Secure arithmetic coding encryption schemes

T. Uehara

University of Wollongong, takeyuki@uow.edu.au

UNIVERSITY OF WOLLONGONG
COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

This work is copyright. Apart from any use permitted under the Copyright Act 1968, no part of this work may be reproduced by any process, nor may any other exclusive right be exercised, without the permission of the author.

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.

Unless otherwise indicated, the views expressed in this thesis are those of the author and do not necessarily represent the views of the University of Wollongong.

Recommended Citation

Secure Arithmetic Coding Encryption Schemes

A thesis submitted in partial fulfillment of the requirements for the award of the degree

Master of Science (Honors)

from

UNIVERSITY OF WOLLONGONG

by

Takeyuki Uehara

Department of Computer Science
July 1999
Declaration

This is to certify that the work reported in this thesis was done by the author, unless specified otherwise, and that no part of it has been submitted in a thesis to any other university or similar institution.

________________________
Takeyuki Uehara
July 21, 1999
Abstract

Image processing is of growing interest in computer applications. Since the size of image files are generally large and image data is usually highly redundant, using compression algorithms provide effective way of increasing communication and storages efficiency. JPEG, which is one of the most widely used image compression standards, uses Huffman or arithmetic coding for its lossless compression part. With rapid growth of the Internet, controlling access to data is of increasing importance and hence encryption is of much wider use. Adding security to the algorithm is an attractive proposal as it could reduce the overall processing cost of providing secure compressed data.

A number of methods for combining encryption and compression had been proposed and various attacks on these proposals were published. This thesis reviews the known proposals for arithmetic coding encryption schemes and examines proposed attacks on such systems. We extend the attacks, introduce new ones, and finally propose efficient methods of enhancing security of these systems.
I would like to thank my supervisor Dr. Rei Safavi-Naini for guiding and encouraging me throughout this project. I would also like to thank Dr. Philip Ogonbuna and Dr. Xing Zhang for their interest in this project. The work of the author is partially supported by Motorola Australian Research Centre (MARC).
## Contents

Abstract iii

Acknowledgements iv

1 Introduction 1
   1.1 Motivation .................................................. 1
   1.2 Objective .................................................. 2
   1.3 Contributions and structure of the thesis .................. 2
   1.4 Notations .................................................. 4

2 Arithmetic coding 5
   2.1 Information theory ........................................... 5
   2.2 Data compression ............................................ 6
      2.2.1 Source coding .......................................... 7
      2.2.2 Optimal codes .......................................... 7
      2.2.3 Constructions of optimal codes ......................... 8
      2.2.4 Data compression models ................................ 9
      2.2.5 PPM models ............................................ 10
   2.3 Security ................................................... 10
      2.3.1 Symmetric encryption system ............................ 10
      2.3.2 Attacks ................................................ 11
      2.3.3 Redundancy of a language ............................... 11
      2.3.4 Unicity distance ....................................... 12
      2.3.5 Data compression and security .......................... 12
   2.4 Arithmetic coding .......................................... 13
      2.4.1 Encoding .............................................. 13
      2.4.2 Decoding .............................................. 14
      2.4.3 Advantage of arithmetic coding ......................... 15
2.4.4 Models in arithmetic coding ............................................. 15
2.5 Implementation of arithmetic coding .................................. 16
  2.5.1 The model ................................................................. 16
  2.5.2 The coder ................................................................. 17
2.6 Conclusion ........................................................................ 20

3 Arithmetic coding encryption schemes .................................. 22
  3.1 Arithmetic coding encryption schemes ................................. 22
    3.1.1 Model-based schemes .................................................. 23
    3.1.2 Coder-based schemes .................................................. 23
    3.1.3 A combined scheme .................................................... 24
    3.1.4 Initialization and reset of the encoder ............................ 25
    3.1.5 Effect on data compression performance ......................... 25
    3.1.6 Security of the schemes .............................................. 26
  3.2 Attack on non-adaptive arithmetic coding encryption scheme .. 27
  3.3 Attacks on adaptive schemes ............................................. 27
    3.3.1 Bergen/Hogan attack .................................................. 27
    3.3.2 Lim/Boyd/Dawson attack .............................................. 29
  3.4 Attack on coder-based scheme ......................................... 30
  3.5 Conclusion ........................................................................ 30

4 New Attacks .......................................................................... 32
  4.1 Strengthening BH attack .................................................... 32
    4.1.1 Detection of the halving point ....................................... 33
    4.1.2 Attacks on the coder .................................................... 39
  4.2 Attack on a combined scheme ............................................. 46
    4.2.1 Outline of the attack .................................................... 46
    4.2.2 Security of the shrinking factors ................................... 48
    4.2.3 A method to derive the shrinking factors from the output ... 49
    4.2.4 Analysis of the output bit sequence ................................. 52
    4.2.5 Cost of the attack ........................................................ 56
  4.3 Conclusion ........................................................................ 56

5 Protection against attacks ................................................... 57
  5.1 The properties used in the attacks .................................... 57
    5.1.1 Repeated experiments .................................................. 57
5.1.2 Controlling the model by an input message .................. 58
5.1.3 Controlling the coder by an input message .................. 58
5.1.4 Analysis of the output considered as a real number value ... 58
5.1.5 Analysis on the output size .................................. 59
5.1.6 Properties exploited to attack systems ...................... 59

5.2 Protection against the attacks .................................. 60
5.2.1 Reset/initialization .......................................... 60
5.2.2 Randomizing the halving periods ............................ 61
5.2.3 Extra-narrowing of interval .................................. 62
5.2.4 Higher precision of the model and the coder ............... 63
5.2.5 Higher order of the model ................................... 63
5.2.6 XORing the output with pseudo-random sequence .......... 64
5.2.7 Blocking ...................................................... 65
5.2.8 Permutation ................................................... 65

5.3 Conclusion ....................................................... 66

6 Comments on integrated encryption after compression schemes 67
6.1 Methods to hide encoder’s output ............................. 67
6.1.1 Models to hide the true output ............................... 67
6.2 Comparison between Type 1 and Type 2 ....................... 68
6.2.1 Speed ......................................................... 68
6.2.2 Security ....................................................... 69
6.3 Pure compression versus compression-encryption schemes .... 70
6.3.1 Controlling input messages of encryption algorithm ....... 70
6.3.2 Resistance against statistical attacks ...................... 70
6.3.3 Efficiency ..................................................... 71
6.4 Conclusion ....................................................... 71

7 Conclusion .......................................................... 72
7.1 Schemes .......................................................... 72
7.2 Attacks .......................................................... 73
7.3 Weaknesses and security ........................................ 73
7.4 Further work ..................................................... 74

Bibliography .......................................................... 75
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Example calculation of the interval for a message ABC</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of output of unknown bits</td>
<td>21</td>
</tr>
<tr>
<td>4.1</td>
<td>Example of $\Delta o_i$ values</td>
<td>37</td>
</tr>
<tr>
<td>4.2</td>
<td>Average bits/symbol of 128 symbol block : bit-oriented output</td>
<td>38</td>
</tr>
<tr>
<td>4.3</td>
<td>Average bits/symbol of 128 symbol block : byte-buffered output</td>
<td>39</td>
</tr>
<tr>
<td>4.4</td>
<td>Groups of ranges with 12 bit output</td>
<td>44</td>
</tr>
<tr>
<td>4.5</td>
<td>Changing ranges of mid-range buffering with 12 bit output</td>
<td>44</td>
</tr>
<tr>
<td>4.6</td>
<td>The change of the range (0x0000,0xBFFF) with 14 bit output</td>
<td>45</td>
</tr>
<tr>
<td>4.7</td>
<td>The change of the range (0x0000,0xFFFF) with 14 bit output</td>
<td>45</td>
</tr>
<tr>
<td>4.8</td>
<td>The change of the range (0x4000,0xFFFF) with 14 bit output</td>
<td>46</td>
</tr>
<tr>
<td>5.1</td>
<td>Halving periods and compression ratios (bits/symbol)</td>
<td>62</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Communication system ........................................... 5
2.2 Example of Huffman code ........................................... 8
2.3 Data compression model ........................................... 10
2.4 Model of symmetric encryption system ......................... 11
2.5 Example of encoding a message "ABC" ......................... 14
2.6 Implementation of the frequency table ......................... 16
2.7 Output the range .................................................. 17
2.8 Possible states of a range ........................................ 18
2.9 State transition of the procedures for State 1, 2 and 3 ..... 19
2.10 Special case of the range : Example 1 ....................... 20
2.11 Special case of the range : Example 2 ....................... 20
2.12 Example of how to fix the unknown bit (1) .................. 20
2.13 Example of how to fix the unknown bit (2) .................. 21

3.1 LFB scheme ......................................................... 24
3.2 Procedure of LFB scheme ......................................... 25
3.3 Example of BH attack ............................................. 28

4.1 Change in the output size for $N_{symbol} = 256$ ............ 34
4.2 Example of average output size of 128 symbols ................ 37
4.3 Relation between the difference of range and output ......... 41
4.4 One cycle of encoding procedure ................................ 49
4.5 Bit patterns in an output without mid-range buffering ....... 54
4.6 Algorithm to find the start bit for a symbol in an output .. 55

5.1 Encryption of output using random number sequence ...... 64

6.1 Models to hide the true output .................................. 68
6.2 Speed of Type 1 system ......................................... 69
6.3 Speed of Type 2 system .................................................. 69

B.1 Smoothing of the bits/symbol values ................................. 89
Image processing is of growing interest in computer applications. In publishing fields, computerized desktop publishing is commonly used and due to the attractiveness of the visual presentations, digital images are widely used on the World Wide Web. High performance computers have become more affordable and multimedia applications are in great demand. Since the required file sizes for images are generally much larger than that of texts, data compression is of high importance for enabling efficient use of storage and communication channels for digital images.

1.1 Motivation

To provide digital images usable in many applications, several standards are proposed. JPEG (Joint Photographic Experts Group) standard, officially Recommendation T.81 by CCITT, is one of the most widely used standards [ITU93]. It provides the specifications of processes for the conversion between source image data and compressed image data, and the specification of coded representations for compressed image data and the guidelines of implementation of the processes. It uses two different classes of compression. That is, lossy and lossless compression. In JPEG, the lossless compression follows the lossy compression, which uses DCT (Discrete Cosine Transform). For the lossless compression, JPEG specifies Huffman and arithmetic coding as its entropy encoder (decoder) to compress (decompress) the DCT coefficients.

Arithmetic coding is an optimal data compression algorithm originally proposed by Pasco and Rissanen [BWC90]. Arithmetic coding encodes a message into a bit string, which represents a real number interval in the interval [0, 1) [BWC90]. Arithmetic coding provides a flexible alternative to the commonly used Huffman compression algorithm and in many applications gradually replaces the latter.

Encryption algorithms protect data against unauthorized access. With rapid growth
of the Internet, controlling access to data is of increasing importance and hence encryption is of much wider use. Digital images are no exception. It can be proved (Section 2.3.5) that compressing plaintext before applying the encryption algorithm effectively increases the security of the overall system. If a data compression algorithm can be made to also provide security, less processing overhead could be expected as a single algorithm achieves two goals.

Witten and Cleary proposed a number of methods for combining encryption and compression [WC88]. Subsequently various attacks on their proposals were published which clearly showed the need for careful security assessment of any such proposal. The results also showed that obtaining security without sacrificing compression ratio and processing speed is not an easy task.

1.2 Objective

Our objective is to investigate compression-encryption schemes applied to digital images. The question addressed in this research is whether it is possible to increase efficiency, measured in terms of the processing time, by integrating the two processes. Our research concentrates on arithmetic encoding encryption schemes.

It is worth noting that such combination will only be successful if the resulting system
1. halving points, the point where the frequencies of the symbols in the model are halved and play a crucial role in Bergen/Hogan attack (BH attack for short)[BH93]. We demonstrate efficient ways of detecting the halving point by observing the output of the encoder. We also show various methods of determining the range when the model is known. The methods are efficient and in one case the number of possible values for range is reduced to two. Finally we give a summary of common weak points of arithmetic coding encryption schemes and propose methods of strengthening the system.
The thesis is organized as follows. In Chapter 2, we give an overview of information theory, data compression and security, and then describe details of how arithmetic coding works. In Chapter 3, we look at arithmetic encoding encryption schemes and review the known attacks on arithmetic coding encryption schemes. In Chapter 4, we demonstrate various new attacks on arithmetic coding encryption schemes. In Chapter 5, we look at weaknesses of arithmetic coding encryption schemes and propose protections against the attacks. In Chapter 6, we comment on the combined compression and encryption scheme proposed in Chapter 5. Finally in Chapter 7 we summarize our results.
### 1.4 Notations

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSB</td>
<td>the most significant bit</td>
</tr>
<tr>
<td>LSB</td>
<td>the least significant bit</td>
</tr>
<tr>
<td>MSBs</td>
<td>the most significant bits</td>
</tr>
<tr>
<td>LSBs</td>
<td>the least significant bits</td>
</tr>
<tr>
<td>( \psi )</td>
<td>a symbol</td>
</tr>
<tr>
<td>( \psi[i] )</td>
<td>a symbol at the ( i^{th} ) position from the top of the frequency table</td>
</tr>
<tr>
<td>( \psi_i )</td>
<td>the ( i^{th} ) input symbol (in a message)</td>
</tr>
<tr>
<td>( \psi_{\text{max}} )</td>
<td>a symbol at the top of the frequency table</td>
</tr>
<tr>
<td>( \psi_{\text{min}} )</td>
<td>a symbol at the bottom of the frequency table excluding the EOF symbol (note: the EOF symbol is at the very bottom of the table)</td>
</tr>
<tr>
<td>( N_{\text{symbol}} )</td>
<td>the number of symbols</td>
</tr>
<tr>
<td>( p_\psi )</td>
<td>the probability of a symbol ( \psi )</td>
</tr>
<tr>
<td>( c_l(\psi) )</td>
<td>the lower cumulative probability of a symbol ( \psi )</td>
</tr>
<tr>
<td>( c_h(\psi) )</td>
<td>the upper cumulative probability of a symbol ( \psi )</td>
</tr>
<tr>
<td>( F(\psi) )</td>
<td>the frequency of a symbol ( \psi )</td>
</tr>
<tr>
<td>( F(\psi_{\text{max}}) )</td>
<td>the frequency of a symbol ( \psi_{\text{max}} )</td>
</tr>
<tr>
<td>( F_{\text{max}} )</td>
<td>the maximum frequency of a symbol ( \psi_{\text{max}} ) when ( C_h(\psi_{\text{max}}) = C_{\text{max}} ) and all other frequencies are 1</td>
</tr>
<tr>
<td>( C_{\text{max}} )</td>
<td>the maximum cumulative frequency (When ( C_l(\psi_{\text{max}}) ) reaches ( C_{\text{max}} ), all frequencies are halved.)</td>
</tr>
<tr>
<td>( C_l(\psi) )</td>
<td>the lower cumulative frequency of a symbol ( \psi )</td>
</tr>
<tr>
<td>( C_h(\psi) )</td>
<td>the upper cumulative frequency of a symbol ( \psi )</td>
</tr>
<tr>
<td>( h_i )</td>
<td>the high value at the beginning of the encoding procedure of ( i^{th} ) symbol</td>
</tr>
<tr>
<td>( l_i )</td>
<td>the low value at the beginning of the encoding procedure of ( i^{th} ) symbol</td>
</tr>
<tr>
<td>( M_i )</td>
<td>the MSBs of the high and low values which are the same</td>
</tr>
<tr>
<td>( O )</td>
<td>the order of a model</td>
</tr>
<tr>
<td>( S_i(x, u) )</td>
<td>the shift function for the output procedure of a coder where ( i \in {0, 1} )</td>
</tr>
<tr>
<td>( o_i )</td>
<td>the ( i^{th} ) output of a coder</td>
</tr>
<tr>
<td>( u )</td>
<td>the output size (for a symbol)</td>
</tr>
</tbody>
</table>
In this chapter we examine arithmetic coding and its implementations. Firstly, we briefly review the information theory, data compression systems and security. Next, we examine arithmetic coding scheme, outline various implementations and describe one implementation in detail.

2.1 Information theory

Communication system

A communication system, Fig. 2.1, consists of a message source, an encoder, a channel, and a decoder. The message source produces messages to be transmitted. The encoder performs source coding, which is to convert the messages into a form suitable for the transmission. The channel is the medium through which the encoded messages are transmitted. The noise may interfere with the communication over the channel. The decoder recovers the encoded messages to their original form for the receiver of the information.

![Communication system diagram]

Figure 2.1: Communication system

Uncertainty and entropy

Information is related to uncertainty. In a communication system, a message is transmitted from an information source. In a discrete source, a message consists of symbols, chosen from an alphabet set. When the source emits a message, a message is chosen
2.2 Data compression

from the set of all possible messages. Once a symbol is chosen, the uncertainty about it is removed.

If the message source is modeled as a discrete random variable, uncertainty can be measured as follows [BP82]. Let \( X \) denote a discrete random variable that takes values \( x_i, 1 \leq i \leq M \), with probabilities \( p(x_i) \). Then the average uncertainty is given by

\[
H_M(X) = -\sum_{i=1}^{M} p(x_i) \log p(x_i).
\] (2.1)

This is called entropy of the random variable.

\( H(X) \) can be interpreted as the average amount of information obtained after observing \( X \). For example, assume that in an experiment a fair die is rolled. Each number appears with equal chance and so the probability of each number from 1 to 6 is \( p(X = 1) = p(X = 2) = p(X = 3) = p(X = 4) = p(X = 5) = p(X = 6) = \frac{1}{6} \). Then \( H(X) \) is given by

\[
H(X) = -\sum_{i=1}^{6} \frac{1}{6} \log_2 \frac{1}{6} = -\log_2 \frac{1}{6} \approx 2.58 \text{ bits.}
\]

Before rolling the die, the average uncertainty over the experiment is 2.58 bits. After the outcome of the experiment is known 2.58 bits of information is obtained and the uncertainty is removed.

When we have a pair of discrete random variables \( X \in \{x_1, x_2, \ldots, x_M\} \) and \( Y \in \{y_1, y_2, \ldots, y_L\} \), with joint probability distribution \( p(x_i, y_j), 1 \leq i \leq M \) and \( 1 \leq j \leq L \), the joint entropy of \( X \) and \( Y \) is given by

\[
H(X, Y) = -\sum_{i=1}^{M} \sum_{j=1}^{L} p(x_i, y_j) \log p(x_i, y_j).
\] (2.2)

Joint entropy satisfies the condition \( H(X, Y) \leq H(X) + H(Y) \) with equality if and only if \( X \) and \( Y \) are independent. The conditional entropy of \( X \) and \( Y \) is given by

\[
H(Y|X) = -\sum_{i=1}^{M} p(x_i) \sum_{j=1}^{L} p(y_j|x_i) \log p(y_j|x_i),
\] (2.3)

and satisfies the condition \( H(Y|X) \leq H(X) \) with equality if and only if \( X \) and \( Y \) are independent.

2.2 Data compression

Data compression is an outgrowth of information theory. The aim of data compression is to find a short description for a message source.

For a channel with a given capacity compressing messages results in a more efficient use of the channels. There are two types of compression algorithms: lossless compression and lossy compression. Our purpose is to assess the security of arithmetic
coding so we only look at the lossless compressions. In lossless compression systems the compressed data can be used to recover an exact replica of the source output. In lossy compression only approximate form of the source output can be recovered.

### 2.2.1 Source coding

The information source may be continuous or discrete. In the former case the source output can be represented by a continuous time signal while in the latter case, the output is represented by a string of symbols from a finite set. In source coding the encoder encodes the output produced by the source into a sequence of codewords. The length of codewords may be fixed or variable. An example of a fixed length code is ASCII code. Morse code is an example of a variable length code. By assigning shorter codewords to more frequent symbols, a more efficient encoding is achieved.

### 2.2.2 Optimal codes

An optimal code is the most efficient code for a given message source. Let the source be represented by a discrete random variable, $X$, that takes value from the set $\{x_1, x_2, ..., x_M\}$. In source coding each symbol is encoded into a codeword, which is a sequence of symbols from another alphabet set, $A = \{a_1, a_2, ..., a_D\}$. Let the length of the codeword for $x_i$ be $l_i$. Assuming that the channel is noiseless, then the most efficient code is the one that has minimum average codeword length.

$$\bar{l} = \sum_{i=1}^{M} p(x_i) l_i$$  \hspace{1cm} (2.4)

An important property of a code is that it should be uniquely decodable. A code is uniquely decodable if any encoded string has a unique source string correspond to it. A prefix code is a code such that no codeword is a prefix of any other codeword and it is uniquely decodable. Kraft Inequality is a necessary and sufficient condition that must be satisfied by a prefix code [Kra49]. For any prefix code over an alphabet $A$, the Kraft inequality is:

$$\sum_{i=1}^{M} D^{-l_i} \leq 1.$$  \hspace{1cm} (2.5)

The relationship between the entropy and the average codeword length is

$$H(X) \leq \bar{l} \log D.$$  \hspace{1cm} (2.6)
Equality holds if and only if \( p(x_i) = D^{-i}, \forall i \); in this case, \( H(X) = \bar{l} \log D \). There exist \( D \)-ary (alphabet consisting of \( D \) symbols) prefix codes which satisfy

\[
\frac{H(X)}{\log D} \leq \bar{l} \leq \frac{H(X)}{\log D} + 1.
\]  (2.7)

The prefix code with the shortest average length is called *optimal* prefix code.

### 2.2.3 Constructions of optimal codes

**Huffman code**

A method of constructing an optimal prefix code is given by Huffman [Huf52]. Assume we have a set of symbols \( X = \{x_1, x_2, ..., x_M\} \) with probabilities \( p(x_1), p(x_2), ..., p(x_M) \), and let \( p(x_1) \geq p(x_2) \geq ... \geq p(x_M) \). Assume there are \( D \) symbols in the code alphabet. Then the algorithm is as follows.

1. Combine \( D \) symbols with the smallest probabilities to construct a symbol \( x_{M-D+1}, ..., x_M \) with probability \( \sum_{i=M-D+1}^{M} p(x_i) \), and replace \( x_{M-D+1}, ..., x_M \) by the new symbol. Repeat the procedure until the number of symbols becomes \( D \).

2. Assign a single codeword symbol to each symbol.

3. If there is any symbol obtained from combining \( D \) symbols, separate it to \( D \) symbols and append each codeword symbol to the codeword assigned to the combined symbol. Repeat the procedure until all original symbols are separated.

An example is shown in Fig 2.2. The symbol set is \( X = \{1, 2, 3, 4, 5\} \) with probabilities 0.25, 0.25, 0.2, 0.15, 0.15 respectively and \( D = 2 \).

