding (Lampel-Ziv compression)

The basic idea of Lampel-Ziv (LZ) compression is to replace a group of consecutive characters with an index into a dictionary that is built during the compression process. There are many implementations of the LZ compression. Different implementations of the LZ compression have different ways of implementing the dictionary. For further discussion of LZ compressions, refer to [1].

FIG. 1 illustrates the steps of combining a random shuffle with a LZ compression to achieve the simultaneous data compression and encryption. In step 110, the encryption key is used to initialize a pseudo random number generator. In step 120, the pseudo random number generator is used to shuffle the initial values of the dictionary.

In a codebook type of implementation, e.g. LZW compression, i.e. Welch's implementation of the LZ compression, the dictionary consists of strings of characters. Initially, it contains all strings of length 1 in alphabetical order. In this case, step 120 shuffles strings of length 1. So, the dictionary begins with strings of length 1 in random order. For a further discussion of LZW compression, refer to [2]

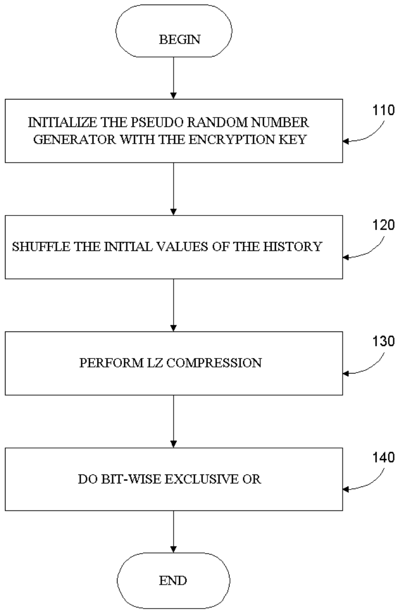

BEGIN
INITIALIZE THE PSEUDO RANDOM NUMBER GENERATOR WITH THE ENCRYPTION KEY
110
SHUFFLE THE INITIAL VALUES OF THE HISTORY
120
PERFORM LZ COMPRESSION
130
DO BIT-WISE EXCLUSIVE OR
140
END

Figure 1

In a sliding window type of implementation, e.g. LZ77, the dictionary is a window that consists of last n characters processed. Initially, the window is empty. In this case, step 120 initializes the window with the set of all characters of the alphabet and then shuffles the window. For a further discussion of LZ77, refer to [1].

In step 130, the compression process is performed on the input string in its usual fashion.

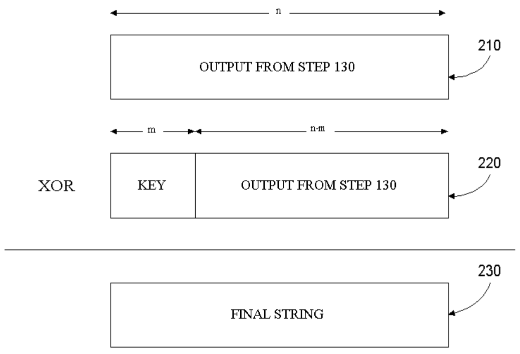


Figure 2

In step 140, the mathematical bit-wise exclusive OR (XOR) operation is performed between the output of step 130 and the concatenation of the encryption key and the output of step 130. FIG. 2 shows the detail of step 140. Assume that the length of the encryption key is *m* and the length of the output of step 130 is *n*. Block 210 is the output of step 130. Block 220 is the concatenation of the encryption key and the first *n-m* characters of the output of step 130. Block 230 is the result of performing the bit-wise XOR between blocks 210 and 220. Block 230 is the final compressed and encrypted string.

Note that the purpose of step 140 is to guard against the chosen plaintext attack. This step is necessary only for those implementations in which an un-identical shuffle results in a finished decompression with a resultant string different from the original uncompressed string. Also note that in an actual implementation, steps 130 and 140 can be done together in one loop.

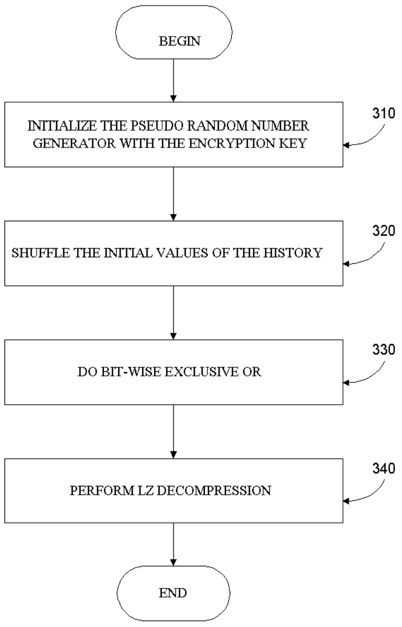

BEGIN
INITIALIZE THE PSEUDO RANDOM NUMBER GENERATOR WITH THE ENCRYPTION KEY
310
SHUFFLE THE INITIAL VALUES OF THE HISTORY
320
DO BIT-WISE EXCLUSIVE OR
330
PERFORM LZ DECOMPRESSION
340
END