<table>
<thead>
<tr>
<th>( X )</th>
<th>( p(X) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.25</td>
</tr>
<tr>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>3</td>
<td>0.2</td>
</tr>
<tr>
<td>4</td>
<td>0.15</td>
</tr>
<tr>
<td>5</td>
<td>0.15</td>
</tr>
</tbody>
</table>

**Figure 2.2: Example of Huffman code**
2.2. Data compression

Shannon-Fano-Elias coding

Shannon-Fano-Elias coding encodes source symbols using cumulative distribution to assign codewords. It is not an optimal coding algorithm but forms the basis of arithmetic coding, to be discussed in Section 2.4, that is an optimal coding algorithm. Let \( X \) be a set of source symbols, where \( X = \{1, 2, \ldots, M\} \) and the probabilities be \( p(X) = \{p(1), p(2), \ldots, p(M)\} \) where \( p(x) > 0 \) for all \( x \). Then the cumulative distribution function \( F(X) \) is defined as \( F(x) = \sum_{i=1}^{x} p(i) \). We define a function \( \tilde{F}(X) \) as \( \tilde{F}(x) = \sum_{i=1}^{x-1} p(i) + \frac{1}{2} p(x) \). If we use binary representation for \( \tilde{F}(x) \) after the decimal point, i.e. removing 0., with the length \( l(x) = \lceil \log \frac{1}{p(x)} \rceil + 1 \), we can construct a uniquely decodable code. Shannon-Fano-Elias coding assigns an integral number of bits to each codewords and in this way it differs from arithmetic coding, which will be discussed later.

Ziv-Lempel code

A dictionary based compression is given by Ziv and Lempel [ZL78]. By parsing a message, Ziv-Lempel coding partitions it into variable-length blocks and constructs a dictionary for it. Let a message, \( X = (x_1, x_2, \ldots, x_n) \), be constructed from an \( M \) symbol alphabet. Then the first entry in the dictionary is \( B_1 = (x_1) \). Then the shortest prefix \( B_2 = (x_2, \ldots, x_i) \) of the sequence \( (x_2, \ldots, x_n) \) is added to the dictionary and the procedure is repeated. Each entry in the dictionary is referred to by a pair of integers \((j, x_k)\) in such a way that \( x_k \) is the last symbol in \( B_i \) and \( B_j \) is the sequence obtained by removing \( x_k \) from \( B_i \). The codewords are given from the pair of integers by \( M_j + x_k \). Ziv-Lempel code is a universal source coding, which compresses data without the prior knowledge of the source distribution.

2.2.4 Data compression models

Statistical compression systems such as Huffman and arithmetic coding can be divided into two parts: a model part that predicts incoming symbols, and a coder part which uses the information given by the model to encode the incoming symbols into an output sequence. If a model is fixed throughout the coding of the message, it is a static model and the system is a non-adaptive data compression system. In an adaptive data compression system, the model is updated by the incoming data to reflect the local statistics of data.

In adaptive statistical compression algorithms such as adaptive Huffman or adaptive
arithmetic coding, the probabilities of source symbols and the codewords assigned to the source symbols are dynamically updated according to the incoming source messages. The encoder can update the distribution of symbols by observing input symbols and the decoder can follow the change of the encoder by observing the decoded symbols.

2.2.5 PPM models

PPM stands for Prediction by Partial Matching. PPM takes the context into account. The probabilities of symbols are given as the conditional probabilities. That is, when a symbol \( x_i \) is seen in a source after a sequence \( x_{i-O}, x_{i-O+1}, \ldots, x_{i-1} \), then the probability of \( x_i \) following the context \( x_{i-O}, x_{i-O+1}, \ldots, x_{i-1} \) is given by \( p(x_i|x_{i-1}, \ldots, x_{i-O+1}, x_{i-O}) \). The number of symbols in the context, \( O \), is the order of the model. When the symbol with the \( O-1 \) preceding symbols is not found in the model, the order is reduced and the shorter string in the model, ie. \( x_{i-O+1}, \ldots, x_{i-1} \), is looked up. When the order drops, a special escape symbol is encoded so that the decoder notices the change of the order. When there is no match in the model, equally distributed symbol probabilities are used. The model keeps the frequencies of symbols in a tree structure (generally called trie). Generally PPM models achieve good compression ratio. The disadvantage is that as the order of the model grows, the number of the nodes may increase by the exponent of the number of symbols.

2.3 Security

2.3.1 Symmetric encryption system

A symmetric key cryptosystem allows secure communication over an insecure channel between two parties who share a key. A symmetric encryption algorithm is a collection \( \varepsilon = \{ E_K : K = 1, 2 \cdots N \} \) of invertible transformations indexed by a piece of information called key. To encrypt a plaintext message \( X \), the transmitter who shares a key \( k \)
with the receiver, finds the ciphertext $Y = E_k(X)$ and sends it to the receiver who can use the inverse transformation to recover $X$. The attacker does not know the key. An attacker can always use an exhaustive key search strategy to determine the key and so the number of keys gives an upper bound on the security of the system.

![Diagram of symmetric encryption system](image)

**Figure 2.4: Model of symmetric encryption system**

### 2.3.2 Attacks

An enemy attempts to find the key, or the plaintext of a ciphertext. An encryption system can be attacked by an enemy. There are several attack models as shown below.

**Ciphertext-only attack** An attacker knows only the ciphertext. That is, he/she has access to $E_k(X)$. This is possible if he/she can eavesdrop the channel.

**Known plaintext attack** An attacker knows a set of pairs of plaintexts and their corresponding ciphertexts and tries to discover a key or a plaintext which is not in the known set. This is possible if he/she can eavesdrop the channel and has partial access to the plaintexts.

**Chosen plaintext attack** An attacker can choose a plaintext and obtain the corresponding ciphertext. This is possible if he/she has access to the encryption system and can conduct some experiments.

**Chosen ciphertext attack** An attacker can choose a ciphertext and observe the corresponding plaintext. This is possible if he/she has access to the decryption system and can conduct some experiments.

### 2.3.3 Redundancy of a language

A natural language can be seen as a message source and so can be analyzed using information theory [BP82]. Let $S$ be the alphabet of a language with $M$ symbols, and
2.3. Security

\( S^k \) denote a string of \( k \) characters. Then the rate of language \( r_k \) for messages of length \( k \) is the average amount of information in each character of messages of length \( k \) and is given by

\[
r_k = \frac{H_2(S^k)}{k}. \tag{2.8}
\]

The absolute rate of a language is the maximum amount of information that could be encoded in each character using the alphabet \( S \) assuming that all combinations of symbols are equally likely. It is given by

\[
R = \log_2 M. \tag{2.9}
\]

The redundancy of a language \( D \) with rate \( r \) is given by

\[
D = R - r. \tag{2.10}
\]

For English, \( r_k \) has been estimated as 1.0 to 1.5 bits/letter and \( R \) is 4.7 bits/letter. High value of \( R \) means that English language is highly redundant. For a language, less redundancy means more statistical independence of the successive characters in a message.

2.3.4 Unicity distance

When a language is redundant, the knowledge of statistical properties of the language can be used to attack an encryption system [BP82]. The unicity distance \( U_\Delta \) is the amount of ciphertext required by the attacker to uniquely identify the plaintext, and is given by

\[
U_\Delta = \frac{H(K)}{R}. \tag{2.11}
\]

When the unicity distance is small, the short ciphertext gives enough information to uniquely identify the key \( K \) and hence the security is weak. By lowering \( R \), that is compressing the source and reducing redundancy, unicity distance is increased and a more secure system is obtained. Unicity distance is only related to a particular type of attack and so large unicity distance does not necessarily mean strong security against all attacks.

2.3.5 Data compression and security

Compressing a message source before encryption increases the security by removing the redundancy from the source messages and increasing the unicity distance. An encryption algorithm produces ciphertexts that look like random sequence and so have less...
redundancy. This means that ciphertexts cannot be much compressed. As encryptions are now widely in use on many computer systems, compression before encryption is an important strategy for efficient use of resources. Combining compression and encryption algorithm has the advantage of added efficiency and automatic compression before encryption.

2.4 Arithmetic coding

Arithmetic coding is an optimal data compression algorithm. It was discovered independently by Pasco and Rissanen [BWC90].

Arithmetic coding encodes a message into a bit string which represents a real number interval within the interval [0,1). It starts with an initial interval, usually [0,1), and then narrows it down as new symbols arrive such that the amount of narrowing is determined by the probability of the incoming symbol.

2.4.1 Encoding

The encoder narrows the interval, \([l, h)\), down based on a pair of cumulative frequencies given by a received symbol. Then it outputs the value that represents the interval as a sequence of bits.

Assume that there are \(N_{symbol}\) symbols and a symbol \(\psi[i]\) is given a probability \(p(\psi[i]), 1 \leq i \leq N_{symbol}\), and \(\sum_{j=1}^{N_{symbol}} p(\psi[j]) = 1\). The probability of \(\psi[i]\) is defined as \(\psi[0] = 0\) and \(\psi[i] \neq 0\) for \(1 \leq i \leq N\). Then the pair of cumulative probabilities, \(c_l(\psi[i])\) and \(c_h(\psi[i])\), for \(\psi[i]\) is given by

\[
\begin{align*}
c_l(\psi[i]) &= \sum_{j=0}^{i-1} p(\psi[j]) \quad (2.12) \\
c_h(\psi[i]) &= \sum_{j=0}^{i} p(\psi[j]) \quad (2.13)
\end{align*}
\]

When a symbol \(\psi[i]\) arrives, the lower bound of the interval \(l\) is raised and the upper bound \(h\) is lowered. That is,

\[
\begin{align*}
l_{\text{new}} &= l + (h - l)c_l(\psi[i]) \quad (2.14) \\
h_{\text{new}} &= l + (h - l)c_h(\psi[i]) \quad (2.15)
\end{align*}
\]

where \(c_l(\psi[i])\) and \(c_h(\psi[i])\) are given by (2.13) and (2.13).

The example Fig.2.5 shows how the interval is narrowed down. Assume that there are 10 symbols, that is, \(N_{symbol} = 10\). For \(1 \leq i \leq N_{symbol}\) let \(\psi[i] \in \{A, B, C, D, E, F, G, H, I, J\}\), and the initial interval is [0,1). Then a message ABC is encoded as follows.
2.4. Arithmetic coding

1. The encoder encodes A. The pair of cumulative probabilities are \( c_h(A) = 0.05 \) and \( c_l(A) = 0.0 \). The interval \([0,1)\) is narrowed down and the new interval is \( h_{new} = 0.0 + (1.0 - 0.0) \cdot 0.05 = 0.05 \) and \( l_{new} = 0.0 + (1.0 - 0.0) \cdot 0.0 = 0.0 \).

2. The encoder encodes B. The pair of cumulative probabilities are \( c_h(B) = 0.1 \) and \( c_l(B) = 0.05 \). The interval \([0,0.05)\) is narrowed down and the new interval is \( h_{new} = 0.0 + (0.05 - 0.0) \cdot 0.1 = 0.005 \) and \( l_{new} = 0.0 + (0.05 - 0.0) \cdot 0.05 = 0.0025 \).

3. The encoder encodes C. The pair of cumulative probabilities are \( c_h(C) = 0.175 \) and \( c_l(C) = 0.1 \). The interval \([0.0025,0.005)\) is narrowed down and the new interval is \( h_{new} = 0.0025 + (0.005 - 0.0025) \cdot 0.175 = 0.0029375 \) and \( l_{new} = 0.0025 + (0.005 - 0.0025) \cdot 0.1 = 0.00275 \).

<table>
<thead>
<tr>
<th>Step</th>
<th>( \psi )</th>
<th>( c_h(\psi) )</th>
<th>( c_l(\psi) )</th>
<th>( h )</th>
<th>( l )</th>
<th>( h_{new} )</th>
<th>( l_{new} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>0.05</td>
<td>0.0</td>
<td>1.0</td>
<td>0.0</td>
<td>0.05</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>0.1</td>
<td>0.05</td>
<td>0.05</td>
<td>0.0</td>
<td>0.005</td>
<td>0.0025</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>0.175</td>
<td>0.1</td>
<td>0.005</td>
<td>0.0025</td>
<td>0.0029375</td>
<td>0.00275</td>
</tr>
</tbody>
</table>

Table 2.1: Example calculation of the interval for a message ABC

2.4.2 Decoding

The decoder transforms encoder's output back into the original symbols. The encoded interval is compared with the cumulative probabilities of symbols and the symbol that includes the interval is chosen.

In the example Fig.2.5, the encoded interval is \([0.00275,0.0029375)\). Then the cumulative probabilities of each symbol is compared with the interval. The symbol A includes the interval so the decoder decodes it. The decoder will choose B as the second symbol after seeing A and so on.
Notice that the output of the encoder is not a unique interval but any value in the interval is sufficient to correctly decode a message.

### 2.4.3 Advantage of arithmetic coding

When a message consists of a sequence of \( m \) symbols, \( \psi_1, \psi_2, \ldots, \psi_m \), then the required length to encode the message is shown as

\[
\sum_{i=1}^{m} (-\log_2 p(\psi_i)) \tag{2.16}
\]

where \( p(\psi_i) \) is the probability of a symbol \( \psi_i \).

The compressed data consists of an integral number of bits. In arithmetic coding, a whole message must consist of integral number of bits but each symbol does not have such restriction. If the result of encoding a symbol includes a fraction of a bit, it is passed to the next symbol. This is the advantage of arithmetic coding over Huffman coding. In case of Huffman coding, each symbol should be translated into an integral number of bits and so the fraction of a bit is rounded up to a bit if there is any. The extra fraction for each output symbol cumulates as encoding proceeds and hence adds to the length of the encoded message.

### 2.4.4 Models in arithmetic coding

A model in arithmetic coding is to provide statistics of symbols for the coder. A model has great significance in the compression system and largely affects the compression ratio. It is obvious from (2.16) that the higher is the probability of an incoming symbol, the less is the output length. An arithmetic coding with a model that produces accurate prediction can achieve a good compression.

There are two types of models. A non-adaptive model provides fixed statistics for symbols throughout a whole message. An adaptive model provides symbol statistics that dynamically change as the frequencies of symbols in a message change. The implementation of non-adaptive models will be easier than adaptive ones. However, for a message in which the probability distribution of symbols changes, adaptive models can achieve better compression.

There are various approaches to implement a model. The most common approach is to use Markov modeling. One of the well known algorithms is Prediction by Partial Matching (PPM) [CW84]. The model predicts the next symbol based on \( O \) last seen symbols. The model is often called \textit{order-0 model}, based on the number of symbols \( O \).
used for the prediction. There are some variations such as PPMA and PPMB [Irv95]. The model is implemented as a trie which has a tree structure.

The system can achieve a reasonably good compression with order-3 or order-4 on English text. However, according to the data shown in [CW84], the memory requirement is large. In the worst case the number of nodes in the tree can grow exponentially. If the order is lower, compression programs such as gzip [Gai93] (which uses Ziv-Lempel scheme) or more recently bzip2 [Sew98] (which uses block-sorting scheme [BW94]) can achieve better compression on text and non-text sources [FB98].

2.5 Implementation of arithmetic coding

A software implementation of arithmetic coding was developed by Witten/Neal/Cleary [WNC87] (WNC implementation). The implementation is order-0 adaptive arithmetic encoder/decoder, which means the model consists of the frequencies of symbols.

2.5.1 The model

The adaptive model is implemented as an ordered frequency table, where the symbol with the highest frequency appears at the top and other symbols are listed in decreasing order. Each symbol has a frequency and a cumulative frequency, which is the sum of the frequencies of all the symbols below it. The adaptiveness is achieved by updating the frequency of a symbol when it is seen in the message sequence. In this case, its frequency together with the cumulative frequencies of that symbol and all the symbols above it are similarly incremented by 1. If the increment results in the violation of the frequency order, the symbol is moved up in the table so that the frequency order is maintained.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Freq.</th>
<th>Cum Freq.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>257</td>
</tr>
<tr>
<td>0x00</td>
<td>1</td>
<td>256</td>
</tr>
<tr>
<td>0x01</td>
<td>1</td>
<td>255</td>
</tr>
<tr>
<td>0x02</td>
<td>1</td>
<td>254</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0xFE</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0xFF</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>EOF</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 2.6: Implementation of the frequency table
2.5.2 The coder

The coding procedure is realized using finite precision integer arithmetic. The interval is represented by two 16 bit numbers representing high and low values. The values are updated based on model probabilities and consist of the matched part of the high and low values. With the $i^{th}$ incoming symbol, $\psi$, the new high and low values are calculated as

\[
\begin{align*}
    h'_i &= l_i + \left\lfloor \frac{(h_i - l_i)C_h(\psi)}{C_h(\psi_{\text{max}})} \right\rfloor - 1 \\
    l'_i &= l_i + \left\lfloor \frac{(h_i - l_i)C_h(\psi')}{C_h(\psi_{\text{max}})} \right\rfloor
\end{align*}
\]

(2.17)

where $\psi'$ is the next symbol below $\psi$ in the table and hence $C_l(\psi) = C_h(\psi')$ and $C_h(\psi_{\text{max}})$ is the sum of all frequencies.

If $h'_i$ and $l'_i$ have $u$ common most significant bits, the following step will be executed $u$ times: the most significant bit of $h'_i$ and $l'_i$ that are the same, is sent to the output and $h'_i$ and $l'_i$ are left-shifted.

Define two functions $S_0(x, u)$ which left-shifts $x$ by $u$ bits and fills the least significant $u$ bits by 0 and $S_1(x, u)$ which left-shifts $x$ by $u$ bits and fills the least significant $u$ bits by 1. Assuming that $h'_i$ and $l'_i$ produce output $o_i$ and the size of $o_i$ is $u$ bits, the resulting high and low values are as follows

\[
\begin{align*}
    h''_i &= S_1(h'_i, u) \\
    l''_i &= S_0(l'_i, u)
\end{align*}
\]

(2.18)

However, there is a special case. If the interval satisfies $0x4000 \leq \text{interval} < 0xC000$, mid-range buffering is used. In this case, $0x4000$ is subtracted from both values, they are left-shifted, and one bit, which is not yet fixed, is stored in bits_to_follow, which is a mid-range buffer. This bit is determined when the next output bit is determined. The detail is described in the following section.
The output is produced by repeating the procedure above. The algorithm that we describe is based on WNC implementation assuming that the size of the high and low values is 16 bits.

How to output bits

A new range from the old high and low values and the cumulative frequencies of a symbol, are calculated by using the following states.

1. State 1

   The MSB of the high and low values are the same (① in Figure 2.8). In this case, the MSB is output and the high and low values are left-shifted by 1 bit and

   - 0 is set to the LSB of the low value.
   - 1 to the LSB of the high value.

2. State 2

   The MSB of the high and low values are different and the low value starts with the bit pattern 01 and the high value starts with the bit pattern 10 (② in Figure 2.8). The following section describes the detail of the procedure in this case.

3. State 3

   The MSB of the high and low values are different and the bit patterns of the high and low values are not in State 2 (③ in Figure 2.8). In this case, encoding of the symbol finishes.

![Figure 2.8: Possible states of a range](image)

The procedure of outputting bits proceeds as follows.

- If the range is either in State 1 or 2, execute the appropriate procedure for the state, output bit sequence and update the high and low values. Then, analyze the new high and low values according to the states again.
- If the range is in State 3, finish encoding.

![State transition diagram](image)

**Figure 2.9: State transition of the procedures for State 1, 2 and 3**

**Special case of the range**

If the high value starts with the bit pattern 10 and the low value starts with the bit pattern 01, then 01000000 00000000 is subtracted from both the high and low values (Transition from ① to ② in Figure 2.10 and Figure 2.11).

Then the two values are left-shifted by 1 bit that is equivalent to multiplying the values by two, and

- the LSB of the low value is set to 0.
- the LSB of the high value is set to 1.

( from ② to ③ in Figure 2.10 and Figure 2.11 ).

Notice that the MSB of the resulting high value is 1 and that of the low value is 0. Originally high value starts with 10 and the low value starts with 01.

When 01000000 00000000 is subtracted from both values the result is that the high value starts with 01 and the low value starts with 00. After the left-shift, the high value is 1????????????????1 and the low value is 0????????????????0. Hence, the result is either in State 2 or 3.

Then, a *unknown bit* ( still unknown if it is 0 or 1 ) is put into a special buffer and the state of the high and low values is analyzed in the same way again. The unknown bits should be fixed when an output bit is produced by the pattern 1.

If the bit produced is 1, the first output is 1 and the rest of the bits are 0. If the bit is 0, the first bit is 0 and following bits are 1.
2.6 Conclusion

Arithmetic coding is an optimal compression scheme. The performance is largely determined by the model. The model is generally based on Markov modeling. PPM models...
Unknown bits with a known bit at the end | Output
---|---
???. . . ?1 | 100. . . 00
???. . . ?0 | 011. . . 11

Table 2.2: Example of output of unknown bits

Figure 2.13: Example of how to fix the unknown bit (2)

can achieve a reasonably good performance but such models with higher order require larger amount of memory compared with other algorithms such as Ziv-Lempel. The performance of the lower order models is not as good as algorithms such as Ziv-Lempel and block-sorting.
Chapter 3

Arithmetic coding encryption schemes

Arithmetic coding is an optimal compression algorithm. Compression provides efficient use of channel by reducing the redundancy of data. Compressing data before applying encryption improves security (Section 2.3). If arithmetic coding can be also made to provide security, less overall processing overhead can be expected. Arithmetic coding encryption schemes were proposed by Witten/Cleary [WC88]. Later several attacks on the proposed schemes were published. In this chapter, we describe arithmetic coding encryption schemes, briefly look at the effect of added security on the compression performance and give an assessment of security of the resulting systems. In Section 3.2 and 3.3 we review three known attacks on model-based schemes. The first attack is against non-adaptive models and the other two are against adaptive systems. Section 3.4 shows an attack on a coder-based scheme which uses binary alphabet and Section 3.5 summarizes attacks and the problems of the schemes.

3.1 Arithmetic coding encryption schemes

Adaptive arithmetic coding encryption schemes were motivated by the following observations [WC88].

1. Models for data compression are often very large and may act as an enormous key.

2. If an adaptive model is used, the key depends on the entire transmitted text and finding the key would require tracking the changes to the model by decoding the entire transmission since initialization.

3. It is very difficult to regain synchronization if the models for compression and decompression are different.
The proposed arithmetic coding encryption schemes are symmetric encryption algorithms. Depending on which part of the arithmetic encoder/decoder contributes to the key, the schemes are divided into two categories, that is, *model-based schemes* and *coder-based schemes*. Also there is a scheme which combines the two. We describe these schemes in the following sections.

### 3.1.1 Model-based schemes

In Witten *et al*'s proposal, the model is used as the encryption key: that is the details of the model are only known to the transmitter and the receiver. These schemes are called *model-based*.

**Model-based scheme 1**

The secret key of the scheme is the *initial model*.

In the scheme, the initial model is transmitted through a secure channel and is shared by the transmitter and the receiver. The other parameters such as the initial range of the coder is public. The information about the type of the model, such as PPM and the order of the model, are public. The secret is the parameters such as the initial frequencies of symbols and the order of symbols in the frequency table. Witten and Cleary suggested an array of single-character frequencies in the range of 1-10.

**Model-based scheme 2**

The secret key of the scheme is an *initial string*.

The initial model and range are public and the key is a secret string, shared by the transmitter and receiver, that is input to the system before the actual message is started. The initial string is sent through a secure channel proceeding the transmission of a message. The key string modifies the models and the ranges in both the encoder and the decoder to one which is unknown to the attacker.

### 3.1.2 Coder-based schemes

An alternative approach proposed by Irvine, Cleary and Rinsma-Melchert is *coder-based* scheme [ICRM95].

The secret key of the scheme is a *bit string which is used to narrow the range*.

Based on the key bit sequence, either the high value \( h \) is decreased or the low value \( l \) is increased by the amount \((h - l)\varepsilon\) where \( \varepsilon \) is a public parameter and \( 0 < \varepsilon < 1 \).
The known parameter $\epsilon$ can be a part of the key but it must be carefully chosen not to affect the compression performance. A variation of this scheme, proposed in [LFB98], is described in detail in the next section.

### 3.1.3 A combined scheme

Liu/Farrell/Boyd proposed a system that combines the model-based and the coder-based schemes [LFB98]. The secret key of the scheme consists of the following.

1. The initial model.
2. The initial range in the coder. The range should be larger than $\frac{2^{16} - 1}{4}$.
3. A 16 bit substitution that is used to substitute the first 16 bits of encoder's output.
4. Two sets of shrinking factors $(\epsilon_{h_0}, \epsilon_{l_0})$ and $(\epsilon_{h_1}, \epsilon_{l_1})$, each consisting of an upper and a lower shrinking factor. This is used to narrow the range after encoding of a symbol. The shrinking factors $\epsilon_{h_j}$ and $\epsilon_{l_0}$ should satisfy $\epsilon_{h_0} \neq \epsilon_{h_1}$ and $\epsilon_{l_0} \neq \epsilon_{l_1}$ and $0.9000 \leq \epsilon_{h_j} \leq 0.9999$ and $0.0000 \leq \epsilon_{l_j} \leq 0.0999$ where $j \in \{0, 1\}$.
5. A 128 bit random sequence to control the two sets of shrinking factors. This is cyclically used.

\[\text{Model} \quad \text{Coder} \quad \text{Symbol} \quad \text{Freq table} \quad \text{range} \quad \text{16-bit mask} \]

\[\epsilon_{h_0} \quad \epsilon_{l_0} \quad \epsilon_{h_1} \quad \epsilon_{l_1} \]

\[\text{Figure 3.1: LFB scheme} \]

The procedure to encode a symbol is as follows.

**Step 1** Encoding a symbol according to (2.17).

**Step 2** Output a bit sequence. The high and low values are modified as shown in Fig.2.7.
3.1. Arithmetic coding encryption schemes

Step 3 Shrink the range.

For a set of shrinking factors \((\varepsilon_{h_0}, \varepsilon_{l_0})\) and \((\varepsilon_{h_1}, \varepsilon_{l_1})\), the new \(h_{i+1}\) and \(l_{i+1}\) are calculated as follows.

\[
\begin{align*}
    h_{i+1} &= l_i'' + (h_i'' - l_i'')\varepsilon_{h_j} \\
    l_{i+1} &= l_i'' + (h_i'' - l_i'')\varepsilon_{l_j}
\end{align*}
\]  

\(j \in \{0, 1\}\), and is chosen according to the 128 bit random sequence.

\[
\text{Range } (h_i, l_i) \rightarrow (h_i', l_i') \rightarrow (h_i'', l_i'') \rightarrow (h_{i+1}, l_{i+1})
\]

![Figure 3.2: Procedure of LFB scheme](image)

Liu/Farrell/Boyd claimed that the scheme is resistant against Bergen/Hogan attack (3.3.1) and estimated the cost of attacking the system to be \(2^{14} \times 2^{30} \times 2^{16} \times 2^{19} = 2^{79}\).

3.1.4 Initialization and reset of the encoder

In all schemes, initialization of a system includes loading of the key and initialization of other parameters. In some cases, a system may need to be reset to provide better security.

Initialization and reset should be clearly differentiated. An arithmetic coder encryption scheme has two kinds of parameters which must be set before start of a message. These are i) secret key parameters, for example model in a model-based scheme, and ii) non-key public parameters such as coder range in model-based schemes. During initialization both groups of parameters are set to chosen values which form default values of the system. In system reset key parameters are always set back to default values; non-key ones may take their default values or have new values (publicly known). Reset may be automatically invoked by the encoder/decoder. The trigger may be an end of the message symbol/signal, i.e. EOF, or may be when a message size reaches a certain size.

3.1.5 Effect on data compression performance

One of the important criteria for the performance of an arithmetic encoding encryption schemes is the effect of the added encryption on the data compression performance,
the compression ratio which is the average number of bits per input symbol, and the speed which is the average time to process an input symbol. In either of the model-based schemes, the initial model or the initial string will be randomly chosen. Hence the given model for a message does not necessarily represent the statistics of symbols in the message. If an adaptive model is used, as data compression proceeds the initial model is overwritten by the statistics of the incoming message. Witten et al estimated that 1,000 symbols are enough for order-0 adaptive models to adjust the model to the input message. So if the length of a message is considerably longer than 1,000, the impact of randomly chosen model or initial string on the data compression performance will be small.

In the case of the coder-based schemes, the added encryption algorithm has continuous influence on the data compression. The drop of the compression ratio may be minimized by appropriate choice of $\varepsilon$. However, the scheme will have reduced compression speed because the extra-narrowing is equivalent to encoding a second symbol. For example, the combined scheme by Liu et al [LFB98] results in approximately 2% drop in the compression rate and almost doubles the processing time.

Encryption schemes should be carefully designed so that the drop in the compression ratio and speed is minimized. Otherwise, arithmetic coding encryption schemes can be beaten by a conventional method of using compression and encryption as two separate subsystems, and there is no benefit in using arithmetic coding encryption schemes.

### 3.1.6 Security of the schemes

One simple method of evaluating security is the size of the key space. In model-based schemes, the key is the model and hence the size of the key space is the number of all possible states of the model. As an example, in case of model-based schemes with the order-0 model, assume that the number of symbols is $N_{\text{symbol}}$ and the initial frequency $F(\psi)$ for a symbol $\psi$ can be any number between 1 and $F_{\text{max}}$; that is $1 \leq F(\psi) \leq F_{\text{max}}$. Then the number of possible combinations is $F_{\text{max}}^{N_{\text{symbol}}}$. If symbols are ordered, then there are $N_{\text{symbol}}!$ possible orders of symbols. So the number of possible initial models (keys) is $F_{\text{max}}^{N_{\text{symbol}}} \cdot N_{\text{symbol}}!$. In the case of coder-based schemes, assuming that the size of the key sequence is $s$, there are $2^s$ possible key values. If the exhaustive key search is used to attack a system, the figures given above show at most how many attempts are required.

However, the numbers only give an upper bound as the security. If a different attack method is used, the difficulty of breaking a system will change. The security
strength is largely determined by the attack method used. Hence it is not an easy task to evaluate the security without investigating every possible attack method.

### 3.2 Attack on non-adaptive arithmetic coding encryption scheme

BH [BH92] demonstrated an attack on non-adaptive model-based arithmetic coding encryption scheme. They showed that each symbol corresponds to a fixed output pattern and so non-adaptive system is not suitable for encryption. Cleary/Irvine/Rinsma-Melchert demonstrated a chosen plaintext attack on non-adaptive model-based encryption using binary alphabet (input) to obtain the model. They showed that $\omega + 2$ characters are sufficient to uniquely determine a $\omega$ bit probability [CIRM95]. This target system is much simpler than 8-bit alphabet system.

### 3.3 Attacks on adaptive schemes

#### 3.3.1 Bergen/Hogan attack

BH attack on adaptive schemes is a chosen plaintext attack where the attacker can feed plaintexts of her/his choice to the encoder. The attack does not discover the key (initial model) but succeeds to modify the model into a form which is known to the attacker, hence allowing the attacker to decrypt the communication afterwards.

For the attack to work, the following assumptions are used:

- The system is a model-based scheme of first or second type. That is, the key is either initial model or the initial string. The initial state of the encoder does not affect the attack.
- The system uses order-0 adaptive arithmetic coding. The attack is based on WNC implementation.
- The attacker is able to send symbols to the encoder and obtain the output.
- The attacker has a decoder.

This attack uses two properties of the WNC implementation: halving of symbols' frequencies when $F(V_{\text{max}}) = F_{\text{max}}$, and imposed ordering on the frequency table. The attack consists of two steps.
3.3. Attacks on adaptive schemes

**Step 1** The attacker sends a long enough string of a single symbol, \( \psi \), to cause halving. If the string is chosen long enough, with enough number of halvings occurring, \( \psi \) eventually moves to the top of the table and frequencies of all other symbols become 1. If more \( \psi \) are sent so that another halving occurs, the frequency of \( \psi_{\text{max}} \) will become a constant, given by

\[
F(\psi_{\text{max}}) = \left[ \frac{F_{\text{max}} - N_{\text{symbols}} - 1}{2} \right]
\]  

(3.2)

where \( N_{\text{symbols}} \) is the number of symbols. We refer to this state of the model as **synchronized state**. We note that once the model is synchronized the number of \( \psi \) required to produce another halving, denoted by \( n \), becomes a constant

\[
n = F_{\text{max}} - (N_{\text{symbols}} - 1) - \left[ \frac{F_{\text{max}} - (N_{\text{symbols}} - 1)}{2} \right]
\]  

(3.3)

At this stage the attacker does not know the order of symbols.

**Step 2** He/She sends other symbols one by one, each time moving the sent symbol above other symbols, the frequencies of which are 1, and hence effectively re-ordering symbols in the frequency table.

However the re-ordering will fail if halving occurs during this step. To guarantee the successful re-ordering, it is necessary for already re-ordered symbols not to move. This condition is satisfied when the frequencies of already re-ordered symbols are 2. After the synchronization of the model (Step 1), the frequency of each symbol except the one at the top of the table becomes 1. By sending symbols to re-order (Step 2), their frequencies become 2. If halving occurs, all those frequencies will become 1 and hence those symbols in the table may move when a symbol is sent.

Now the model can be verified by decoding a message. We note that for correct decoding the attacker must know the range of the coder at this stage. This is an assumed knowledge in BH attack.
To protect against this attack, BH proposed regular changing of $F_{\text{max}}$. Transmitter and receiver use a shared pseudo random sequence to generate consecutive values of $F_{\text{max}}$. Since halving occurs when $F(\psi_{\text{max}}) = F_{\text{max}}$ the halving points become unpredictable. This randomization in general reduces the compression rate (although in special cases compression rate might be improved). Lim/Boyd/Dawson suggest regular resetting of the model during encoding a message [LBD97] which effectively discards a model modified by an attacker. However, as shown by their experiment, more frequent reset results in reduced compression rate and this reduction is higher for higher order models.

The attack uses the properties of the adaptive model. Hence any adaptive model has a possibility of the attack although how easy to control the model by a message sent will vary with the type of the model. A model of quick adaptation to an input message can also be quick to be taken control. Obviously the adaptation speed affects the compression so the protection against the attack by slowing down the adaptation will also drop the compression. As can be seen from the attack, an order-0 model can be easily controlled by the attack. However a model of the higher order such as PPM can be attacked by the method but whether or not it is practical in terms of the cost of the attack is unknown.

### 3.3.2 Lim/Boyd/Dawson attack

As noted earlier, BH attack does not find the key. Lim et al showed a chosen plaintext attack that discovers the key. For the attack to work, the following assumptions are used.

- The system is a model-based scheme of type 1. The initial frequencies of symbols are unknown. The order of symbols in the symbol table and the initial range are known.
- The system can be an adaptive arithmetic coding of any order.
- The attacker can send symbols to the encoder and obtain the output.
- The attacker can reset the system, that is, after sending a string through the coder, it can return to the starting state.

The attacker first tries to find out the frequency of the symbol at the bottom of the table, $F(\psi_{\text{min}})$, and the total cumulative frequency $C_h(\psi_{\text{max}})$. For this purpose, s/he
3.4. Attack on coder-based scheme

Irvine, Cleary and Rinsma-Melchert [ICRM95] described an attack on the coder-based scheme. The analysis is based on a static arithmetic coding using an alphabet of size two. An encoding result can be represented by a polynomial in symbol probability of symbols, a constant $\epsilon$ and the key bits, which are the unknown. It is showed that the security can be reduced to a subset sum problem. Irvine et al concluded that the security must come from the model rather than the arithmetic coder.

3.5 Conclusion

Arithmetic coding encryption schemes are to provide security using arithmetic data compression systems by hiding the internal information required to decompress encoder's output. If this can be achieved without adding costly algorithms, better efficiency is expected. The information, which is kept secret, i.e. the key of the encryption system, can be the parameters in the model or the coder. The schemes are categorized into model-based, coder-based and their combined schemes. In model-based schemes, the drop in compression performance can be minimized using the advantage of adaptive models. Coder-based schemes have the disadvantage of drop in the processing speed. In both cases the security of the system cannot be simply evaluated based on the size of the key space.

There are several attacks which use different properties of arithmetic coding. BH attack on adaptive system uses the adaptive property of the model. The attack by Lim et al uses the public knowledge about the model. It is known that the frequency of a symbol takes a value within a certain range and how it is updated. In the attack on
coder-based schemes, encoder's output is used to solve a polynomial and determines the key bits.

In the following chapter, we demonstrate new attacks on the schemes.
Previously known attacks showed that the properties of arithmetic coding can be exploited to break the system. In this chapter we demonstrate new attacks on arithmetic coding encryption schemes and show various new ways of exploiting these properties for attacking the system.

In Section 4.1, new attacks which strengthen the BH attack on the model-based schemes are described. There are two weak points in the BH attack: i) if halving occurs during the re-ordering procedure, re-ordering fails and ii) the BH attack cannot find the coder’s secrets, i.e. the high and low values. In Section 4.1 we show how to detect halving and then describe the methods of finding the high and low values. In Section 4.2 we demonstrate an attack against the combined-scheme which is an extended BH attack. Finally we summarize the attacks and the properties used in them.

4.1 Strengthening BH attack

The major weakness of the BH attack is the re-ordering procedure and that after overflooding, the model is only roughly known and it is necessary to verify the state of the model by decoding the output of the encoder. To correctly decode encoder’s output, it is necessary to know the order of symbols in the frequency table and the high and low values in the coder. If halving of frequencies occurs during the re-ordering procedure, re-ordering fails and it must be performed again. If the attacker can determine the halving point, then this problem can be avoided by doing re-ordering right after the halving. Since the halving period is approximately 8000 symbols and re-ordering requires only 255 symbols, re-ordering right after halving guarantees successful re-ordering. After the successful re-ordering, there are several methods either to investigate the high and low values from encoder’s output, or to force the high and low values to take certain values.
In the following, we show two methods of finding halving points by observing encoder’s output. Then we demonstrate several attacks on the system that allow the range of the coder to be estimated.

### 4.1.1 Detection of the halving point

The first method, which can detect the exact halving point, uses repeated experiments on the output and hence is more expensive. It can be avoided by initializing the system before each message. The second one, uses the size of output per symbol to find $F(\psi_{\text{max}})$, resulting in the number of possible values to be reduced from $2^{13}$ to $2^8$. This is because after synchronization of the model, $F(\psi_{\text{max}})$ satisfies $8064 \leq F(\psi_{\text{max}}) \leq 16383$ and hence, there are $8320 \approx 2^{13}$ possible values for $F(\psi_{\text{max}})$. BH attack blindly tries to decode using all possible values one at a time. Using our proposed attack the number of possible values can be reduced to $2^8$.

#### The relationship between halvings and the size of the encoder’s output

The length of the encoder’s output will give some information about halvings. Let a symbol $\psi$ be sent to synchronize the model. From (2.17), let $t$, given by $t = \left\lceil \frac{C_{\text{max}} - N_{\text{symbol}} + 1}{2} \right\rceil + 1$, denote the frequency $F(\psi_{\text{max}})$ right after a halving after the synchronization of the model, and $f_{\text{max}}$, given by $f_{\text{max}} = C_{\text{max}} - N_{\text{symbol}} - t$, denote the frequency $F(\psi_{\text{max}})$ just before a halving. Notice that because $\left\lceil \frac{t + f_{\text{max}} + 1}{2} \right\rceil = t$, $f_{\text{max}}$ is equal to either $t$ or $t - 1$. This means that every other symbol has a frequency of 1. Now assume we keep sending the symbol $\psi$. In this case $F(\psi_{\text{max}})$ will consecutively take the values from $t$ to $f_{\text{max}}$ and the probability of $\psi$ will consecutively take the values

$$
\frac{t}{t + N_{\text{symbol}} - 1} + \frac{t + 1}{t + N_{\text{symbol}} - 1 + 1} + \frac{t + 2}{t + N_{\text{symbol}} - 1 + 2} + \cdots + \frac{t + f_{\text{max}}}{t + N_{\text{symbol}} - 1 + f_{\text{max}}}
$$

This can be written as

$$
P_t(\psi) = \frac{t + i}{t + N_{\text{symbol}} - 1 + i}
$$

(4.1)

where $0 \leq i \leq f_{\text{max}}$. Just before halving, $i = f_{\text{max}}$. When halving occurs at this stage the probability changes from $P_{f_{\text{max}}}(\psi) = \frac{t + f_{\text{max}}}{t + N_{\text{symbol}} - 1 + f_{\text{max}}}$ to $P_0(\psi) = \frac{t}{t + N_{\text{symbol}} - 1}$. The change in the size of the output is $\log_2 P_{f_{\text{max}}}(\psi)$ to $\log_2 P_0(\psi)$. The graph 4.1 shows the change with different values of $t$ when $N_{\text{symbol}} = 256$.

As the graph shows, a halving almost doubles the output rate (bits/symbol) regardless of the value of $t$. The change will result in a noticeable change in the encoder.
output in terms of values and size. We give two methods of using this information to obtain a good estimate of the halving point.

(1) Using repeated experiments on the output

The change can be observed as the change in the narrowing down of the interval. Between two halving points, the change in the probability of the symbol that is used to over-flood the model, compared to the change caused by the halving, is relatively small. The probability is directly reflected in how much the interval is narrowed down. Hence by observing the change in the rate of the narrowing of the interval, it is possible to detect the halving point.

Assume that the model is synchronized and $t$ is as defined in the previous section. Let $l_j$ and $h_j$ denote the low value and high values, respectively, and $r_j$ denote the range given by $r_j = h_j - l_j$, and the probability of symbol $\psi_{\text{max}}$ is $P_j(\psi_{\text{max}})$. After the synchronization, when the symbol $\psi_{\text{max}}$ is encoded, $h'_j = l_j + \frac{(h_j - l_j)F(\psi_{\text{max}})}{F(\psi_{\text{max}})} = h_j$ and hence the high value does not change and takes the value 0xFFFF. At the point right before halving, $j = f_{\text{max}} - 1$ and the probability of $\psi$ is $P_{f_{\text{max}}-1}(\psi_{\text{max}}) = \frac{t + f_{\text{max}} - 1}{t + N_{\text{symbol}} - 1 + f_{\text{max}} - 1}$.

In the following, we calculate the low value and the range just before the halving
point, after the halving point and then at two more consecutive points. Then we look at the behavior of $\frac{\Delta l_{i+1}}{\Delta l_j}$, $j \in \{1, 2, 3\}$, and show how to have a good estimate for the halving point.

Just before the halving point, we have

\[
\begin{align*}
l_1 &= l_0 + r_0 \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1 + f_{\text{max}}} \\
r_1 &= h_1 - l_1 = h_0 - l_0 - r_0 \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1 + f_{\text{max}}} = r_0 \frac{t + f_{\text{max}}}{t + N_{\text{symbol}} - 1 + f_{\text{max}}} \\
\Delta l_1 &= l_1 - l_0 \\
&= r_0 \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1 + f_{\text{max}}} \\
&= \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1 + f_{\text{max}}} (4.2)
\end{align*}
\]

Similarly,

\[
\begin{align*}
l_2 &= l_1 + r_1 \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1} \\
r_2 &= h_2 - l_2 = h_1 - l_1 - r_1 \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1} = r_1 \frac{t}{t + N_{\text{symbol}} - 1} \\
\Delta l_2 &= l_2 - l_1 = r_1 \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1} \\
&= r_0 \frac{(N_{\text{symbol}} - 1)(t + f_{\text{max}})}{(t + N_{\text{symbol}} - 1 + f_{\text{max}})(t + N_{\text{symbol}} - 1)} (4.3) \\
l_3 &= l_2 + r_2 \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1 + 1} \\
r_3 &= h_3 - l_3 = h_2 - l_2 - r_2 \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1 + 1} = r_2 \frac{t + 1}{t + N_{\text{symbol}} - 1 + 1} \\
\Delta l_3 &= l_3 - l_2 = r_2 \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1 + 1} \\
&= r_1 \frac{(N_{\text{symbol}} - 1)t}{(t + N_{\text{symbol}} - 1)(t + N_{\text{symbol}} - 1 + 1)} (4.4) \\
l_4 &= l_3 + r_3 \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1 + 2} \\
r_4 &= h_4 - l_4 = h_3 - l_3 - r_3 \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1 + 2} = r_3 \frac{t + 2}{t + N_{\text{symbol}} - 1 + 2} \\
\Delta l_4 &= l_4 - l_3 = r_3 \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1 + 2} \\
&= r_2 \frac{(N_{\text{symbol}} - 1)(t + 1)}{(t + N_{\text{symbol}} - 1 + 1)(t + N_{\text{symbol}} - 1 + 2)} (4.5)
\end{align*}
\]

From (4.2) and (4.3), the ratio of $\Delta l_1$ and $\Delta l_2$ is

\[
\frac{\Delta l_2}{\Delta l_1} = \frac{\frac{(N_{\text{symbol}} - 1)(t + 2)}{(t + N_{\text{symbol}} - 1 + f_{\text{max}})(t + N_{\text{symbol}} - 1)}}{r_0 \frac{N_{\text{symbol}} - 1}{t + N_{\text{symbol}} - 1 + f_{\text{max}}}}
\]
4.1. Strengthening BH attack

\[
\begin{align*}
\Delta l_3 &= \frac{r_1 (N_{\text{symbol}} - 1) t}{t + N_{\text{symbol}} - 1 + 1} \\
\Delta l_2 &= \frac{r_1}{t + N_{\text{symbol}} - 1} \\
&= \frac{t}{t + N_{\text{symbol}} - 1 + 1} \approx \frac{(t + N_{\text{symbol}} - 1)}{(t + N_{\text{symbol}} - 1)} \\
\Delta l_4 &= \frac{r_2 (N_{\text{symbol}} - 1) (t + N_{\text{symbol}} + 2)}{t + N_{\text{symbol}} - 1 + 1} \\
&= \frac{t + 1}{t + N_{\text{symbol}} - 1 + 2} \approx \frac{(t + N_{\text{symbol}} - 1)}{t + N_{\text{symbol}} - 1}
\end{align*}
\]

Approximation steps are true because \( t \) is large.