Figure 3

FIG. 3 illustrates the steps of simultaneous decompression and decryption. In step 310, the encryption key is used to initialize a pseudo random number generator. Note that the pseudo random number generator used in step 310 should be identical to the one used in step 110. In step 320, the pseudo random number generator is used to shuffle the initial values of the dictionary. In step 330, the bit-wise XOR is performed on the input string and the encryption key as in FIG. 4. In step 340, the decompression is performed on the output of step 330 in its usual fashion. The output of step 340 is the final decompressed and decrypted string.

In FIG. 4, block 410 is the input string. Logically, block 420 is the concatenation of the encryption key and block 430. However, block 430 is the result of performing the bit-wise XOR operation between blocks 410 and 420. In other words, blocks 420 and 430 depend on each other and thus must be built gradually. First, the bit-wise XOR is performed between the encryption key and the corresponding portion in block 410 to produce SEG1 in block 430. Then the bit-wise XOR is performed between the SEG1 of block 420 and the corresponding portion in block 410 to produce SEG2, etc. Block 430 is the output of step 330.

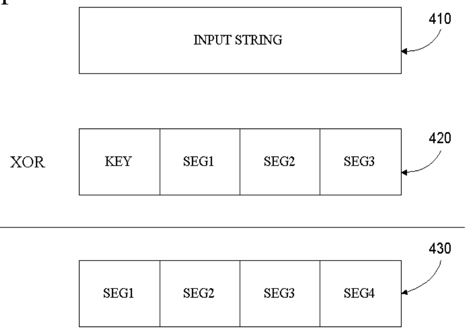


Figure 4

3. Adding a Pseudo Random Shuffle to the Arithmetic Coding

In the arithmetic coding, a message of any length is coded as a real number between 0 and 1. The longer the message the more precision is used to code the message. This is done as follows:

a) Initialize the current interval with the interval [0,1), i.e. the set of real numbers from 0 to 1, including 0 and excluding 1.
b) Divide the current interval into smaller intervals such that each character has a corresponding smaller interval with a length proportional to its probability.
c) From these new intervals, choose the one corresponding to the next character in the message to be coded as the new current interval.
d) Continue to do steps b) and c) until the whole message is coded.
e) Represent the interval' s value using a binary fraction.

character	probability	interval
A	0.24	[0.00, 0.24)
B	0.12	[0.24, 0.36)
C	0.15	[0.36, 0.51)
D	0.18	[0.51, 0.69)
E	0.31	[0.69, 1.00)

(a) Interval table before C is coded

character	probability	interval
A	0.24	[0.36, 0.396)
B	0.12	[0.396, 0.414)
C	0.15	[0.414, 0.4365)
D	0.18	[0.4365, 0.4635)
E	0.31	[0.4635, 0.51)

(b) Interval table before A is coded

character	probability	interval
A	0.24	[0.36, 0.36864)
B	0.12	[0.36864, 0.37296)
C	0.15	[0.37296, 0.37836)
D	0.18	[0.37836, 0.38484)
E	0.31	[0.38484, 0.396)

(c) Interval table before B is coded

Figure 5

FIG. 5 shows an example. The message to be coded is “CAB”. Probabilities of characters are repeated in all three tables. Part (a) shows the intervals before the coding of the first character ‘C’. Part (b) shows the intervals before the coding of the second character ‘A’. Part (c) shows the intervals before the coding of the third character ‘B’. The number 0.36864 is the final result of the arithmetic coding. For further discussion of arithmetic coding, refer to [3, 4].

The basic idea of concealing information in the process of arithmetic coding is to use an encryption key to shuffle the interval table before the coding process. Without the encryption key, the interval table cannot be shuffled in the same way and the division of an interval into smaller intervals won’t be the same and thus decompression cannot be done properly. Consequently, the original information cannot be retrieved

FIG. 6 illustrates the steps of combining a random shuffle with the arithmetic coding. In step 610, the encryption key is used to initialize a pseudo random number generator. In step 620, the pseudo random number generator is used to shuffle the interval table. In step 630, the arithmetic coding process is performed on the input message in its usual fashion.