Comparing (4.6) with (4.7) and (4.8), we note that the ratio is almost doubled right after the halving point and stayed nearly constant afterwards. (compare (4.7) with (4.8)). This difference can be used to detect the halving point.

The outline of the attack is as follows:

**Step 1** Send a long enough sequence of a single character, \( \psi \), to synchronize the model.

Finish encoding and obtain encoder’s output, \( o_j \) where \( j = 1 \). Increment \( j \).

**Step 2** Repeat the following step at most \( f_{\text{max}} \) times.

Send a message 1 character longer than the previous one, using \( \psi \). Finish encoding and obtain encoder’s output, \( o_j \). Calculate \( \Delta o_j = o_j - o_{j-1} \). Increment \( i \).

Observe the sequence \( o_j \). Between two halving points, \( \Delta o_j \) gradually decreases but it is doubled just after the halving point.

An example for **step 2** calculations, that is \( \Delta o_j = o_j - o_{j-1} \), is as follows.

\[
\begin{array}{c}
0110100 \ldots 11011010100 \quad (o_j) \\
- 0110100 \ldots 11001101000 \quad (o_{j-1}) \\
0000000 \ldots 00001101100 \quad (\Delta o_j)
\end{array}
\]

Change in \( \Delta o \) at the halving point could be as recorded in Table 4.1.

This method relies on accurate output values and so it is necessary to finish encoding for each sequence of symbols. Although this method enables accurate detection of the
4.1. Strengthening BH attack

<table>
<thead>
<tr>
<th>$\Delta a_j$</th>
<th>Halving point</th>
</tr>
</thead>
<tbody>
<tr>
<td>5128</td>
<td></td>
</tr>
<tr>
<td>5048</td>
<td></td>
</tr>
<tr>
<td>4972</td>
<td></td>
</tr>
<tr>
<td>9624</td>
<td></td>
</tr>
<tr>
<td>9328</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Example of $\Delta a_j$ values

halving point, it is expensive and can be easily avoided by using different non-key values or initializing the system for each message.

(2) Non-repeated analysis of the output

Since there is a noticeable change in the probability before and after a halving, it is possible to detect the halving point by observing how many bits are produced when a symbol is encoded. The advantage of this method is that it does not require repeated experiments. In the implementation that are considered, $t = 8064$, $N_{symbol} = 256$ and $f_{max} = 8063$, and so the number of required bits for a symbol changes from $-\log_2\frac{8064}{8064+256} = -\log_20.9692 = 0.045$ to $-\log_2\frac{8064+8063}{8064+256+8063} = -\log_20.9844 = 0.023$.

In practice, the number of bits per symbol will not precisely follow these exact values. The reasons for this are i) mid-range buffering ii) the output is always an integral number of bits while a symbol might need a fractional number of bits. The fractional part is passed to the next symbol and is added to the bits required by the next symbol. Also if the probability of a symbol $P(\psi)$ is large and hence, $-\log_2 P(\psi) << 1$, encoding the symbol may not produce any output. From above discussion, it is not possible to detect the halving point by observing the encoding result of a single symbol. However it is possible to detect it if the average output size per symbol is considered, although some accuracy will be lost.

![Figure 4.2: Example of average output size of 128 symbols](image)

A rough estimation for the halving point can be obtained if the average bit per symbol is calculated for each $m$ input symbols. In the WNC implementation, the output is byte-buffered and hence, there is an output only when the buffer is filled.
Although this makes the analysis less accurate due to the delay in the output, it is still possible to roughly detect the halving points. In Table 4.2 and 4.3, we give an example of the calculation when \( m = 128 \). The block in which the halving occurs is marked with *.

<table>
<thead>
<tr>
<th></th>
<th>0.023438</th>
<th>0.031250</th>
<th>0.031250</th>
<th>0.023438</th>
<th>0.031250</th>
<th>0.031250</th>
<th>0.023438</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.046875</td>
<td>0.031250</td>
<td>0.023438</td>
<td>0.046875</td>
<td>0.031250</td>
<td>0.023438</td>
<td>0.046875</td>
<td></td>
</tr>
<tr>
<td>0.039062</td>
<td>0.031250</td>
<td>0.031250</td>
<td>0.039062</td>
<td>0.031250</td>
<td>0.031250</td>
<td>0.039062</td>
<td></td>
</tr>
<tr>
<td>0.046875</td>
<td>0.031250</td>
<td>0.023438</td>
<td>0.046875</td>
<td>0.031250</td>
<td>0.023438</td>
<td>0.046875</td>
<td></td>
</tr>
<tr>
<td>0.039062</td>
<td>0.031250</td>
<td>0.031250</td>
<td>0.039062</td>
<td>0.031250</td>
<td>0.031250</td>
<td>0.039062</td>
<td></td>
</tr>
<tr>
<td>0.039062</td>
<td>0.031250</td>
<td>0.023438</td>
<td>0.039062</td>
<td>0.031250</td>
<td>0.023438</td>
<td>0.039062</td>
<td></td>
</tr>
<tr>
<td>0.046875</td>
<td>0.031250</td>
<td>0.023438</td>
<td>0.046875</td>
<td>0.031250</td>
<td>0.023438</td>
<td>0.046875</td>
<td></td>
</tr>
<tr>
<td>0.039062</td>
<td>0.031250</td>
<td>0.023438</td>
<td>0.039062</td>
<td>0.031250</td>
<td>0.023438</td>
<td>0.039062</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.2: Average bits/symbol of 128 symbol block: bit-oriented output
It can be seen that it is possible to find one or two blocks which are likely to include the halving point. The result of the same test performed on byte-oriented output implementation, ie. original WNC implementation, is given in Table 4.3. The number of bits per symbol is either 0 or \( \frac{8}{128} = 0.0625 \), and so there is either no output or one byte output per 128 symbols.

In this latter case detection of halving is less accurate but if the rate of producing output bytes is carefully monitored, it will be possible to detect the halving point in a way similar to bit-oriented output since there will be a sudden increase in the output rate.

### 4.1.2 Attacks on the coder

It is necessary for an attacker to know the range of the coder to decrypt the message. We give three attacks on the coder. The first attack uses attacker’s encoder to discover the range and the cost is \( 2^{15} \). This is the cost of finding the high value. The second one only discovers the low value of the range but can reduce the number of possible values to 7. The last attack forces the range of the coder to take one of the two possible values.
The range of the coder changes as encoding proceeds and its value depends on the encoded symbols’ cumulative frequencies. Without the coder range, it is impossible to decode a message even if the model is known. The encoder’s output leaks information about the model and the range, and the amount of information depends on the input string and the state of the model.

Our proposed attacks on the coder use the following properties of the coder.

**Property 1** If the same plaintext is encoded using two encoders which have the same values for \( h_0 - l_0 \) and identical model, but have different values for \( l_0 \), the difference of the two output strings is equal to the difference between the two low values, which will consists of 16 bits (because 16 bit integer arithmetic is used) followed by zeros.

**Property 2** If the cumulative probability of \( \psi_{\min} \) is small, encoding the symbol produces a considerable length output which is very close to the low value used to encode the symbol.

**Property 3** If \( n \) bits are output, the least significant \( n \) bits of the high and low values become known because they are filled with 1s and 0s for the high and low values, respectively. If \( n \) is large, the number of unknown bits becomes small.

**Property 4** If symbols with very small probabilities are continuously encoded, the high and low pair has two possible values and in each case once the high and low pair takes that value, it will stays at that value. (Table 4.6, Table 4.7, Table 4.8)

In the rest of this section, we describe several methods to find the range of the coder when the model is known. Firstly we describe the attack using property 1. Then we give two more attacks based on properties 2, and 3 together with 4. The latter attack is particularly interesting as it produces only two alternatives for the range and hence effectively allows the attacker to discover the range.

1. **Attack on the coder using properties of encoder’s output**

This attack uses **Property 1** above. The assumptions are as follows.

- The attack is a chosen plaintext attack.
- The key is the initial range.
- The model is known.
The attacker has her/his own encoder and is able to initialize it. Notice that the assumption that the model is known above is almost satisfied after the synchronization of BH attack except for the frequency \( F(\psi_{\text{max}}) \). The method is as follows.

**Step 1** Encode a text and obtain the output \( o_1 \). The text should be chosen to produce an output considerably longer than 16 bits.

**Step 2** To find range, set the low value to 0 and find possible high values. The most significant bit of \( h \) and \( l \) should be different and it should not satisfy the condition for mid-range buffering so \( h - l \geq 0x8000 \). The high value starts from 0x8000. Encoding the same text used in step 1 gives the output \( o_2 \). Calculate \( o_1 - o_2 \).

Since the offset of the correct range from the experimental one is 16 bits (\( d \) in Fig.4.3), the most significant 16 bits of \( o_1 - o_2 \), denoted by \( [o_1 - o_2]_M \), represents the low value. The subsequent least significant bits of \( o_1 - o_2 \), denoted by \( [o_1 - o_2]_L \), must be 0 if \( h_0 - l_0 \) is correctly evaluated. However due to various inaccuracies of integer arithmetic \( [o_1 - o_2]_L \) may not be 0 but will be a value very close to 0.

**Step 3** Increment the high value by 1 and repeat the above **Step 2** until \( [o_1 - o_2]_L \) becomes 0 or very close to 0.

![Figure 4.3: Relation between the difference of range and output](image)

If such a range is found, even if it is not the correct one, it can be used to correctly decode the message. Let \( u \) be \( \left\lfloor \frac{(h - l)F(\psi')}{F(\psi_{\text{max}})} \right\rfloor \) then \( uF(\psi_{\text{max}}) \leq (h - l)F(\psi') < (u + 1)F(\psi_{\text{max}}) \). \( F(\psi_{\text{max}}) \) is constant and for a given \( F(\psi') \) and \( u \), there are many possible \( h \) and \( l \) that satisfy the condition. Since the calculation of the new range only depends on the previous range, once the range takes the correct value the sequence of symbols that follow can be correctly decoded. The number of the valid ranges depends on the
frequency of the symbol and the range, and in general, is not one. The cost of the attack on WNC implementation is at most $2^{15}$. This is because we only search for the high value and the low value is fixed to 0. The high value is 16 bits and its MSB will be 1 and so the number of the possible values for the high value is $2^{15}$.

(2) Finding the low value from output

The following attack is based on Property 2. Assume that the mid-range buffering does not occur. When a symbol at the bottom of the frequency table is encoded, the high and low values are updated as (2.17) and $0x8000 \leq \frac{(h_i-l_i)}{F(\psi_{max})} < 0xFFFF$, because $h_i-l_i$ is between 0x8000 and 0xFFFF. If $F(\psi_{max})$ is around 16000, then $1 \leq \frac{(h_i-l_i)}{F(\psi_{max})} < 5$.

Assuming that encoding the symbol $\psi$ produces $u$ bits of output, then $m \leq l'_i \leq M$ where $M$ is a 16 bit value having the $u$ MSB equal to the $u$ MSB of $l'_i$ and the rest equal to 1, while $m$ is the same as $M$ with $16 - u$ LSB equal to 0.

If $u = 14$, then $M-m = 3$ because $M = ***************11$ and $m = **************00$ and $m \leq l'_i \leq m + 3$. Because $l'_i = l_i + \lfloor \frac{(h_i-l_i)}{F(\psi_{max})} \rfloor$ we have $l_i = l'_i - \lfloor \frac{(h_i-l_i)}{F(\psi_{max})} \rfloor$ and so $m - 4 \leq l_i < m + 3$. Hence there are only 7 possible values for $l_i$ which can be derived from the output as shown above.

(3) Discovering the range

This part describes how Property 3 and Property 4 can discover the range. Assume the model to be synchronized, using the symbol 0x00 for over-flooding and the symbols are in ascending order in the table, ie. 0x00, 0x01, ... 0xFF. The procedure is as follows.

1. After over-flooding and re-ordering, frequencies of all symbols except the one used for halving are 2.

Then, cause another halving to make the frequency of each symbol equal to 1. Send a message consisting of the symbol used to synchronize the model to make $C_h(\psi_{max})$ close to the upper limit $C_{max}$.

2. Use the following procedure to generate a long output:

i) Input symbol 0xFF at the bottom of the frequency table. The symbol goes to the second place in the table and the second symbol 0x01 moves to the bottom.

ii) Input symbol 0x01. The symbol goes to the third place in the frequency table and the third symbol 0x02 is shifted to the bottom.
iii) Input symbol 0x02. The symbol goes to the fourth place in the frequency table and the fourth symbol 0x03 moves to the bottom.

iv) Repeat as above for all symbols up to 0xFE. This will give an attacker a long enough output.

3. Try to decode the output using one of the ranges below and verify the decoding result.

   (a) (0x0000, 0xFFFF)
   (b) (0x4000, 0xFFFF)

The reason that Step 3 above works is the following. If the last symbol in the table is encoded, the output size would be $-\log_2 \frac{1}{8000} \approx 12$ bits when the total frequency is close to 8000, and would be $-\log_2 \frac{1}{16000} \approx 14$ bits when the total is around 16000. If 14 bits are output, each high and low value is left-shifted by 14 bits and the lower 14 bits are filled by 1 for the high value and by 0 for the low value. The only unknown is the first 2 bits. There are 4 possible combinations: that is, $(l, h) = \{(00, 10), (00, 11), (01, 10), (01, 11)\}$. However, (01, 10) will cause the mid-range buffering, resulting in (0x0000, 0xFFFF), and so there are only three possibilities for the range of the next symbol. That is, (0x0000, 0xBFFF), (0x0000, 0xFFFF) and (0x4000, 0xFFFF). If the range is (0x0000, 0xBFFF), the new range is calculated from (2.17) as given in Table 4.6. Hence the range (0x0000, 0xBFFF) changes into (0x4000, 0xFFFF). Similarly, the ranges (0x0000, 0xFFFF) and (0x4000, 0xFFFF) can be calculated as given in Table 4.7 and Table 4.8. Since both of these give the same values, there are only 2 possible ranges. Once the range drops into one of them, the same output pattern is repeated. That is, the range (0x0000, 0xFFFF) repeats output pattern 00000000000001 and the range (0x4000, 0xFFFF) repeats output pattern 01100000000000.

A 14 bit output results in 2 possible ranges and a 12 bit output will result in 48 possible ranges and hence the number of possible ranges is greatly reduced.

Assume that the output size is $u$, then the number of the unknown bits of $h_i$ and $l_i$ is $2^{16-u}$. Hence after the output, the $2^{16-u-1}$ MSB of $h_i$ and $l_i$ must be different and so are unknown. Now we have $2^{16-u-1} \times 2^{16-u-1}$ possible $h_i$ and $l_i$ pairs but some may satisfy the condition of the mid-range buffering and if so, $h_i$ and $l_i$ are left-shifted and LSB of $h_i$ and $l_i$ are filled by 1 and 0, respectively. As a result of the mid-range buffering, new $h_i$ and $l_i$ will be equal to those of the other pairs and the above procedure will be repeated until $h_i$ and $l_i$ does not satisfy the condition of the mid-range buffering.
We define group 1 as $h_i$ and $l_i$ which does not satisfy the condition of the mid-range buffering and group $g$, $1 < g \leq 4$, is defined as $(h_i, l_i)$ pairs which satisfy the condition of the mid-range buffering and result in the values found in group $g - 1$ after the mid-range buffering occurs. The high/low values in group 1 is outside of the mid-range (0x4000 to OxBFFF). If either of the high or low value belongs to this group, no mid-range buffering occurs. The high or low values in group 2 may cause the mid-range buffering but at most once, i.e. 1 bit in the mid-range buffer. As a result of the mid-range buffering, the value changes into one of the values in group 1. Similarly, the high/low values in group 3 may cause the mid-range buffering at most twice and after the mid-range buffering, the values change into the values in group 2. For the values in group 4, at most three bits may go to the mid-range buffer and so on.

<table>
<thead>
<tr>
<th>Group</th>
<th>Low values</th>
<th>High values</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x0000, 0x1000, 0x2000, 0x3000</td>
<td>0xCFFF, 0xDFFF, 0xEFFF, 0xFFFF</td>
</tr>
<tr>
<td>2</td>
<td>0x4000, 0x5000</td>
<td>0xAFFF, 0xBFFF</td>
</tr>
<tr>
<td>3</td>
<td>0x6000</td>
<td>0x9FFF</td>
</tr>
<tr>
<td>4</td>
<td>0x7000</td>
<td>0x8FFF</td>
</tr>
</tbody>
</table>

Table 4.4: Groups of ranges with 12 bit output

For a $h_i$ and $l_i$ pair, the one which has the lower number group determines how many times the mid-range buffering occurs. The following tables shows the movement of the higher number groups to the lower number groups.

<table>
<thead>
<tr>
<th>Group 4</th>
<th>Group 3</th>
<th>Group 2</th>
<th>Group 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x7000</td>
<td>0x6000</td>
<td>0x4000</td>
<td>0x0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x1000</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x5000</td>
<td>0x2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0x3000</td>
<td>0xCFFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0xAFFF</td>
<td>0xDFFF</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>0xEFFF</td>
</tr>
<tr>
<td>0x8FFF</td>
<td>0x9FFF</td>
<td>0xBFFF</td>
<td>0xFFFF</td>
</tr>
</tbody>
</table>

Table 4.5: Changing ranges of mid-range buffering with 12 bit output

Cost of attack

In the WNC implementation, halving occurs when the total frequency reaches $C_{\text{max}}$. At the beginning of the attack, frequencies are not known. For a $\psi$, the value $F(\psi)$
4.1. Strengthening BH attack

<table>
<thead>
<tr>
<th></th>
<th>Low value</th>
<th>High value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial value</td>
<td>0000 0000 0000 0000</td>
<td>1011 1111 1111 1111</td>
</tr>
<tr>
<td>After encoding</td>
<td>0000 0000 0000 0011</td>
<td>0000 0000 0000 0101</td>
</tr>
<tr>
<td>After output</td>
<td>0110 0000 0000 0000</td>
<td>1011 1111 1111 1111</td>
</tr>
<tr>
<td>Mid range buff.</td>
<td>0010 0000 0000 0000</td>
<td>0111 1111 1111 1111</td>
</tr>
<tr>
<td>Final value</td>
<td>0100 0000 0000 0000</td>
<td>1111 1111 1111 1111</td>
</tr>
</tbody>
</table>

Table 4.6: The change of the range (0x0000,0xBFFF) with 14 bit output

<table>
<thead>
<tr>
<th></th>
<th>Low value</th>
<th>High value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial value</td>
<td>0000 0000 0000 0000</td>
<td>1111 1111 1111 1111</td>
</tr>
<tr>
<td>After encoding</td>
<td>0000 0000 0000 0100</td>
<td>0000 0000 0000 0111</td>
</tr>
<tr>
<td>After output</td>
<td>0000 0000 0000 0000</td>
<td>1111 1111 1111 1111</td>
</tr>
</tbody>
</table>

Table 4.7: The change of the range (0x0000,0xFFFF) with 14 bit output

satisfies $1 \leq F(\psi) \leq C_{\max} - N_{\text{symbol}} - 1$; i.e. $1 \leq F(\psi) \leq 16127$. To synchronize the model, all $F(\psi)$ except $F(\psi_{\max})$ should be reduced to 1. Assuming that the attacker is going to synchronize the model using symbol $\psi$, and the frequency of another symbol $\psi'$ is $F(\psi') = 16127$ (and so all other frequencies are 1), $\log_2 16127 \approx 14$ halvings are required to reduce $F(\psi')$ to 1. Hence, after at most 14 halvings any frequency can be reduced to 1.

In the above case, after the first halving, $F(\psi')$ becomes $\left\lfloor \frac{F(\psi')+1}{2} \right\rfloor = \left\lfloor \frac{16127+1}{2} \right\rfloor = 8064$ and $F(\psi'_{\max}) = 8064 + N_{\text{symbol}} - 1 + 1 = 8064 + 256 + 1 = 8321$. To cause another halving, $16383 - 8321 + 1 = 8063$ symbols must be sent. So in general, to synchronize the model, at most a message of length $14 \times 8063 = 112882$ of symbols is required. After synchronization, 255 symbols for re-ordering and approximately 8000 symbols for another halving should be sent. To attack the coder, the small probability of symbols are required. If the symbol at the bottom of the table is chosen, the probability is $\frac{1}{C_h(\psi_{\max})}$ and when $C_h(\psi_{\max})$ is close to its maximum value, the probability becomes close to the minimum and the output rate of 14 bits/symbol is achieved. Another 8000 symbols will achieve this. Hence, the total number of required symbols is $112882 + 255 + 8000 + 8000 \approx 130000$. $F(\psi_{\max})$ is unknown and there are $2^8$ possible values for $F(\psi_{\max})$ and the range of the coder will be one of the two values.

The pattern of the bit sequence of encoder's output is known and so the relationship between a symbol and its encoded output is clear.

Notice that the attack on the coder to discover the range can be used on coder-based scheme in general. If the two consecutive ranges are known, the key information that is used to narrow the range can be obtained.
4.2 Attack on a combined scheme

The two schemes, ie. the model-based and the coder-based schemes, can be combined together. An example of the combined scheme is Liu/Farrell/Boyd scheme 3.1.3. The system can be attacked using a method similar to the attack on the model-based system. The model can be modified into a known form by BH attack and the secret in the coder can be discovered by the attacks described in the previous sections.

In this section we describe the attack on the Liu/Farrell/Boyd scheme as an example of the combined scheme. The attack has two steps and can reduce the key space to $2^{20}$. In the first step the attacker uses BH attack to take control over the model by sending a chosen message and finding an approximate value for the halving point, and in the second step he uses a method of discovering the shrinking factors by encoding symbols of small frequencies.

### 4.2.1 Outline of the attack

The relationship between an input symbol, the model and the coder is i) the model is modified according to an input symbol and ii) the coder encodes a symbol, which is represented by two consecutive cumulative frequencies given by the model. Once the model is known, the coder can be controlled using the knowledge of the model. BH attack modifies the model into a known form. The weakness of BH attack is that if halving occurs during the re-ordering procedure, the attack fails. The attack strengthens BH attack using the method which enables detecting of the halving point. The secret parameters of the coder, $h_i$ and $l_i$, can be controlled by encoding symbols of small frequencies. This is possible if the model is known. From (2.18), when output size is $u$ bits, the number of unknown bits after output is $16 - u$ because the least significant $u$ bits are filled with the known values. If $u$ is large, the number of the unknown bits becomes small. The maximum of $u$ is given by

$$-\log_2 \frac{1}{C_{\text{max}}} = -\log_2 \frac{1}{16383} \approx 14$$

Table 4.8: The change of the range (0x4000,0xFFFF) with 14 bit output
and hence, the unknown bits of $h_i$ and $l_i$ are reduced to 2. Using the knowledge of the model it is possible to selectively encode symbols of small frequencies. Furthermore, it is known that $C_h(\psi_{\text{max}})$ is maximized right before halving.

The assumptions are as follows.

1. The scheme is Liu/Farrell/Boyd scheme.

2. The implementation is based on Neal/Witten/Cleary implementation. The output is byte-buffered, ie. 8 bit blocks are output. We do not consider any I/O buffering of stdio, which may be used by the operating system since the scheme does not explicitly take into account the blocking and the buffering as part of the system. The effect of the blocking on the attack is discussed in Section 4.2.4.

3. The attack is a chosen plaintext attack. The attacker has access to the encoder and can see the output. The attack does not require any reset of the system.

The steps of the attack are

**Step 1**

1. Over-flood the model by sending a long enough message consisting of single symbol $\psi$ and observe the change of the output length per symbol. Obtain a rough estimate of the halving point. Details of how to detect the halving point is described in Section 4.1.1.

2. Re-order the symbols in the frequency table. At this stage, the model becomes known to the attacker. That is, the frequencies of all the symbols except $\psi$ at the top of the table becomes 2.

**Step 2**

1. Send a message consisting of the symbol $\psi$ again to cause a halving and to make $C_h(\psi)$ close to $C_{\text{max}}$, ie. 16383. After this, the frequencies of all the symbols except $\psi$ at the top of the table become 1 again.

2. Repeatedly send the symbols $\psi_{\text{min}}$ at the bottom of the frequency table and obtain the output. There are 255 symbols with frequency 1. Encoding these symbols produces a considerably long output. Details are given in Section 4.2.3.

3. Identify the output bit sequence for each of the 255 symbols by analyzing the bit patterns. Details are given in Section 4.2.4.

4. Calculate the lower shrinking factors from the bit sequence. Details are given in Section 4.2.3.

5. Find the 128 bit random sequence using the lower shrinking factors found.
4.2. Attack on a combined scheme

In the following, we describe how to detect the halving point and analyze the security that the shrinking factors provide, followed by a method to find the shrinking factors and the random bit sequence.

4.2.2 Security of the shrinking factors

This section analyzes the security that is provided by the shrinking mechanism in the coder and describes a method to find the shrinking factors.

Assumptions are as follows.

1. The model is known. This means observation is done after synchronization of the model.
2. \( C_h(\psi_{\text{max}}) \) is close to \( C_{\text{max}} \), i.e., 16383.
3. The frequencies of the symbols other than the one at the top of the table are 1.
4. The first bit of output bit sequence as a result of encoding a symbol is known.
5. There is no mid-range buffering.

When a symbol with a probability close to \( \frac{1}{C_{\text{max}}} \), is encoded, if there is no shrinking, the output length is approximately 14 bits. This follows from (4.9). The extra-narrowing shrinks the range by at most approximately 80% and so the output length will be \( -\log_2 0.8 \frac{1}{C_{\text{max}}} \approx 14.3 \) bits. This affects the required number of bits for a symbol. For example, right before halving and without shrinking, the required number of bits for \( \psi_{\text{max}} \) is \( -\log_2 0.9844 \). If it is shrunk by 80%, the required length is \( -\log_2 (0.9844 \times 0.8) = 0.3446 \) bits. In fact, extra-narrowing gives two possible ranges for each symbol, one of which will be selected depending on the pseudo-random bit. Although for a single character, the extra-narrowing can disturb detection of the halving point but if the average length over blocks of 128 characters is calculated, the variation in the range can be averaged and the halving point can be detected.

From (2.18), if the output size is 14 bits, the high and low values are \( h_i'' = **11111111111111 \) and \( l_i'' = **00000000000000 \) respectively. From the conditions for the output in Section 2.5, \( i \) \( h_i'' \) and \( l_i'' \) do not have any common most significant bit and \( ii \) the interval defined by \( h_i'' \) and \( l_i'' \) does not satisfy \( 0x4000 \leq \text{interval} < 0xC000 \). There are three pairs of \( h_i'' \) and \( l_i'' \) which satisfy these conditions; possible pairs are, \( (0xFFF0, 0x4000), (0xFFF0, 0x0000) \) and \( (0xBF0F, 0x0000) \).
4.2. Attack on a combined scheme

$h_i''$ and $l_i''$ are narrowed down by either of the two sets of shrinking factors according to the 128 bit random sequence. Hence, there are six possible ranges in Step 1 of Section 3.1.3. Each of the six ranges becomes one of the three possible ranges after the output procedure in Step 2 in Section 3.1.3.

![Diagram showing the encoding procedure]

Figure 4.4: One cycle of encoding procedure

Weakness in the coder

Assume $o_i$ is a 16 bit value, where its 14 MSB are the output and the rest is 0, and there is no mid-range buffering. Then, $o_i$ is very close to $l_i'$ because $\frac{h_i - l_i}{C_h(\psi_{\text{max}})}$, given in (2.17), is very small. So the attacker can derive $l_i'$ from the output $o_i$ if a symbol of small frequency is encoded and mid-range buffering does not occur. Also the high and low values in Step 2 of Section 3.1.3 are known by the attacker.

4.2.3 A method to derive the shrinking factors from the output

If after the synchronization of the model a certain size sequence consisting of the symbol at the top of the frequency table is sent to the encoder, another halving will occur and the model will take a form in which frequencies of all symbols except the one at the top of the frequency table are 1 and $C_h(\psi_{\text{max}})$ is close to $C_{\text{max}}$. At that moment, the cumulative frequencies $C_h(\psi_{\text{min}})$ and $C_l(\psi_{\text{min}})$ for the symbol at the bottom of the frequency table, $\psi_{\text{min}}$, are 2 and 1, respectively. This is because there is an EOF symbol below $\psi_{\text{min}}$. From (2.17) in Step 2 of Section 4.2.1, the high and low values $h_i'$ and $l_i'$, after encoding the symbol $\psi_{\text{min}}$ are calculated as,

$$h_i' = l_i + \frac{2(h_i - l_i)}{C_h(\psi_{\text{max}})} \quad \text{and} \quad l_i' = l_i + \frac{h_i - l_i}{C_h(\psi_{\text{max}})}$$

(4.10)
where $C_h(\psi_{\text{max}})$ is close to $C_{\text{max}}$. 

From (3.1) and (4.10),

\[
h_{i+1}' = l_{i+1} + \frac{2(h_{i+1}'' - l_{i+1}'')(\varepsilon_{h_j} - \varepsilon_{l_j})}{C_h(\psi_{\text{max}}) + 1} + \frac{2(h_{i}'' - l_{i}'')(\varepsilon_{h_j} - \varepsilon_{l_j})}{C_h(\psi_{\text{max}}) + 1}
\]

\[
l_{i+1}' = l_{i+1} + \frac{h_{i+1}'' - l_{i+1}''}{C_h(\psi_{\text{max}}) + 1} = (h_{i}'' - l_{i}'')(\varepsilon_{h_j} - \varepsilon_{l_j})
\]

(4.11)

When $C_h(\psi_{\text{max}})$ is close to 16383, it is known by the attacker that the length of encoder's output is 14 bits and that the high and low values $h_i''$ and $l_i''$ take one of the values in Section 4.2.2.

**Pattern 1** $h_i = \text{0xBFFF}$, $l_i = \text{0x0000}$,

\[
h_{i+1}' = \text{0xBFFF} \varepsilon_{l_j} + \frac{0x17FFE(\varepsilon_{h_j} - \varepsilon_{l_j})}{16383}
\]

\[
l_{i+1}' = \text{0xBFFF} \varepsilon_{l_j} + \frac{0x17FFE(\varepsilon_{h_j} - \varepsilon_{l_j})}{16383}
\]

(4.12)

**Pattern 2** $h_i = \text{0xFFFF}$, $l_i = \text{0x4000}$,

\[
h_{i+1}' = \text{0x4000} + \text{0xBFFF} \varepsilon_{l_j} + \frac{0x17FFE(\varepsilon_{h_j} - \varepsilon_{l_j})}{16383}
\]

\[
l_{i+1}' = \text{0x4000} + \text{0xBFFF} \varepsilon_{l_j} + \frac{0x17FFE(\varepsilon_{h_j} - \varepsilon_{l_j})}{16383}
\]

(4.13)

**Pattern 3** $h_i = \text{0xFFFF}$, $l_i = \text{0x0000}$,

\[
h_{i+1}' = \text{0xFFFF} \varepsilon_{l_j} + \frac{0x1FFE(\varepsilon_{h_j} - \varepsilon_{l_j})}{16383}
\]

\[
l_{i+1}' = \text{0xFFFF} \varepsilon_{l_j} + \frac{0x1FFE(\varepsilon_{h_j} - \varepsilon_{l_j})}{16383}
\]

(4.14)

From (3.1),

\[
0.8001 \leq \varepsilon_{h_j} - \varepsilon_{l_j} \leq 0.9999
\]

\[
4 \leq \frac{0x17FFE(\varepsilon_{h_j} - \varepsilon_{l_j})}{16383} \leq 5
\]

\[
2 \leq \frac{0x1FFE(\varepsilon_{h_j} - \varepsilon_{l_j})}{16383} \leq 7
\]

\[
3 \leq \frac{0xFFFF(\varepsilon_{h_j} - \varepsilon_{l_j})}{16383} \leq 7
\]

(4.15)

From (4.12), (4.13), (4.14) and (4.15), it is shown that the effect of $\varepsilon_{h_j} - \varepsilon_{l_j}$ is small.

Define $\alpha$ as $\frac{\beta(\varepsilon_{h_j} - \varepsilon_{l_j})}{16383}$ where $\beta = \{0x1FFFF, 0x17FFE, 0xFFFF, 0xBFFF\}$ and assume that there is no mid-range buffering, and $\alpha_{i+1}$ is a 16 bit value, where its most significant 14 bits is the output and the rest is 0. Because the output $\alpha_{i+1}$ is very close to $l_{i+1}'$, $\varepsilon_{l_j}$ can be shown as follows.
4.2. Attack on a combined scheme

Pattern 1 $h_i = \text{oxBFFF}, l_i = \text{0x0000},$

$$o_{i+1} \approx \text{oxBFFF} \varepsilon_{ij} + \alpha \text{ and } \varepsilon_{ij} \approx \frac{o_{i+1} - \alpha}{\text{oxBFFF}} \tag{4.16}$$

Pattern 2 $h_i = \text{oxFFFF}, l_i = \text{0x4000},$

$$o_{i+1} \approx \text{oxFFFF} + \text{oxFFFF} \varepsilon_{ij} + \alpha \text{ and } \varepsilon_{ij} \approx \frac{o_{i+1} - \text{oxFFFF} - \alpha}{\text{oxFFFF}} \tag{4.17}$$

Pattern 3 $h_i = \text{oxFFFF}, l_i = \text{0x0000},$

$$o_{i+1} \approx \text{oxFFFF} \varepsilon_{ij} + \alpha \text{ and } \varepsilon_{ij} \approx \frac{o_{i+1} - \alpha}{\text{oxFFFF}} \tag{4.18}$$

From (4.15), $\frac{\alpha}{\text{oxFFFF}} \leq 0.0001$ and $\frac{\alpha}{\text{oxBFFF}} \leq 0.0001$. So the lower shrinking factor $\varepsilon_{ij}$ can be accurately calculated from the output.

There are three possible patterns of the high and low values and there may be 0.0001 error in the calculation of the lower shrinking factor from above, so the cost of obtaining the lower shrinking factor is $3 \times 2$. Notice that $o_{i+1}$ in Pattern 2 is equal to $o_{i+1} + \text{oxFFFF}$ in Pattern 1.

Now the lower shrinking factor $\varepsilon_{ij}$ is given and from (4.12), (4.13) and (4.14), the upper shrinking factor can be calculated as follows.

Pattern 1 $h_i = \text{oxBFFF}, l_i = \text{0x0000},$

$$\varepsilon_{hj} = \varepsilon_{lj} + \frac{16383}{\text{oxBFFF}}(l'_{i+1} - \text{oxBFFF} \varepsilon_{lj})$$
$$\approx \varepsilon_{lj} + \frac{16383}{\text{oxBFFF}}(o_{i+1} - \text{oxBFFF} \varepsilon_{lj}) \tag{4.19}$$

Pattern 2 $h_i = \text{oxFFFF}, l_i = \text{0x4000},$

$$\varepsilon_{hj} = \varepsilon_{lj} + \frac{16383}{\text{oxFFFF}}(l'_{i+1} - \text{oxFFFF} \varepsilon_{lj})$$
$$\approx \varepsilon_{lj} + \frac{16383}{\text{oxFFFF}}(o_{i+1} - \text{oxFFFF} \varepsilon_{lj}) \tag{4.20}$$

Pattern 3 $h_i = \text{oxFFFF}, l_i = \text{0x0000},$

$$\varepsilon_{hj} = \varepsilon_{lj} + \frac{16383}{\text{oxFFFF}}(l'_{i+1} - \text{oxFFFF} \varepsilon_{lj})$$
$$\approx \varepsilon_{lj} + \frac{16383}{\text{oxFFFF}}(o_{i+1} - \text{oxFFFF} \varepsilon_{lj}) \tag{4.21}$$

However, the terms $\frac{16383}{\text{oxBFFF}}$ and $\frac{16383}{\text{oxFFFF}}$ are very small and $o_{i+1}$ is the approximation of $l'_{i+1}$ and so the accurate upper shrinking factor is not given by this calculation. Hence, to find the upper shrinking factor, brute-force is to be used. The cost is approximately $2^{10}$ because the range for the factor is between 0.9000 and 0.9999.
4.2.4 Analysis of the output bit sequence

To derive the shrinking factors from the output, it is necessary to identify the output bit sequence for a symbol in the whole output. This section analyzes the output bit sequence, the relationship between a symbol encoded and its resulting output bit sequence, and shows a method to identify the bit sequence produced by encoding a symbol.

There are two factors which hide the direct relationship between an encoded symbol and its output bit sequence. One is the byte buffering, that is, 8 bit blocking of output, and the second is mid-range buffering. Byte buffering is required for the file I/O in the software implementation and merges two consecutive encoding results. It causes the delay in the output but does not change the sequence of the output bits. The mid-range buffering occurs when the range is between 0x0000 and 0x4000. As a result of encoding a symbol \( \psi_i \), some bits may be stored in the mid-range buffer and the final output is produced at the beginning of the output of the following symbol, \( \psi_{i+1} \). If the mid-range buffer, as a result of encoding symbol \( \psi_i \), contains \( n \) bits, then \( n \) bits are inserted between the first bit and the second bit of the output of the symbol \( \psi_{i+1} \). These \( n \) bits are opposite of the first bit, i.e., if the first bit is 0, the \( n \) bits are all 1 and if it is 1, the \( n \) bits are all 0. Hence it is necessary to identify the inserted bits to correctly derive the low value from the output.

Method to identify the output bit sequence

After synchronization of the model, assume that the total frequency is close to \( C_{\text{max}} \), i.e. 16383, and symbols of frequency = 1 are encoded. The following properties of the output bit sequence are useful in finding the beginning of a symbol in the output.

1. If the first two bits are the same, there is no mid-range buffering. This is because the inserted \( n \) bits are always opposite of the first bit.

2. The first bit of the output is always 0.

From (4.12), (4.13), (4.14) and (4.15), the range just before the output is as follows.

**Pattern 1** \( h_i = 0xBFFF, l_i = 0x0000 \),

\[
0x0004 \leq h_{i+1} \leq 0x1334 \\
0x0002 \leq l_{i+1} \leq 0x1331
\]  

(4.22)
Pattern 2 \( h_i = 0xFFFF, l_i = 0x4000, \)
\[
\begin{align*}
0x4004 & \leq h_{i+1}' \leq 0x5334 \\
0x4002 & \leq l_{i+1}' \leq 0x5331
\end{align*}
\] (4.23)

Pattern 3 \( h_i = 0xFFFF, l_i = 0x0000, \)
\[
\begin{align*}
0x0006 & \leq h_{i+1}' \leq 0x199A \\
0x0003 & \leq l_{i+1}' \leq 0x1996
\end{align*}
\] (4.24)

All the above ranges produce an output which starts with a 0.

3. The same bit sequences repeatedly appear in the output. This is because there are only 6 possible ranges and if there is no mid-range buffering, the same range produces the same bit sequence. So with no mid-range buffering there will be 6 different bit sequence patterns in the output. However, if the mid-range buffering occurs, the content of the buffer will affect the output during the output of the following symbol and so there may be more than 6 patterns in the output.

The method to find the beginning of the bit sequence for a symbol is as follows (Figure 4.5).

1. Choose a bit 0 in the output and assume it is the beginning of the sequence.

   If the length of the output for a symbol is 14 bits, there is a start bit in any 14 consecutive bits of the output. If \( \gamma \) bits are 0, there are \( \gamma \) possible start bits.

2. If the bit after the chosen bit is also 0, there has been no mid-range buffering.

   Notice that since a 14 bit sequence includes a start bit, there must be at least one 0.

3. If \( \sigma \) bits after the chosen bit are 1, the first \( i \) bits, \( 0 \leq i \leq \sigma \), of \( \sigma \) bits may be the result of mid-range buffering. Remove \( i \) bits from the output and construct the sequence without mid-range buffering.

4. Assume that the sequence of 14 bits without mid-range buffering obtained above is the output for a symbol and calculate the lower shrinking factor. Using the method given in Section 4.2.3, calculate the value for lower shrinking factor for each possible range.
4.2. Attack on a combined scheme

Since there are 3 possible ranges, 3 calculations per symbol is required to obtain the lower shrinking factor. The factor may include error but it is no larger than 0.0001. Hence the cost to find the factor is $3 \times 2$.

5. Verify the result using an encoder. Encode a symbol of frequency = 1 with the total frequency close to 16363, applying the lower shrinking factor obtained in the above step. Compare the result with the original bit sequence. If the shrinking factor is correct, it should produce exactly the same sequence as the original. If it produces a different sequence, it is either because the assumption of the start bit is wrong or the shrinking factor is wrong. Then change $i$ in step 3 and repeat the above procedure.

From our experiments, for the sequence produced by encoding symbols at the bottom of the frequency table, the number of mid-range buffered bits is generally small. Hence the output produced by encoding one of these symbols, including mid-range buffering, will not be much larger than 14 bits. In a bit sequence of slightly longer than 14 bits, if there are $\gamma$ possible start bits, ie. 0s, and each of them produces $\beta_j$, $1 \leq j \leq \gamma$, possible bit sequences because of the removing of the possible mid-range buffered bits, there are $\Sigma_{j=1}^{\gamma} \beta_j$ possible sequences for a symbol. Each sequence is verified by the above procedure (Figure 4.6). When the sequence for the first symbol is determined, it automatically determines the bit sequence for the next symbol in the output and results in a chain of possible bit sequences which grows like a tree. If the next sequence does not satisfy the conditions, ie. i) it does not start with 0, ii) the
4.2. Attack on a combined scheme

mid-range buffered bits that are inserted into the output of the next symbol result in a conflicting number, or iii) applying the shrinking factor does not result in the expected sequence, then the sequence is wrong and hence the branch is pruned and does not grow further.

Once one of the lower shrinking factors is found, the 128 bit random sequence can be found by using the encoder, to encode symbols of small frequencies in a way similar to that used in the verification of the result. By using the lower shrinking factor to encode the symbols, all parameters, including the high and low values and the size of the mid-range buffering, can be found. If the experimental results differ from the original sequence, the original bit sequence is analyzed in the same way and the other lower shrinking factor needs to be calculated. Once the two lower shrinking factors are known, the 128 bit random sequence can be found by applying either of the two shrinking factors. From our experiments, it appears that the verification process, i.e. the comparison between the experimental results using the lower shrinking factor found and the original bit sequence, is sensitive to the output size of the encoding of a symbol. When the output size is smaller than 14 bits, the verification fails. Hence the sum of
4.3 Conclusion

All the frequencies, $C_\text{h}(\psi_{\text{max}})$, must be large enough to produce 14 bit output for each symbol.

4.2.5 Cost of the attack

The cost of finding the beginning of output bit sequence for a symbol is at most $14 \approx 2^{3.8}$ and the cost of calculating the lower shrinking factor is $3 \times 2 \approx 2^{2.6}$. To verify the lower shrinking factor, it is necessary to encode a symbol using the upper shrinking factor, which is unknown. However, the influence of the upper shrinking factor is very small and so it is only sufficient to only try the maximum, i.e. 9999, the minimum, i.e. 9000, and a few values in between. The total cost is $2^{6.4} + \theta$ where $\theta$ is the cost for the verification.

The unknown part of the key is only the two upper shrinking factors, each of which could take a value from 0.9000 to 0.9999. Hence, the attack reduces the unknown key to $2^{10} \times 2^{10} = 2^{20}$ trials.

4.3 Conclusion

As shown above, there are various methods of exploiting properties of arithmetic coding to attack an arithmetic coding encryption system. The changes in the length of the output gives information about halvings. Once the model is under control, the coder can also be controlled using the knowledge of the model. Encoding takes the cumulative frequencies, the total frequency and the high and low values as parameters. Encoder's output which can be seen as a value, leaks information about these parameters, and so can be used to determine the state of the model and the coder. To protect the system, these properties should be carefully hidden from an attacker.
As can be seen in the previous chapters, there are various methods to attack arithmetic coding encryption schemes. Since different attacks use different properties of the schemes, different protection methods are required. In this chapter we first analyze properties that are used in the attacks, then propose various protection strategies against the attacks and examine their effectiveness. The strategies are i) periodic reset/initialization strategy in which the model is periodically reset/initialized during the encoding of a message, ii) random halving point and iii) extra-narrowing which is used in LFB scheme, iv) higher precision for the model and the coder and v) the higher order of the model. Then we propose new methods, ie. blocking, XORing and permutation of output bits, and demonstrate how each can add security to a system.

5.1 The properties used in the attacks

Different attacks use different properties of arithmetic coding. Although each may not reduce security to a large extent but by combining more than one method, it is possible to dramatically reduce the security of the system. In this section, we give an overview of the attacks and the properties, and examine how they are used in the attacks. The common methods used in the attacks are as follows.

5.1.1 Repeated experiments

LBD attack uses repeated experiments with the encoder. Repeated experiments allow the information about the secret, ie. the model, to be extracted while a single experiment will not give enough information to break a system. If a system goes back to the same state, ie. the initial model, by a reset, it can be repeatedly investigated for different input messages and hence more and more secret information is leaked. The weakness of the system is that it goes back to the same state and so an attacker has
more opportunity to investigate the system.