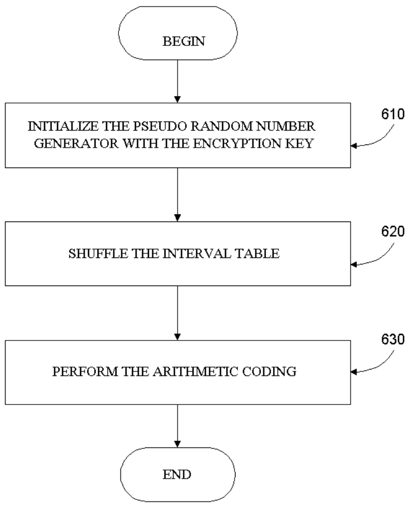

BEGIN
INITIALIZE THE PSEUDO RANDOM NUMBER GENERATOR WITH THE ENCRYPTION KEY
610
SHUFFLE THE INTERVAL TABLE
620
PERFORM THE ARITHMETIC CODING
630
END

Figure 6

FIG. 7 shows the effect of a pseudo random shuffle. Part (a) shows the intervals before the coding of the first character 'C'. Part (b) shows the intervals before the coding of the second character 'A'. Part (c) shows the intervals before the coding of the third character 'B'. The number 0.0477 is the final result of the arithmetic coding.

4. Adding a Pseudo Random Shuffle to the Huffman Coding

Huffman coding is a compression algorithm introduced by David Huffman in 1952. The basic idea of Huffman coding is to construct a tree, called a Huffman tree, in which each character has it' s own branch determining its code.

A Huffman coding could be static or adaptive. In a static Huffman coding, the Huffman tree stays the same in the entire coding process. In an adaptive Huffman coding, the Huffman tree changes according to the data processed. For further discussion about static and adaptive Huffman coding, refer to [5, 6, 7, 8, 9, 10].

character	probability	interval
C	0.15	[0.00, 0.15)
A	0.24	[0.15, 0.39)
E	0.31	[0.39, 0.70)
B	0.12	[0.70, 0.82)
D	0.18	[0.82, 1.00)

(a) Interval tabl before C is coded

character	probability	interval
C	0.15	[0.00, 0.0225)
A	0.24	[0.0225, 0.0585)
E	0.31	[0.0585, 0.105)
B	0.12	[0.105, 0.123)
D	0.18	[0.123, 0.15)

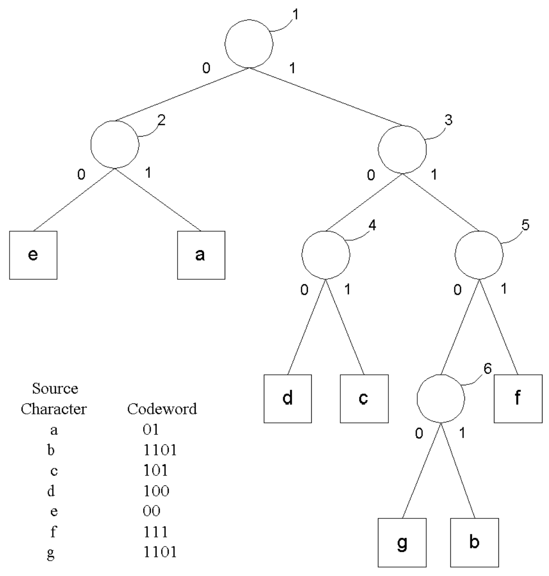
(b) Interval table before A is coded

character	probability	interval
C	0.15	[0.0225, 0.0279)
A	0.24	[0.0279, 0.03654)
E	0.31	[0.03654, 0.0477)
B	0.12	[0.0477, 0.05202)
D	0.18	[0.05202, 0.0585)

(c) Interval table before B is coded

Figure 7

Once the Huffman tree is built, regardless of it being static or adaptive, the encoding process is identical. The codeword for each source character is the sequence of labels along the path from the root to the leave node representing that character. For example, in FIG. 8, the codeword for 'a' is '01', 'b' is '1101', etc.



Source Character	Codeword
a	01
b	1101
c	101
d	100
e	00
f	111
g	1101

Figure 8

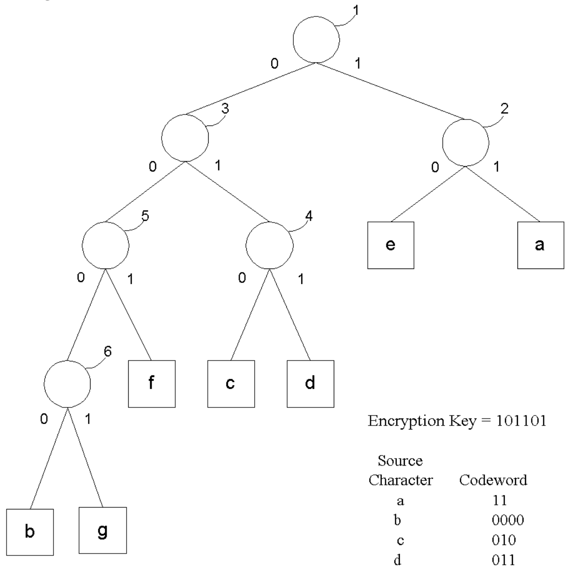
The basic idea of concealing information in Huffman coding is to use an encryption key to shuffle the Huffman tree before the encoding process. Without the encryption key, the Huffman tree cannot be shuffled in the same way and thus the decompression cannot be done properly. Consequently, the original information cannot be retrieved.

To shuffle a Huffman tree, first, the interior nodes i.e. nodes with 2 children, are numbered. There are many ways of numbering these interior nodes. For example, by performing a queue traversal on the Huffman tree, the interior nodes can be numbered in the top-down, left-right fashion. In FIG. 8, the labeling of the interior nodes shows the top-down, left-right numbering of the interior nodes.

Secondly, bits of the encryption key are associated with the interior nodes according to the numbering; the interior node 1 is associated with the first bit of the encryption key, the interior node 2 is associated with the second bit of the encryption key, etc. Finally, of each interior node that has a corresponding encryption bit of 1, the left child is swapped with the right child. In FIG. 9, the encryption key used is "101101". Thus, the two children of interior nodes 1, 3, 4, and 6 are swapped. After the shuffling, the codeword of source characters are changed dramatically and cannot be decoded without the identically shuffled Huffman tree.

seed of the pseudo random number generator. Since there are 256! (factorial of 256) different permutations for 8-bit characters, the maximum size of the seed of the pseudo random number generator could be as big as $\log_2 256!$ which is equivalent to a 2048-bit encryption key.

The encryption strength of the method of combining a random shuffle with a Huffman coding depends on the length of the encryption key. Since a Huffman tree can have at most 255 interior nodes, the maximum effective key length of the third method is 255.



Encryption Key = 101101

Source Character	Codeword
a	11
b	0000
c	010
d	011
e	10
f	001
g	0001

Figure 9

5. Conclusions

Besides the obvious execution time advantage of combining the two processes of data compression and encryption, the encryption strengths of our methods are as good as any other encryption algorithms such as DES [11], triple DEA [12], and RC5 [13].

In our first two methods, i.e. combining random shuffles with dictionary coding and arithmetic coding, the encryption key is only used to initialize the pseudo random number generator. Thus, the strength of the resulting encryption doesn't depend on the length of the encryption key. Instead, the strength depends on the size of an internal variable called the

6. References

[1] Ziv, J. and Lampel A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* 23, 3 (May), 337-343.

[2] Welch, T. A. 1984. A technique for high-performance data compression. *Computer* 17, 6 (June), 8-19.

[3] Witten, I.H., Neal, R.M., and Cleary, J.G. 1987. Arithmetic coding for data compression. *Communications of the ACM*, vol. 30, 520-540.

[4] Moffat, A., Neal, R.M., and Witten, I.H. 1995. Arithmetic coding revisited. *ACM Transactions on Information Systems*, vol. 16, 256-294.

[5] Cormack, G.V., and Horspool, R.N. 1984.

Algorithms for Adaptive Huffman Codes. *Inform. Process. Lett*. 18, 3 (Mar.), 159-165.

[6] Faller, N. 1973. An adaptive system for data compression. In Record of the 7th Asilomar Conference on Circuits, Systems and Computers (Pacific Grove, Calif., Nov.). Naval Postgraduate School, Monterey, Calif., pp. 593-597.

[7] Huffman, D.A. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proc. IRE* 40, 9 (Sept.), 1098-1101.

[8] Knuth, D.E. 1985. Dynamic Huffman Coding. *J. Algorithms* 6, 2 (June), 163-180.

[9] Gallager, R.G. 1978. Variations on a theme by Huffman. *IEEE Trans. Inf. Theory* 24, 6 (Nov.), 668-674.

[10] Vitter, J.S. 1987. Design and analysis of dynamic Huffman codes. *J. ACM* 34, 4 (Oct.), 825-845.

[11] National Bureau of Standards, Data Encryption Standard, FIPS PUB 46 (January 1977).

[12] Tuchman, W. 1979. Hellman presents no shortcut solutions to DES. *IEEE Spectrum*, 16, 7(July) 40-41.

[13] Rivest, R. 1995. The RC5 Encryption Algorithm. *Dr. Dobb's Journal*, v20, n1 (January).