5.1.2 Controlling the model by an input message

In BH attack, an input message is used to modify the model into a known form. The attack is divided into two steps. The first step is to modify the frequencies of the symbols into specific values. The second step is to arrange symbols in a specific order in the model. After this step the model is completely known to the attacker.

Ordering the symbols is not a theoretical requirement for arithmetic coding but is used for less processing time. Obviously if the model does not use an ordered frequency table, BH attack is only successful to modify the frequencies and not the order of the symbols. However this will significantly slow down the processing speed.

If a chosen plaintext attack is used, an attacker can choose a message which updates the model into an intended form. Since an adaptive model is updated based on an input message, any adaptive system can be attacked by this type of chosen plaintext attack.

5.1.3 Controlling the coder by an input message

In strengthened BH attack and the attack against LFB scheme, an input message is used to control the coder. If a symbol of a small probability is encoded, a long bit sequence is output. This means that the same number of bits are removed from the high and low values and the removed bits in the high and low values are filled by known values, ie. 1 and 0, respectively. Hence the secret part of the high and low values are reduced by the number of output bits. If a few bits are unknown, there are only few possible values that the high and low values can take and so it is possible to force the high and low values to take specific values. This attack is only possible if the model is known because it is necessary for an attacker to know which symbols have small probabilities.

5.1.4 Analysis of the output considered as a real number value

In the attack against LFB scheme, the shrinking factors are calculated from the output. Using the method in Section 5.1.3, a long output is produced by using a symbol of small probability. The high and low values will take one of the few possible values and the result of encoding a symbol is obtained as an output. Since the encoding calculations are simple, it is easy to calculate the shrinking factor from these values.
In the attack on the coder (Section 4.1.2), the analysis is performed on the output by considering it as a real number value. If two encoders having the same model but different initial high and low values encode the same message, there is a certain relationship between the two resulting outputs. If the high and low values of one of the encoders is known, those of the other one can be recovered by comparing the two outputs.

The output is the result of a calculation that uses cumulative frequencies of symbols and the range. The calculations corresponding to the encoding of a single symbol and updating of the model, are very simple. A message is encoded by repeating this calculation. Encoder's output is a real number between 0 and 1, which represents the interval of the message. Since the output is a real value, it is possible to apply operations such as addition and subtraction to the output. Although the inverse of the calculation which produces the output from a large number of input symbols is complex and hence expensive and difficult to solve, the calculation for a single symbol, same as the calculation performed in the attack on LFB scheme, is simple and can be easily used in the attacks.

5.1.5 Analysis on the output size

The change of the output length per symbol can be used to detect the halving points. The probability of a symbol directly affects the length of the output. Halving causes noticeable change in the probabilities of symbols when the same symbol is repeatedly sent to the encoder, and so results in noticeable change in the output length.

5.1.6 Properties exploited to attack systems

From the above discussion, we can summarize the properties used in the attacks. They are:

1. Reset to the same state
2. Adaptiveness of the model
3. Ordering of symbols in the frequency table
4. Replacement of output bits of the high and low values by known values
5. Relationship between the probability of a symbol and the output length
6. Simplicity of encoding calculation
5.2 Protection against the attacks

Noting the properties used in the attacks, to provide protection against the attacks, the following strategies can be used.

5.2.1 Reset/initialization

It is obvious from the above observation that after reset, a system should not return to the same state. That is, a reset should not be limited to default values. Rather, non-key parameters of the system must be changed in every reset. That is, resetting should result in a slightly different range each time. This will protect against attacks which attempt to obtain information about the key by repeated experiment on the same encoder state.

In the model-based scheme, it is possible to initialize the model when a new message is encoded. If input messages with the same probability distribution are consecutively encoded, initialization results in a drop in the compression ratio as all previous adaptation of the model is lost. If the model is used as a key, the model is randomly chosen and it is very unlikely that the model represents the probability distribution of the message that is to be encoded. Hence, the compression ratio for the first part of the message will be worse than the case that the initial model matches the probability distribution of the message. Cleary et al estimate that for English text 1,000 bytes will be enough for the model to adapt.

To protect against BH attack, Lim et al suggested periodic initialization of the model. However as they pointed out, if the initialization period is short, the compression ratio will be dropped.

To initialize the encoder, because of the large number of required key bits, a pseudo random number generator (PRNG) can be used, and the key will be the seed value for the PRNG. During the initialization, the seed of the PRNG is loaded and the output of the PRNG is used to initialize the model and the range. In each reset, the PRNG output is used to reset non-key parameters of the system. If the same seed is used in the transmitter and the receiver and resets in the two are synchronized, encoding and decoding will be correctly performed.

Transmission errors may result in a loss of synchronization between the encoder and the decoder. In such a case, the reset of non-key parameters may not be sufficient and re-initialization of models may be needed because as a result of decoding a bit sequence which is changed by the transmission errors the decoder's model may be in
a state different from the encoder's state. To initialize the encoder and the decoder using the PRNG output, the initialization of the model and non-key parameters in the encoder and in the decoder must be from the same part of the PRNG output bit sequence. The condition above must be satisfied regardless of the initialization algorithm used, otherwise the decoder cannot correctly decode the encoded messages. Since the encoder and the decoder share the same seed for the PRNG and consume the same amount of the PRNG output for each reset and initialization, they will regain the synchronization after transmission errors by the initialization. To regain the synchronization it is important that the encoder and the decoder are simultaneously initialized and to achieve this, an appropriate error control strategy will be needed.

5.2.2 Randomizing the halving periods

BH proposed to randomly choose $C_{\text{max}}$. We note, as shown above, that detection of the halving point will not be affected by the randomization as the halving points can be directly detected from the output and successful re-ordering is guaranteed as long as a halving point is correctly detected.

However, the attack to reduce the secret in the coder (Section 4.1.2) may be disturbed. The attack requires the small probability symbols to produce long enough outputs and hence a small number of unknown bits in the high and low values. It is best if the attack is used right before a halving to minimize the number of unknown bits in the high and low values. This is because when $C_h(\psi_{\text{max}})$ is maximized, the probability of a symbol is minimized. If a halving occurs before the start of the attack, $C_h(\psi_{\text{max}})$ is relatively small and hence, the probabilities of symbols are not small enough to produce a long output. Although it is possible to run the procedure before maximizing $C_h(\psi_{\text{max}})$, the output size is smaller and hence the number of unknown bits in the high and low values is larger and the attack becomes more expensive. Once the attacker successfully decrypts the communication (by modifying $C_h(\psi_{\text{max}})$), it will be possible to adjust her/his decoder to the changes in the halving period.

The impact of this strategy on the compression ratio can be examined by noting that compression ratio depends on the halving period used. The shorter the halving period is the quicker the latest statistics becomes dominant in the model. When a halving occurs, all frequencies are halved and hence each symbol in the previous part of a message is counted as 0.5 while each symbol in the latest part is counted as 1.

---

1 The details of the reset/initialization algorithms are not described in this thesis. The detailed analysis will be required to achieve the secure reset/initialization.
and so the latest part becomes dominant in the model. If the probability distribution of symbols does not change through the whole message, the halving period will not have much influence on the compression ratio. However, for a message with changing distribution, the compression ratio will depend on how fast the latest distribution becomes dominant in the model. In other words, the halving period determines the speed of adaptation of the model. The optimal period varies with messages and if the period is randomly chosen, it is unlikely that it is optimal for a message. Hence this strategy restricts the optimal choice of the halving period.

Examples of compression ratios with different halving periods are shown in Table 5.1. The files are from Calgary/Canterbury text compression corpus [WB89]. In the initial model, all frequencies are set to 1. For English text sources, larger halving periods result in better compression but for the object files and the source codes, if the periods are larger than certain values, reduction in the ratios will be reduced. For objl file, the best compression is achieved at $C_{\text{max}} = 1,000$ and the compression ratio drops by about 7.5% when $C_{\text{max}} = 16,000$. Similarly, obj2 file is best compressed when $C_{\text{max}} = 2,000$ and has about 2.5% drop in compression ratio when $C_{\text{max}} = 16,000$. This could be because object files will have specific structures, such as references to external entries, static data and instructions and so it is quite likely that each part has different probability distribution of symbols. The quicker the model adapts to the source statistics, the better the compression ratio will be. In such cases, the halving period will have influence on the compression ratio.

<table>
<thead>
<tr>
<th>File</th>
<th>Size</th>
<th>Contents</th>
<th>Halving period</th>
</tr>
</thead>
<tbody>
<tr>
<td>bib</td>
<td>111261</td>
<td>English text</td>
<td>500 1,000   2,000 4,000 8,000 16,000</td>
</tr>
<tr>
<td>book1</td>
<td>788771</td>
<td>English text</td>
<td>6.144 5.625 5.405 5.304 5.256 5.234</td>
</tr>
<tr>
<td>geo</td>
<td>102400</td>
<td>Geophysical data</td>
<td>5.904 5.701 5.657 5.652 5.654 5.656</td>
</tr>
<tr>
<td>obj1</td>
<td>21504</td>
<td>Compiled code for Vax</td>
<td>5.743 5.551 5.618 5.743 5.866 5.968</td>
</tr>
<tr>
<td>obj2</td>
<td>246814</td>
<td>Compiled code for Mac</td>
<td>6.219 5.936 5.891 5.929 5.997 6.067</td>
</tr>
<tr>
<td>progc</td>
<td>39611</td>
<td>C source code</td>
<td>5.910 5.457 5.297 5.243 5.227 5.234</td>
</tr>
<tr>
<td>progp</td>
<td>49379</td>
<td>Pascal source code</td>
<td>5.617 5.131 4.962 4.900 4.890 4.894</td>
</tr>
<tr>
<td>gs</td>
<td>689352</td>
<td>Compiled code for Linux</td>
<td>5.983 5.709 5.669 5.705 5.763 5.827</td>
</tr>
</tbody>
</table>

Table 5.1: Halving periods and compression ratios (bits/symbol)

5.2.3 Extra-narrowing of interval

This method is used by LFB scheme and was claimed to be resistant against BH attack. However, as long as it is possible to detect halving points, this mechanism can
not avoid over-flooding and re-ordering and so it cannot prevent the attack. The aim of this approach is to add extra-complexity to the coder and to make synchronization of the coder more difficult. However as shown in the previous chapter, it is possible to largely reduce the attacking cost and hence the method does not provide strong security.

The cost of this strategy is nearly doubling the processing time of the input data. This is because extra-narrowing is equivalent to encoding another symbol and the encoding procedure takes the major part of the processing time.

### 5.2.4 Higher precision of the model and the coder

Using 32 bit integer arithmetic for the frequencies in the model will allow setting of $F_{\text{max}}$ to much larger values and hence longer strings for over-flooding are required. This provides protection against the BH attack. For example, if $F_{\text{max}} = 268,000,000$, a string of more than 100,000,000 symbols is required to cause a halving. This will increase security but will affect the compression ratio because increase in halving period means slower adaptation to source statistics which could drop the compression ratio. More detail is given in Section 5.2.2.

To increase coder’s security, higher precision for the high and low values can be used. Assume the number of bits for the high and low values is $B_r$. If the smallest probability in the model is $P_{\text{min}}$, then the maximum length of the output for a symbol is $-\log_2 P_{\text{min}}$ bits. During the output procedure the output bits are removed from the high and low values and $-\log_2 P_{\text{min}}$ bits of the high and low values are filled with known values, i.e. 1 and 0. So the number of unknown bits in the high and the low values is $B_r + \log_2 P_{\text{min}}$. In the case of WNC implementation, the number of unknown bits is $16 - 14 = 2$. If the high and the low values are 32 bits and the same model is used, the number of unknown bits becomes $32 - 14 = 18$ bits. In this case, even if the symbols of the smallest probability are sent, the high and the low values can take $2^{18}$ possible values and hence coder’s security is increased. The cost of employing higher precision is additional calculation time.

### 5.2.5 Higher order of the model

To avoid BH attack, the length of the sequence required for synchronization can be increased. A method of increasing this length is to use higher order models such as PPM models. If the sequence is long enough (compared with the size of most messages),
then by using a reset that is automatically invoked after a certain length input, the attacker will be effectively stopped from synchronizing the model. If the reset interval is chosen longer than most messages, it will not have any effect on the data compression performance.

Irvine [Irv95] showed a method of attacking PPM system using BH approach. In this attack it is necessary to produce all possible nodes of the tree. To protect against this attack he suggested a method of pruning the model using a pseudo-random sequence.

An important consideration in using higher order models is the required memory and processing time. Although using higher order models will result in better compression ratio, because of memory and speed requirements it might not be suitable for many applications.

In general requiring longer sequences to control the model implies slower adaptation of the model to an input. Since the initial model is randomly chosen (key), it may be far from optimal for an incoming message and so slower adaptation will result in a drop in compression ratio.

5.2.6 XORing the output with pseudo-random sequence

Various methods can be used to protect against mathematical analysis of the output and hiding the true output of the encoder. A simple method is to mask the output by XORing the output with a pseudo-random sequence. The seed of the pseudo-random generator is part of the key and the encoder's output is XORed with the sequence.

![Encryption of output using random number sequence](image)

Figure 5.1: Encryption of output using random number sequence

This system is attractive because the PRNG need not be cryptographically strong and a linear feedback shift register (LFSR) will suffice. This is because the coder output is compressed and hence has very high entropy and the role of PRNG is to simply mask this output. This means that masking does not introduce large computation cost and
no notable degradation of speed will occur. However the weakness of this system is that if the true encoder output is known, the pseudo-random sequence can be revealed and in the case of LFSR systems, the seed of PRNG will be determined. As shown above, one of the proposed attacks (Section 4.1.2) produces a known output bit sequence and so can be used to find the output of LFSR and derive the seed.

5.2.7 Blocking

Another method of providing protection against attacks is blocking that is controlled by the encryption system. That is, an output will be released as blocks of bits where the size of the block is fixed and determined by the combined encryption-compression software. Blocking muddles the relationship between input symbols and the output bit string and makes the analysis of the output more difficult. However, if it is the sole method of protection, the added strength is not sufficient. This is because the relationship between the last output symbol and the last block is clear and it is known that all or part of the substring for the last symbol will be in the block. More security can be obtained by permuting bits in a block.

5.2.8 Permutation

Using permutation of encoder's output, it is possible to hide the properties of the output and the relationship between input and output, and so it is more difficult to do the inverse calculation of encoding. For the permutation to be executed on a fixed size output block, the output will be buffered and hence, blocking is applied.

The weakness is that if the true output bits are known, they may leak information about how each bit is permuted. For example, the attack in Section 4.1.2 can produce an output of known bit patterns and because the patterns are known and they are repeated in the output, it could leak information about the permutation.

XORing can hide the true information so if it is combined with permutation, it will give better security to the system. The advantage of this method is that it is possible to have a less expensive and more secure system by combining simple algorithms.
5.3 Conclusion

Attacks use different properties and so protection mechanisms vary with attacks. Adding security in general reduces the compression ratio but the amount of reduction depends on the chosen method. Repeated experiments can be easily avoided by reset-initialization strategy. Attacks that use adaptiveness of the model are more difficult to avoid without sacrificing compression ratio. By applying higher precision for the frequencies in the model, the cost of the attack will grow to impractical level but the speed of adaptation will drop. Higher order models possibly provide better security but with much larger memory and to some extent computational requirement. This limits their applications. Though higher precision for the coder does not provide strong enough security by itself, combined with other methods, it may result in reasonably good security. The cost may not be a problem because 32 bit architecture, and hence 64 bits for the result (multiplying two 32 bit numbers), is becoming common in CPU technologies. Analysis of encoder's output can be avoided by hiding the true output. The security of XOR/permutation/blocking is discussed in the following chapter.
Chapter 6

Comments on integrated encryption after compression schemes

In arithmetic coding encryption systems the secret key consists of the model and the coder, individually or together. The model has a larger number of variables (frequency counts of the symbols) than the coder (the high and low values). Although there are some additional variables in the LFB scheme, the model has many more variables. Most of the parameters change dynamically with an input message and security is achieved by hiding the initial values of the parameters. In this chapter, we analyze security of the methods discussed in the previous chapter. Firstly we introduce two general models of hiding compressed output. Then we compare the two models when i) pure compression followed by encryption is used, and ii) a secure compression algorithm is used. Finally we summarize the result.

6.1 Methods to hide encoder’s output

Encoder’s output leaks information. To achieve strong security, it is necessary to hide the true output of the encoder. As shown in the previous chapter, XOR, permutation and blocking could possibly enhance the security. In this section, we analyze each of these techniques and examine their properties.

6.1.1 Models to hide the true output

Type 1 This is the simple combination of compression and encryption algorithms. The encryption is performed after compressing the input. In this case encryption algorithm input depends on the output of the compression and hence, indirectly on an input message.

Type 2 In this system encoder’s output is masked by the output of the encryption part. The assumed encryption algorithm is a pseudo-random generator which
6.2 Comparison between Type 1 and Type 2

We assume that the data compression algorithm is an adaptive arithmetic coding and its performance in both cases is the same. Also in both cases the encryption algorithm provides the same level of security and has the same cost. We also assume that the compression algorithm does not provide security, i.e., it is a pure compression algorithm. The encryption is assumed to be a block cipher.

6.2.1 Speed

In terms of throughput, both systems are equivalent. However, Type 1 introduces a larger delay in producing the output from the input.

Let $\delta_{\text{comp}}$ and $\tau_{\text{comp}}$ be the average required time to compress a symbol and to output a block, respectively. Let $\delta_{\text{enc}}$ and $\tau_{\text{enc}}$ be the average required time to encrypt a block and to output an encrypted block, respectively. We assume that the compression and the encryption can be executed in parallel in Type 2 system. In Type 1 system, the encryption process has to wait for the output of the compression. We also assume that the procedures of compression and encryption ($\delta_{\text{comp}}$ and $\delta_{\text{enc}}$) and outputting their results ($\tau_{\text{comp}}$ and $\tau_{\text{enc}}$) can be executed in parallel. This allows overlapping of $\delta_{\text{comp}}$ and $\delta_{\text{enc}}$ with $\tau_{\text{comp}}$ and $\tau_{\text{enc}}$ respectively. The values of $\delta_{\text{comp}}$ and $\tau_{\text{comp}}$ may vary with symbols' probabilities but for the simplicity, we assume that the required time is equal.

In Type 1 system, the delay of the output from its input is given by $\delta_{\text{comp}} + \tau_{\text{comp}} + \delta_{\text{enc}}$. We assume that the time required for XOR is very small and hence, can be ignored.

In Type 2 system, the delay to produce the output from the corresponding input is $\delta_{\text{comp}} + \tau_{\text{comp}}$ when $\delta_{\text{comp}} + \tau_{\text{comp}} \geq \delta_{\text{enc}} + \tau_{\text{enc}}$ and $\delta_{\text{enc}} + \tau_{\text{enc}}$ when $\delta_{\text{comp}} + \tau_{\text{comp}} < \delta_{\text{enc}} + \tau_{\text{enc}}$. So the throughput is determined by the slower of the compression or the encryption.
6.2. Comparison between Type 1 and Type 2

<table>
<thead>
<tr>
<th>Compression input</th>
<th>Compression output</th>
<th>Encryption input</th>
<th>Encryption output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 symbol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \delta_{\text{comp}} ) : delay of the compression output from the input (time required to compress 1 symbol)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \tau_{\text{comp}} ) : time for the compression part to produce 1 block output for the encryption part</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \delta_{\text{enc}} ) : delay of the encryption output from the input (time required to encrypt 1 block)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \tau_{\text{enc}} ) : time for the encryption part to produce 1 block output</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\delta_{\text{comp}} & : \text{delay of the compression output from the input (time required to compress 1 symbol)} \\
\tau_{\text{comp}} & : \text{time for the compression part to produce 1 block output for the encryption part} \\
\delta_{\text{enc}} & : \text{delay of the encryption output from the input (time required to encrypt 1 block)} \\
\tau_{\text{enc}} & : \text{time for the encryption part to produce 1 block output}
\end{align*}
\]

Figure 6.2: Speed of Type 1 system

<table>
<thead>
<tr>
<th>Compression input</th>
<th>Compression output</th>
<th>Encryption input</th>
<th>Encryption output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 symbol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \delta_{\text{comp}} ) : delay of the compression output from the input (time required to compress 1 symbol)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \tau_{\text{comp}} ) : time for the compression part to produce 1 block output for the encryption part</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \delta_{\text{enc}} ) : delay of the encryption output from the input (time required to encrypt 1 block)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( \tau_{\text{enc}} ) : time for the encryption part to produce 1 block output</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\delta_{\text{comp}} & : \text{delay of the compression output from the input (time required to compress 1 symbol)} \\
\tau_{\text{comp}} & : \text{time for the compression part to produce 1 block output for the encryption part} \\
\delta_{\text{enc}} & : \text{delay of the encryption output from the input (time required to encrypt 1 block)} \\
\tau_{\text{enc}} & : \text{time for the encryption part to produce 1 block output}
\end{align*}
\]

Figure 6.3: Speed of Type 2 system

6.2.2 Security

Both systems are capable of hiding the true output of the compression system. However, Type 2 system has a restriction on the type of the encryption system. Since the encryption output is XORed with the compressed output, the encryption of Type 2 system is effectively a substitution cipher. When the output of the compression part is known, the output sequence of the encryption part can be revealed by a plaintext attack. Since the compression system is assumed not to provide any security, an attacker can easily reproduce the compressed output if an input message is known. Then the sequence produced by the encryption part can be discovered by XORing the reproduced sequence and the output of Type 2 system. Since a simple pseudo-random generator is known to be weak, the system cannot provide strong security. In case of Type 1 system, there are more choices for the encryption algorithm. It can be a substitution cipher, a permutation or a combination of both.

For both systems, the security strength is determined solely by the security of the...
6.3 Pure compression versus compression-encryption schemes

In both Type 1 and 2 systems, the compression algorithm can be either a pure compression or a compression-encryption scheme. In the following, we look at the benefits of replacing a pure compression algorithm by a compression-encryption algorithm.

6.3.1 Controlling input messages of encryption algorithm

It will be much more difficult to produce a specific output sequence in a system with a combined compression-encryption algorithm than with a pure compression algorithm. This is because there is a considerable number of unknown parameters in the compression part of a compression-encryption system. BH attack may be used to reveal the parameters but it is more costly. That is, the cost of BH attack to make the parameters known must also be included in the assessment. However, if BH attack is feasible (in terms of the cost,) the system does not provide strong security. Hence if the compression-encryption system itself does not provide the resistance against BH attack, the whole system will not be resistant against BH attack either.

6.3.2 Resistance against statistical attacks

For the encryption part of the system, statistical attacks would become more difficult because the input to the encryption algorithm is compressed and hence has less redundancy. However, statistical attacks against the compression-encryption part would work because the model is adaptive and hence reflects the statistical properties of the input sequence. When the model is simple, ie. models of lower order, if the statistical
behavior of the source is known, it would not be difficult to estimate the parameters of the model. Security of the higher order model is still an open problem but the behavior of the model is more or less the same. In general, in arithmetic coding, re-synchronizing the system in the middle of transmission is difficult. However, the difficulty is from communication point of view and not security point of view and even with extra measure the system still cannot provide enough security.

6.3.3 Efficiency

As long as a lower order model is used, the combined compression and encryption scheme does not provide strong security. The question to be answered is whether adding secret parameters into the coder is more efficient, or adding encryption algorithm to the coder. Taking into account factors such as speed, compression ratio, and flexibility of the design, with a careful choice of the encryption algorithm, adding encryption after compression may have numerous benefits. In general regardless of the chosen approach, the drop in the compression performance should be minimized. Using the encryption after compression is not limited to arithmetic coding compression but can be used with any other compression method. Making the two processes separate results in each algorithm in the system to be simple and less expensive. This also requires fewer extra-bits in the output which will be only added to the padding of the last block of the output. The impact on the compression performance, that is, compression ratio and compression speed, will also be negligible compared to methods such as extra-narrowing.

6.4 Conclusion

Lower order models cannot provide strong security. Adding secret parameters to the coder could improve security but it would impose restrictions on the system. By adding encryption algorithm after arithmetic coding encryption schemes, security can be improved and more flexibility can be obtained. The advantage of this approach is that it can be used in combination with other compression algorithms.
Chapter 7

Conclusion

Secure compression schemes have various benefits. By achieving compression and security through a single algorithm, less overhead could be expected. In general efficiency is improved by compressing data before encryption. This goal is also achieved by secure compression schemes. Arithmetic coding is an optimal coding scheme. If it can also provide security the benefits are achieved. Adaptive arithmetic coding encryption schemes have a large key space (initial model), and it is difficult to re-gain the synchronization in the middle of the transmission. These properties make them attractive for providing security.

We had a careful study of the arithmetic coding encryption schemes and demonstrated various attacks. We proposed a number of protection mechanisms and looked at their security. In this chapter, first we summarize the schemes and then review the attacks. Next we comment on the weaknesses of the schemes and their security, and conclude with some final remarks.

7.1 Schemes

The schemes are divided into two classes: model-based schemes which use the model as the secret, and coder-based schemes in which additional secret parameters are used to strengthen the security of the coder. The two classes can be combined together. The secret key of a model-based arithmetic coding encryption scheme is the initial model which may be explicitly given as a key or may be the result of updating by the initial string. An attack against the schemes may try to discover the key, i.e. the initial model, or try to gain synchronization in the middle of the transmission without finding the key. As the attacks show, both approaches are valid.
7.2 Attacks

BH attack shows that it is easy to reduce the key space by a chosen plaintext attack when the order of the model is low. We showed that the output of the encoder leaks much more information than expected. The average output length per symbol can give information about halving points. When the model is known, it is easy to discover coder's parameters, i.e. the high and low values, from encoder's output and analyzing encoder's output as a value, is useful in some attacks. The output obtained by encoding a symbol at the bottom of the frequency table leaks the information about the low value. Some of the attacks use the property that the encoding/decoding calculation for each symbol is simple although the calculation of a whole message will include high order parameters. In the calculation, only five parameters are involved, i.e. the upper and lower cumulative frequencies of a symbol, the total cumulative frequency, and the high and low values.

7.3 Weaknesses and security

One of the weaknesses of the schemes is the adaptiveness of the model. Since the model is updated as to follow the distribution of symbols in a message, it can be manipulated by an input message. This can be exploited by statistical attacks when distribution of the source is known. Using the higher order model may be one way to improving security. As shown in [Ir95], BH attack will require a much longer message to synchronize the model and hence it becomes more impractical. However, the level of security against BH attack and other statistical attacks is still unknown. Also the disadvantage of the higher order model is its memory requirement. Lower order models will achieve less compression compared to the higher order models but its smaller memory requirement could be attractive for various applications.

If the model cannot provide strong security, the security of the system has to rely on the coder. The coder has little security without any additional mechanism and so the coder-based schemes use additional mechanisms. As shown in the attack, known coder-based schemes do not provide strong security. Adding other secret parameters to the coder may strengthen the security but they would impose restrictions on the design.

Adding encryption algorithms such as XORing with a pseudo-random generator and a permutation can be a better choice. It will give much flexibility in the design and it
can be also used with other types of compression systems. Also compared to the model-based schemes a better resistance against statistical attacks can be expected. The adaptive model itself would not have strong resistance against statistical attacks due to its adaptive nature. However, the additional encryption algorithm after combined encryption and compression, will have an input with less redundancy (compressed by the encoder), and so it has the advantage against the attacks. To achieve a certain level of security, encryption after a combined compression and encryption system will require less security than the encryption after a compression only system. The important point for the added encryption system is that the compressor and the encryption algorithm should be indivisible. Otherwise the encryption part can be independently attacked.

7.4 Further work

As we mentioned above, the security of arithmetic coding encryption schemes with the higher order model is an open problem. There are several questions to be answered such as whether or not it is possible to improve BH attack so as to reduce the message size required to over-flood the model, and how much resistant against statistical attacks the model is. There are different types of higher order models such as PPMA [Irv95], PPMZ and PPM* [Blo98] which have different properties [Irv95]. Then an interesting question would be whether or not it is possible to design a model which takes the security into account.

To choose an initial model, it is generally said that the model is randomly chosen. However the conditions that the random model must satisfy are not described in details. For example, when the frequency of each symbol is uniformly chosen from the values between 1 and 10 then the cardinality of the set of symbols that have the same frequency is roughly the same for all values from 1 to 10. Then two models initialized in this way will have the same frequency sets but different order of symbols. For such models, discovering the order is easy by a plaintext attack. For example, when A is encoded and the result of attacker's decoder is X, by replacing X by A, it is possible to correct the order of symbols. In this case, the order of symbols does not provide any security. This means that to design a practically secure system, much more details of the system must be examined.
Bibliography


[ICRM95] Sean A. Irvine, John G. Cleary, and Ingrid Rinsma-Melchert. The subset sum problem and arithmetic coding

[Irv95] Sean A. Irvine. PhD thesis


[WNC87] Ian H. Witten, Radford Neal, and John G. Cleary. The code that was printed in the COMM ACM 1987 article

Appendix A

Finding the high and low values of the coder

A.1 Experiment

In this experiment the method described in Section 4.1.2 is used.

The original compressed data was produced using a message “Test message” and the high and low values 54321 (0xd431) and 12345 (0x3039) respectively. Then 0 was set to the low value and the high value was changed from 32769 to 65535, one at a time, and in each case the encoder’s output was compared with the original output. The experiment was run on Pentium 150 MHz machine with 32M bytes memory, with Linux OS.

We found more than one possible high values. For example, the correct pair is (54321, 12345) but also the pairs (54225, 12540) and (54353, 12279) can correctly decode the message. There may be more than one pair of high and low values which can correctly decode.

A.2 Result of the experiment

```plaintext
### message = "Test message"
low  = 12345 (0x3039)
high = 54321 (0xd431)
range = 41976 (0xa3f8)
### low =  0 (0x0000)
range = 32769 (0x8001)
increment by 1 (0x0001)
while high <= 65535 (0xffff)
### ( 1) low  =  0 (0x0000) high  = 41685 (0xa2d5)
```
A.2. Result of the experiment

6d 5c 44 86 ae 14 fc d4 64 f7 09 68 48
30 fc 00 00 00 00 00 00 00 00 00 00 00 00
### (2)  low = 0 (0x0000)  high = 41688 (0xa2d8)
6d 5e 44 86 ae 14 fc d4 64 f7 09 68 48
30 fa 00 00 00 00 00 00 00 00 00 00 00 00
### (3)  low = 0 (0x0000)  high = 41691 (0xa2db)
6d 60 44 86 ae 14 fc d4 64 f7 09 68 48
30 f8 00 00 00 00 00 00 00 00 00 00 00 00
### (4)  low = 0 (0x0000)  high = 41834 (0xa36a)
6d c0 44 86 ae 14 fc d4 64 f7 09 68 48
30 98 00 00 00 00 00 00 00 00 00 00 00 00
### (5)  low = 0 (0x0000)  high = 41836 (0xa36c)
6d c1 44 86 ae 14 fc d4 64 f7 09 68 48
30 97 00 00 00 00 00 00 00 00 00 00 00 00
### (6)  low = 0 (0x0000)  high = 41837 (0xa36d)
6d c2 44 86 ae 14 fc d4 64 f7 09 68 48
30 96 00 00 00 00 00 00 00 00 00 00 00 00
### (7)  low = 0 (0x0000)  high = 41839 (0xa36f)
6d c3 44 86 ae 14 fc d4 64 f7 09 68 48
30 95 00 00 00 00 00 00 00 00 00 00 00 00
### (8)  low = 0 (0x0000)  high = 41840 (0xa370)
6d c4 44 86 ae 14 fc d4 64 f7 09 68 48
30 94 00 00 00 00 00 00 00 00 00 00 00 00
### (9)  low = 0 (0x0000)  high = 41841 (0xa371)
6d c5 44 86 ae 14 fc d4 64 f7 09 68 48
30 93 00 00 00 00 00 00 00 00 00 00 00 00
### (10) low = 0 (0x0000)  high = 41842 (0xa372)
6d c6 44 86 ae 14 fc d4 64 f7 09 68 48
30 92 00 00 00 00 00 00 00 00 00 00 00 00
### (11) low = 0 (0x0000)  high = 41843 (0xa373)
6d c7 44 86 ae 14 fc d4 64 f7 09 68 48
30 91 00 00 00 00 00 00 00 00 00 00 00 00
### (12) low = 0 (0x0000)  high = 41844 (0xa374)
6d c7 44 86 ae 14 fc d4 64 f7 09 68 48
30 91 00 00 00 00 00 00 00 00 00 00 00 00
### (13) low = 0 (0x0000)  high = 41845 (0xa375)
A.2. Result of the experiment

### (14)  low = 0 (0x0000)  high = 41846 (0xa376)
6d c8 44 86 ae 14 fc d4 64 f7 09 68 48
30 90 00 00 00 00 00 00 00 00 00 00

### (15)  low = 0 (0x0000)  high = 41847 (0xa377)
6d c9 44 86 ae 14 fc d4 64 f7 09 68 48
30 8f 00 00 00 00 00 00 00 00 00 00

### (16)  low = 0 (0x0000)  high = 41848 (0xa378)
6d ca 44 86 ae 14 fc d4 64 f7 09 68 48
30 8e 00 00 00 00 00 00 00 00 00 00

### (17)  low = 0 (0x0000)  high = 41849 (0xa379)
6d cb 44 86 ae 14 fc d4 64 f7 09 68 48
30 8d 00 00 00 00 00 00 00 00 00 00

### (18)  low = 0 (0x0000)  high = 41850 (0xa37a)
6d cc 44 86 ae 14 fc d4 64 f7 09 68 48
30 8c 00 00 00 00 00 00 00 00 00 00

### (19)  low = 0 (0x0000)  high = 41851 (0xa37b)
6d cd 44 86 ae 14 fc d4 64 f7 09 68 48
30 8b 00 00 00 00 00 00 00 00 00 00

### (20)  low = 0 (0x0000)  high = 41852 (0xa37c)
6d ce 44 86 ae 14 fc d4 64 f7 09 68 48
30 8a 00 00 00 00 00 00 00 00 00 00

### (21)  low = 0 (0x0000)  high = 41853 (0xa37d)
6d cf 44 86 ae 14 fc d4 64 f7 09 68 48
30 89 00 00 00 00 00 00 00 00 00 00

### (22)  low = 0 (0x0000)  high = 41854 (0xa37e)
6d d0 44 86 ae 14 fc d4 64 f7 09 68 48
30 89 00 00 00 00 00 00 00 00 00 00

### (23)  low = 0 (0x0000)  high = 41855 (0xa37f)
6d d1 44 86 ae 14 fc d4 64 f7 09 68 48
30 89 00 00 00 00 00 00 00 00 00 00

### (24)  low = 0 (0x0000)  high = 41856 (0xa380)
6d d2 44 86 ae 14 fc d4 64 f7 09 68 48
30 89 00 00 00 00 00 00 00 00 00 00

### (25)  low = 0 (0x0000)  high = 41857 (0xa381)
6d d3 44 86 ae 14 fc d4 64 f7 09 68 48
30 89 00 00 00 00 00 00 00 00 00 00

### (26)  low = 0 (0x0000)  high = 41858 (0xa382)
6d d4 44 86 ae 14 fc d4 64 f7 09 68 48
A.2. Result of the experiment

### (27)  low  =  0  (0x0000)  high  =  41859  (0xa383)
6d d1 44 86 ae 14 fc d4 64 f7 09 68 48
30 87 00 00 00 00 00 00 00 00 00 00 00 00 00 00

### (28)  low  =  0  (0x0000)  high  =  41860  (0xa384)
6d d1 44 86 ae 14 fc d4 64 f7 09 68 48
30 87 00 00 00 00 00 00 00 00 00 00 00 00 00 00

### (29)  low  =  0  (0x0000)  high  =  41861  (0xa385)
6d d2 44 86 ae 14 fc d4 64 f7 09 68 48
30 86 00 00 00 00 00 00 00 00 00 00 00 00 00 00

### (30)  low  =  0  (0x0000)  high  =  41862  (0xa386)
6d d3 44 86 ae 14 fc d4 64 f7 09 68 48
30 85 00 00 00 00 00 00 00 00 00 00 00 00 00 00

### (31)  low  =  0  (0x0000)  high  =  41863  (0xa387)
6d d3 44 86 ae 14 fc d4 64 f7 09 68 48
30 85 00 00 00 00 00 00 00 00 00 00 00 00 00 00

### (32)  low  =  0  (0x0000)  high  =  41864  (0xa388)
6d d4 44 86 ae 14 fc d4 64 f7 09 68 48
30 84 00 00 00 00 00 00 00 00 00 00 00 00 00 00

### (33)  low  =  0  (0x0000)  high  =  41865  (0xa389)
6d d5 44 86 ae 14 fc d4 64 f7 09 68 48
30 83 00 00 00 00 00 00 00 00 00 00 00 00 00 00

### (34)  low  =  0  (0x0000)  high  =  41866  (0xa38a)
6d d5 44 86 ae 14 fc d4 64 f7 09 68 48
30 83 00 00 00 00 00 00 00 00 00 00 00 00 00 00

### (35)  low  =  0  (0x0000)  high  =  41867  (0xa38b)
6d d6 44 86 ae 14 fc d4 64 f7 09 68 48
30 82 00 00 00 00 00 00 00 00 00 00 00 00 00 00

### (36)  low  =  0  (0x0000)  high  =  41868  (0xa38c)
6d d7 44 86 ae 14 fc d4 64 f7 09 68 48
30 81 00 00 00 00 00 00 00 00 00 00 00 00 00 00

### (37)  low  =  0  (0x0000)  high  =  41869  (0xa38d)
6d d7 44 86 ae 14 fc d4 64 f7 09 68 48
30 81 00 00 00 00 00 00 00 00 00 00 00 00 00 00

### (38)  low  =  0  (0x0000)  high  =  41870  (0xa38e)
6d d8 44 86 ae 14 fc d4 64 f7 09 68 48
30 80 00 00 00 00 00 00 00 00 00 00 00 00 00 00

### (39)  low  =  0  (0x0000)  high  =  41871  (0xa38f)
A.2. Result of the experiment

6d d9 44 86 ae 14 fc d4 64 f7 09 68 48
30 7f 00 00 00 00 00 00 00 00 00
### (40) low = 0 (0x0000) high = 41872 (0xa390)
6d da 44 86 ae 14 fc d4 64 f7 09 68 48
30 7e 00 00 00 00 00 00 00 00 00 00
### (41) low = 0 (0x0000) high = 41873 (0xa391)
6d db 44 86 ae 14 fc d4 64 f7 09 68 48
30 7d 00 00 00 00 00 00 00 00 00 00 00
### (42) low = 0 (0x0000) high = 41874 (0xa392)
6d dc 44 86 ae 14 fc d4 64 f7 09 68 48
30 7c 00 00 00 00 00 00 00 00 00 00 00 00
### (43) low = 0 (0x0000) high = 41875 (0xa393)
6d dd 44 86 ae 14 fc d4 64 f7 09 68 48
30 7b 00 00 00 00 00 00 00 00 00 00 00 00
### (44) low = 0 (0x0000) high = 41876 (0xa394)
6d de 44 86 ae 14 fc d4 64 f7 09 68 48
30 7a 00 00 00 00 00 00 00 00 00 00 00 00
### (45) low = 0 (0x0000) high = 41877 (0xa395)
6d df 44 86 ae 14 fc d4 64 f7 09 68 48
30 79 00 00 00 00 00 00 00 00 00 00 00 00
### (46) low = 0 (0x0000) high = 41878 (0xa396)
6d d9 44 86 ae 14 fc d4 64 f7 09 68 48
30 78 00 00 00 00 00 00 00 00 00 00 00 00
### (47) low = 0 (0x0000) high = 41879 (0xa397)
6d e0 44 86 ae 14 fc d4 64 f7 09 68 48
30 77 00 00 00 00 00 00 00 00 00 00 00 00
### (49) low = 0 (0x0000) high = 41881 (0xa399)
6d e1 44 86 ae 14 fc d4 64 f7 09 68 48
30 76 00 00 00 00 00 00 00 00 00 00 00 00
### (50) low = 0 (0x0000) high = 41882 (0xa39a)
6d e2 44 86 ae 14 fc d4 64 f7 09 68 48
30 75 00 00 00 00 00 00 00 00 00 00 00 00
### (51) low = 0 (0x0000) high = 41883 (0xa39b)
Result of the experiment

### (52)  low = 0 (0x0000)  high = 41884 (0xa39c)
6d e1 44 86 ae 14 fc d4 64 f7 09 68 48
30 77 00 00 00 00 00 00 00 00 00 00

### (53)  low = 0 (0x0000)  high = 42025 (0xa429)
6e 40 44 86 ae 14 fc d4 64 f7 09 68 48
30 18 00 00 00 00 00 00 00 00 00 00

### (54)  low = 0 (0x0000)  high = 42026 (0xa42a)
6e 41 44 86 ae 14 fc d4 64 f7 09 68 48
30 17 00 00 00 00 00 00 00 00 00 00

### (55)  low = 0 (0x0000)  high = 42028 (0xa42c)
6e 42 44 86 ae 14 fc d4 64 f7 09 68 48
30 16 00 00 00 00 00 00 00 00 00 00

### (56)  low = 0 (0x0000)  high = 42029 (0xa42d)
6e 43 44 86 ae 14 fc d4 64 f7 09 68 48
30 15 00 00 00 00 00 00 00 00 00 00

### (57)  low = 0 (0x0000)  high = 42031 (0xa42f)
6e 44 44 86 ae 14 fc d4 64 f7 09 68 48
30 14 00 00 00 00 00 00 00 00 00 00

### (58)  low = 0 (0x0000)  high = 42032 (0xa430)
6e 45 44 86 ae 14 fc d4 64 f7 09 68 48
30 13 00 00 00 00 00 00 00 00 00 00

### (59)  low = 0 (0x0000)  high = 42034 (0xa432)
6e 46 44 86 ae 14 fc d4 64 f7 09 68 48
30 12 00 00 00 00 00 00 00 00 00 00

### (60)  low = 0 (0x0000)  high = 42035 (0xa433)
6e 47 44 86 ae 14 fc d4 64 f7 09 68 48
30 11 00 00 00 00 00 00 00 00 00 00

### (61)  low = 0 (0x0000)  high = 42037 (0xa435)
6e 48 44 86 ae 14 fc d4 64 f7 09 68 48
30 10 00 00 00 00 00 00 00 00 00 00

### (62)  low = 0 (0x0000)  high = 42038 (0xa436)
6e 49 44 86 ae 14 fc d4 64 f7 09 68 48
30 0f 00 00 00 00 00 00 00 00 00 00

### (63)  low = 0 (0x0000)  high = 42040 (0xa438)
6e 4a 44 86 ae 14 fc d4 64 f7 09 68 48
30 0e 00 00 00 00 00 00 00 00 00 00

### (64)  low = 0 (0x0000)  high = 42041 (0xa439)
6e 4b 44 86 ae 14 fc d4 64 f7 09 68 48
A.2. Result of the experiment

30 0d 00 00 00 00 00 00 00 00 00 00 00
### (65) low = 0 (0x0000) high = 42043 (0xa43b)
6e 4c 44 86 ae 14 fc d4 64 f7 09 68 48
30 0c 00 00 00 00 00 00 00 00 00 00 00
### (66) low = 0 (0x0000) high = 42044 (0xa43c)
6e 4d 44 86 ae 14 fc d4 64 f7 09 68 48
30 0b 00 00 00 00 00 00 00 00 00 00 00
### (67) low = 0 (0x0000) high = 42047 (0xa43f)
6e 4f 44 86 ae 14 fc d4 64 f7 09 68 48
30 09 00 00 00 00 00 00 00 00 00 00 00
### (68) low = 0 (0x0000) high = 42050 (0xa442)
6e 51 44 86 ae 14 fc d4 64 f7 09 68 48
30 07 00 00 00 00 00 00 00 00 00 00 00
### (69) low = 0 (0x0000) high = 42053 (0xa445)
6e 53 44 86 ae 14 fc d4 64 f7 09 68 48
30 05 00 00 00 00 00 00 00 00 00 00 00
### (70) low = 0 (0x0000) high = 42056 (0xa448)
6e 55 44 86 ae 14 fc d4 64 f7 09 68 48
30 03 00 00 00 00 00 00 00 00 00 00 00
### (71) low = 0 (0x0000) high = 42059 (0xa44b)
6e 57 44 86 ae 14 fc d4 64 f7 09 68 48
30 01 00 00 00 00 00 00 00 00 00 00 00
### (72) low = 0 (0x0000) high = 42062 (0xa44e)
6e 59 44 86 ae 14 fc d4 64 f7 09 68 48
2f ff 00 00 00 00 00 00 00 00 00 00 00
### (73) low = 0 (0x0000) high = 42065 (0xa451)
6e 5b 44 86 ae 14 fc d4 64 f7 09 68 48
2f fd 00 00 00 00 00 00 00 00 00 00 00
### (74) low = 0 (0x0000) high = 42068 (0xa454)
6e 5d 44 86 ae 14 fc d4 64 f7 09 68 48
2f fb 00 00 00 00 00 00 00 00 00 00 00
### (75) low = 0 (0x0000) high = 42071 (0xa457)
6e 5f 44 86 ae 14 fc d4 64 f7 09 68 48
2f f9 00 00 00 00 00 00 00 00 00 00 00
### (76) low = 0 (0x0000) high = 42074 (0xa45a)
6e 61 44 86 ae 14 fc d4 64 f7 09 68 48
2f f7 00 00 00 00 00 00 00 00 00 00 00
A.3 Source list

A.3.1 shell script

#!/bin/sh

TXT="Test message"
CNT=0
ORGL=12345
ORGH=54321
BASE=0
WIDTH='expr ${ORGH} - ${ORGL}''
RANGE='expr 65539 / 2'
MAXHIGH=65535
INC=1

echo "### message = \"${TXT}\""
printf " low  = %d (0x%04x)\n" ${ORGL} ${ORGL}
printf " high  = %d (0x%04x)\n" ${ORGH} ${ORGH}
printf " range  = %d (0x%04x)\n" ${WIDTH} ${WIDTH}

echo -n "$\{TXT\}" >orgtxt
adaptive_encode 0 $\{ORGL\} $\{ORGH\} <orgtxt 2>/dev/null >orgdat
../tool/hexdsp <orgdat >hexdat

printf "### low  = %d (0x%04x)\n" $\{BASE\} $\{BASE\}
printf " range  = %d (0x%04x)\n" $\{RANGE\} $\{RANGE\}
printf " increment by %d (0x%04x)\n" $\{INC\} $\{INC\}
printf " while high <= %d (0x%04x)\n" $\{MAXHIGH\} $\{MAXHIGH\}

LOW=$\{BASE\}
HIGH='expr $\{BASE\} + $\{RANGE\}'

while [ $\{HIGH\} -le $\{MAXHIGH\} ]
do
adaptive_encode 0 $\{LOW\} $\{HIGH\} <orgtxt 2>/dev/null | ../tool/hexdsp >temp
NEWLOW='../tool/hexdiff hexdat temp 2>dat1'
if [ $? = 1 ]
then
  CNT='expr ${CNT} + 1'
  printf "### (%2d) " ${CNT}
  printf "  low = %d (0x%04x) " ${LOW} ${LOW}
  printf "  high = %d (0x%04x)\n" ${HIGH} ${HIGH}
  cat temp
  cat dat1
  NEWHIGH='expr ${NEWLOW} + ${HIGH}''
  adaptive_decode 0 \\
  ${NEWLOW} ${NEWHIGH} 2>/dev/null <orgdat >cmptxt 2>/dev/null
  diff orgtxt cmptxt
fi
RANGE='expr ${RANGE} + ${INC}''
HIGH='expr ${BASE} + ${RANGE}''
done

A.3.2 hexdsp.c

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

/*
 * Reverse the bits in the output.
 */
int reverse(int val)
{
  int cnt;
  int ret;

  for (cnt = 0, ret = 0; cnt < 8; cnt++) {
    ret <<= 1;
    ret |= val & 1;
val >>= 1;
}

return ret;

int main()
{
int ch;
int val;

while ((ch = getchar()) != EOF) {
    val = reverse(ch); /* reverse bits. */
    printf("%02x ", val & 0x0ff); /* print a byte in hex. */
}
putchar(\n);

return 0;

}

A.3.3 hexdiff.c

#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <memory.h>

/*
 * Input two files, each of which contains a number in hex format
 * and calculate the difference of the two numbers.
 * Print the difference to stderr
 * and print the MSB 16 bits of the difference to stdout.
 * The exit code is 1 when 17th and the following bits are 0.
 * Otherwise, the exit code is 0.
 */
int main(int argc, char* argv[]) {
    FILE* fp1; /* file 1 */
    FILE* fp2; /* file 2 */
    int ch1[4096];
    int ch2[4096];
    int val[4096];
    int cnt1, cnt2;
    int flg1, flg2;
    int ov;
    int ret;

    if (argc != 3) {
        fprintf(stderr, "Usage: hexdiff file1 file2\n");
        return 1;
    }

    /* open files. */
    fp1 = fopen(argv[1], "r");
    fp2 = fopen(argv[2], "r");
    if (fp1 == NULL || fp2 == NULL) {
        fprintf(stderr, "cannot open file\n");
        return 1;
    }

    /* read two numbers in the files. */
    memset(ch1, 0, sizeof(ch1));
    memset(ch2, 0, sizeof(ch2));
    flg1 = flg2 = 1;
    cnt1 = cnt2 = 0;
    while (fscanf(fp1, "%x", &ch1[cnt1]) == 1) {
        ch1[cnt1] &= 0x0ff; /* must be 2 digits */
        cnt1++;
    }

    while (fscanf(fp2, "%x", &ch2[cnt2]) == 1) {
        ch2[cnt2] &= 0x0ff; /* must be 2 digits */
A.3. Source list

```c
    cnt2++;

} /* close files. */
    fclose(fp1);
    fclose(fp2);

/* adjust the precision (length) of the numbers. */
    if (cnt1 < cnt2)
        cnt1 = cnt2;

/* calculate the difference of 2 numbers. */
    cnt2 = cnt1;
    ov = 0;
    while (cnt1 > 0) {
        cnt1--;
        val[cnt1] = ch1[cnt1] - ch2[cnt1] - ov;
        if (val[cnt1] < 0) {
            ov = 1;
            val[cnt1] += 0x100;
        } else
            ov = 0;
    }

/* print the difference. */
    ret = 1;
    for (cnt1 = 0; cnt1 < cnt2; cnt1++) {
        fprintf(stderr, "02x ", val[cnt1]);
        if (cnt1 >= 2 && val[cnt1] != 0)
            ret = 0;
    }
    fputc(\n, stderr);

/* print MSB 16 bits of the difference. */
    printf("%d
", val[0] * 256 + val[1]);

return ret;
}```
Appendix B

Calculation of bits/symbol

The average bits/symbol for each $n$ symbols can be used to detect halving points (4.1.1). It would be easier to detect halving points by smoothing the change in the average by calculating the average of consecutive $j$ values. The example is shown as follows.

The calculation of the average is

$$V_i = \frac{v_i + v_{i+1} + \ldots + v_{i+j}}{j}$$ (B.1)

where $v_i$ is the bits/symbol of the $i$th block. The following graph shows the bits/symbol using the above calculation with $j = 3, 5, 7$ and 9.

![Graph showing smoothing of bits/symbol values](image)

Figure B.1: Smoothing of the bits/symbol values
Appendix C

Finding lower shrinking factors of the combined scheme

C.1 The experiment

We conducted an attack on the LFB combined scheme (4.2). All the frequencies of the initial model is set to 1. Whether or not the model is randomly chosen is not important in this attack because the attack does not try to discover the initial model and so the initial model does not have any influence on the attack. The attack starts in a part of the output where the symbols at the bottom of the frequency table are encoded.

The parameters used in the experiment are as follows.

- initial low value = 0x1234
- initial high value = 0xDBA9
- lower shrink factor 0 = 349
- upper shrink factor 0 = 9124
- lower shrink factor 1 = 541
- upper shrink factor 1 = 9978
- 128 bit random sequence = 0x714556410728AD711B16755967F8EE7C

After sampling the output of the encoder, a program which tries to find the lower shrinking factors, analyzed the output. The result of the experiment is shown in the next section.
C.2 The result of the experiment

Bit 0 (00100011011101110000100011110010)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 1 (01000110111011100001000111100100)
  sh high 9000
  sh low (541) bits : org (0000110111011100) tst (0100011011101111)
  mid-range buf : before (1) after (0)
  see Bit 16

Bit 2 (00011011101110000100011110010001)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 3 (0001101110111000010001111100010001)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 4 (00110111011100001000111100110111)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 5 (01101110111000010001111100010001)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 8 (01110111000010001111001000110111)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 12 (011100001000111100100011110111)
  sh high 9000
C.2. The result of the experiment

sh high 9500
sh high 9999

Bit 16 (00001000111100100110111110000)
  sh high 9000
       sh low(349) bits: org(00001000111100100110111110000) tst(00001000111100100110111110000)
mid-range buf: before(0) after(1)
  see Bit 29
  sh high 9500
       sh low(349) bits: org(00001000111100100110111110000) tst(00001000111100100110111110000)
mid-range buf: before(0) after(1)
  see Bit 29
  sh high 9999
       sh low(349) bits: org(00001000111100100110111110000) tst(00001000111100100110111110000)
mid-range buf: before(0) after(1)
  see Bit 29

Bit 17 (0001000011110010011011111000000)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 18 (001000111100100011011110111000000)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 19 (01000111100011011110111000011011)
  sh high 9000
       sh low(788) bits: org(000001111001100011011100011011) tst(010001111001100011011100011011)
mid-range buf: before(1) after(1)
  see Bit 33
  sh high 9500
       sh low(788) bits: org(000001111001100011011100011011) tst(010001111001100011011100011011)
mid-range buf: before(1) after(1)
  see Bit 33
  sh high 9999
       sh low(788) bits: org(000001111001100011011100011011) tst(010001111001100011011100011011)
mid-range buf: before(1) after(1)
  see Bit 33

Bit 21 (00011110010001101110111000001101)
  sh high 9000
C.2. The result of the experiment

sh high 9500
sh high 9999

Bit 22 (00111100100011011101110000110110)
sh high 9000
sh high 9500
sh high 9999

Bit 23 (01111001000110111011100001101100)
sh high 9000

  sh low(923) bits : org(0001000110111011) tst(011110001000110111)
  mid-range buf : before(4) after(1)
  see Bit 40

  sh low(692) bits : org(0001000110111011) tst(011110001000110111)
  mid-range buf : before(4) after(1)
  see Bit 40

sh high 9500

  sh low(923) bits : org(0001000110111011) tst(011110001000110111)
  mid-range buf : before(4) after(1)
  see Bit 40

  sh low(692) bits : org(0001000110111011) tst(011110001000110111)
  mid-range buf : before(4) after(1)
  see Bit 40

sh high 9999

  sh low(923) bits : org(0001000110111011) tst(011110001000110111)
  mid-range buf : before(4) after(1)
  see Bit 40

  sh low(692) bits : org(0001000110111011) tst(011110001000110111)
  mid-range buf : before(4) after(1)
  see Bit 40

Bit 28 (00100011011101110000110111011100)
sh high 9000
sh high 9500
sh high 9999

Bit 29 (01000110111011100001101110111000)
sh high 9000

  sh low(541) bits : org(0001101110111011) tst(01000110111011)
  mid-range buf : before(1) after(0)
  see Bit 44

sh high 9500
C.2. The result of the experiment

sh low(541) bits: org(00001101110111011100) tst(0100011011101111)
mid-range buf: before(1) after(0)
  see Bit 44

sh high 9999

sh low(541) bits: org(00001101110111011100) tst(0100011011101111)
mid-range buf: before(1) after(0)
  see Bit 44

Bit 31 (00011011110111011100110111100001)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 32 (0011011101110000110111111000010)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 33 (01101111011100001101111110000100)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 36 (01110111011100001101110000110111)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 40 (01110000110111011100001101110111)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 44 (0000110111011101110001101111000000)
  sh high 9000
    sh low(541) bits: org(00001101110111011100) tst(0000110111101111)
    mid-range buf: before(0) after(0)
      see Bit 58
  sh high 9500
    sh low(541) bits: org(00001101110111011100) tst(0000110111101111)
    mid-range buf: before(0) after(0)
      see Bit 58
  sh high 9999
    sh low(541) bits: org(00001101110111011100) tst(0000110111101111)
C.2. The result of the experiment

mid-range buf: before(0) after(0)

see Bit 58

Bit 45 (00011011101110000110111011100000)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 46 (00110111011100001101110111000000)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 47 (01101110111000011011101110000000)
  sh high 9000
  sh high 9500
  sh high 9999

Bit 50 (01110111000011011101110000100011)
  sh high 9000
  sh high 9500
  sh high 9999

Firstly, at Bit 1, one shrinking factor was detected. The next output should start at Bit 16, as the comment in the program output indicates. The next shrinking factor was actually detected there. At Bit 19, another shrinking factor was detected. However, the next output should have started at Bit 33 but no shrinking factor was detected at that point. Hence, this was a wrong detection.

As can be seen, two lower shrinking factors were correctly detected by the program. The program did not find the upper shrinking factors and 3 different upper shrinking factors were used. However, the influence of the upper shrinking factors are small and so it is possible to correctly obtain the lower shrinking factors.
Appendix D

Source code for the LFB combined scheme

D.1  Implementation of LFB combined scheme

We implemented the LFB combined scheme based on WNC implementation. The program was not developed for the practical use but only for the experiment. Since the program was to be used to examine the resistance against BH attack, the features such as 16 bit substitution and randomly chosen initial model which are irrelevant for the experiment, are omitted.

The key is given by a file in text format. They are as follows.

- shrinking factor 0 (low)
- shrinking factor 0 (high)
- shrinking factor 1 (low)
- shrinking factor 1 (high)
- random 128 bits

The list only includes the new codes and the source files which were modified from the original WNC implementation.

D.2  Source list

D.2.1  encode.c

/* MAIN PROGRAM FOR ENCODING. */

#include <stdio.h>
#include <stdlib.h>
#include "model.h"
#include "arithmetic_coding.h"
#include "secret.h"

int main(int argc, char* argv[]) {
    if (argc != 3) {
        fprintf(stderr, "Usage: adaptive_encode low high \n");
        return 1;
    }
    initsecret();
    start_model(); /* Set up other modules. */
    start_outputing_bits();
    start_encoding(atol(argv[1]), atol(argv[2]));
    for (;;) { /* Loop through characters. */
        int ch; int symbol;
        ch = getc(stdin); /* Read the next character. */
        if (ch==EOF) break; /* Exit loop on end-of-file. */
        symbol = char_to_index(ch); /* Translate to an index. */
        encode_symbol(symbol, cum_freq); /* Encode that symbol. */
        update_model(symbol); /* Update the model. */
    }
    encode_symbol(EOF_symbol, cum_freq); /* Encode the EOF symbol. */
    done_encoding(); /* Send the last few bits. */
    done_outputing_bits();
    exit(0);
}

D.2.2 decode.c

/* MAIN PROGRAM FOR DECODING. */

#include <stdio.h>
#include "model.h"
#include "arithmetic_coding.h"
#include "secret.h"

int
main(int argc, char* argv[])
{
    if (argc != 3) {
        fprintf(stderr, "Usage: adaptive_decode low high \n");
        return 1;
    }

    initsecret();
    start_model(); /* Set up other modules. */
    start_inputing_bits();
    start_decoding(atol(argv[1]), atol(argv[2]));
    for (;;) { /* Loop through characters. */
        int ch; int symbol;
        symbol = decode_symbol(cum_freq); /* Decode next symbol. */
        if (symbol==EOF_symbol) break; /* Exit loop if EOF symbol. */
        ch = index_to_char[symbol]; /* Translate to a character. */
        putc(ch,stdout); /* Write that character. */
        update_model(symbol); /* Update the model. */
    }
    exit(0);
}

D.2.3 arithmetic_encode.c

/* ARITHMETIC ENCODING ALGORITHM. */

#include <stdio.h>

#include "arithmetic_coding.h"
#include "parameter.h"
#include "secret.h"

static void bit_plus_follow(); /* Routine that follows */
/* CURRENT STATE OF THE ENCODING. */

static code_value low, high; /* Ends of the current code region */
static long bits_to_follow; /* Number of opposite bits to output after */
/* the next bit. */

/* START ENCODING A STREAM OF SYMBOLS. */

start_encoding(code_value l, code_value h)
{
    low = l; /* Full code range. */
    high = h;
    bits_to_follow = 0; /* No bits to follow next. */
}

/* ENCODE A SYMBOL. */

encode_symbol(symbol, cum_freq)
{
    int symbol; /* Symbol to encode */
    int cum_freq[]; /* Cumulative symbol frequencies */
    long range; /* Size of the current code region */
    range = (long)(high-low)+1;
    high = low + /* Narrow the code region */
        (range*cum_freq[symbol-1])/cum_freq[0]-1; /* to that allotted to this */
    low = low + /* symbol. */
        (range*cum_freq[symbol])/cum_freq[0];
    for (;;) { /* Loop to output bits. */
        if (high<Half) {
            bit_plus_follow(0); /* Output 0 if in low half. */
        } else if (low>=Half) { /* Output 1 if in high half. */
            bit_plus_follow(1);
            low -= Half;
            high -= Half; /* Subtract offset to top. */
        } else if (low>=First_qtr /* Output an opposite bit */
                && high<Third_qtr) { /* later if in middle half. */
bits_to_follow += 1;
low -= First_qtr; /* Subtract offset to middle*/
high -= First_qtr;
}
else break; /* Otherwise exit loop. */
low = 2*low;
high = 2*high+1; /* Scale up code range. */
}
range = (long)(high-low)+1;
high = low + range * SHRINK_VALH / 10000;
low = low + range * SHRINK_VALL / 10000;
nextsecret();

/* FINISH ENCODING THE STREAM. */

done_encoding()
{
    bits_to_follow += 1; /* Output two bits that */
    if (low<First_qtr) bit_plus_follow(0); /* select the quarter that */
    else bit_plus_follow(1); /* the current code range */
} /* contains. */

/* OUTPUT BITS PLUS FOLLOWING OPPOSITE BITS. */

static void bit_plus_follow(bit)
    int bit;
{
    output_bit(bit); /* Output the bit. */
    while (bits_to_follow>0) {
        output_bit(!bit); /* Output bits_to_follow */
        bits_to_follow -= 1; /* opposite bits. Set */
    } /* bits_to_follow to zero. */
}

D.2.4 arithmetic_decode.c
/* ARITHMETIC DECODING ALGORITHM. */

#include <stdio.h>

#include "arithmetic_coding.h"
#include "parameter.h"
#include "secret.h"

/* CURRENT STATE OF THE DECODING. */

static code_value value; /* Currently-seen code value */
static code_value low, high; /* Ends of current code region */

/* START DECODING A STREAM OF SYMBOLS. */

start_decoding(code_value l, code_value h)
{
    int i;
    value = 0; /* Input bits to fill the */
    for (i = 1; i<=Code_value_bits; i++) { /* code value. */
        value = 2*value+input_bit();
    }
    low = l;
    high = h;
}

/* DECODE THE NEXT SYMBOL. */

int decode_symbol(cum_freq)
{
    int cum_freq[]; /* Cumulative symbol frequencies */
    { long range; /* Size of current code region */
        int cum; /* Cumulative frequency calculated */
        int symbol; /* Symbol decoded */
        range = (long)(high-low)+1;
        cum = /* Find cum freq for value. */
        (((long)(value-low)+1)*cum_freq[0]-1)/range;
D.2. Source list

```c
for (symbol = 1; cum_freq[symbol] > cum; symbol++) ; /* Then find symbol. */
high = low + /* Narrow the code region */
    (range*cum_freq[symbol-1])/cum_freq[0]-1; /* to that allotted to this */
low = low + /* symbol. */
    (range*cum_freq[symbol])/cum_freq[0];
for (;;) { /* Loop to get rid of bits. */
    if (high<Half) {
        /* nothing */ /* Expand low half. */
    }
    else if (low>=Half) { /* Expand high half. */
        value -= Half;
        low -= Half; /* Subtract offset to top. */
        high -= Half;
    }
    else if (low>=First_qtr && high<Third_qtr) {
        value -= First_qtr;
        low -= First_qtr; /* Subtract offset to middle*/
        high -= First_qtr;
    }
    else break; /* Otherwise exit loop. */
    low = 2*low;
    high = 2*high+1; /* Scale up code range. */
    value = 2*value+input_bit0; /* Move in next input bit. */
}
range = (long)(high-low)+1;
high = low + range * SHRINK_VALH / 10000;
low = low + range * SHRINK_VALL / 10000;
nextsecret();

return symbol;
}
```

D.2.5 secret.h

/*
* parm list
* 1 initial low value
* 2 initial high value
* 3 shrinking factor 0 (low)
* 4 shrinking factor 0 (high)
* 5 shrinking factor 1 (low)
* 6 shrinking factor 1 (high)
* 7 random 128 bits
*/

#define SHRINK_LOW 0
#define SHRINK_HIGH 1

#define SHRINK_VALL \  
(g_shrink[(g_randbit[g_posrandbyte] >> g_posrandbit) & 1][SHRINK_LOW])
#define SHRINK_VALH \  
(g_shrink[(g_randbit[g_posrandbyte] >> g_posrandbit) & 1][SHRINK_HIGH])

#ifndef SECRET
long g_shrink[2][2];
unsigned char g_randbit[16];
int g_posrandbit;
int g_posrandbyte;
#else
extern long g_shrink[2][2];
extern unsigned char g_randbit[16];
extern int g_posrandbit;
extern int g_posrandbyte;
extern void initsecret();
extern void nextsecret();
extern void printsecret(FILE* fp);
#endif

D.2.6 secret.c

#include <stdio.h>
#include <stdlib.h>

#define SECRET
#include "secret.h"
#define SECRET

void initsecret()
{
    FILE* fp;
    int cnt;
    char val[3];
    char buf[256];

    fp = fopen("secret", "r");
    if (fp == NULL) {
        perror("cannot open secret file");
        exit(1);
    }
*/ shrink-0 low */
    fgets(buf, sizeof(buf), fp);
    g_shrink[0][SHRINK_LOW] = atol(buf);
*/ shrink-0 high */
    fgets(buf, sizeof(buf), fp);
    g_shrink[0][SHRINK_HIGH] = atol(buf);
*/ shrink-1 low */
    fgets(buf, sizeof(buf), fp);
    g_shrink[1][SHRINK_LOW] = atol(buf);
*/ shrink-1 high */
    fgets(buf, sizeof(buf), fp);
    g_shrink[1][SHRINK_HIGH] = atol(buf);
*/ random 128 bit sequence */
    fgets(buf, sizeof(buf), fp);
    val[2] = \0;
    for (cnt = 0; cnt < 16; cnt++) {
        val[0] = buf[cnt * 2];
        val[1] = buf[cnt * 2 + 1];
        g_randbit[cnt] = strtol(val, NULL, 16);
    }
}
g_posrandbit = 7;
g_posrandbyte = 0;

fclose(fp);

return;
}

void nextsecret()
{

g_posrandbit--;
if (g_posrandbit < 0) {
    g_posrandbit = 7;
    g_posrandbyte++;
    if (g_posrandbyte == 16)
        g_posrandbyte = 0;
}
}

void printsecret(FILE* fp)
{
    int cnt;

    fprintf(fp, "s0-l:%%ld\n", g_shrink[0][SHRINK_LOW]);
    fprintf(fp, "s0-h:%%ld\n", g_shrink[0][SHRINK_HIGH]);
    fprintf(fp, "s1-l:%%ld\n", g_shrink[1][SHRINK_LOW]);
    fprintf(fp, "s1-h:%%ld\n", g_shrink[1][SHRINK_HIGH]);
    fprintf(fp, "rand:"
);
    for (cnt = 0; cnt < sizeof(g_randbit); cnt++)
        fprintf(fp, "%%02x ", g_randbit[cnt] & 0xff)
    fprintf(fp, "\n");