

A NEW ENCODING DECODING SCHEME FOR TEXT COMPRESSION WITH EMBEDDED SECURITY

Ebru Celikel and Mehmet E. Dalkilic

International Computer Institute, Ege University, 35100 Bornova/Izmir, Turkey
{celikel, dalkilic}@ube.ege.edu.tr

Abstract- In today's communication world, finding out and improving the means of better channel utilization, as well as providing the security of data transmitted are the key issues. For some data types, e.g. text, full (lossless) data recovery might also be required. Therefore, a *lossless compression algorithm* that is reasonably fast and secure enough would meet the needs of many in the networking environments.

While standard compression tools (e.g. *gzip* of Unix and *pkzip* of Windows platforms) achieve compression rates at around 3 *bpc* (*bits per character*) for text type of data, they are poor at security. This study presents the design and implementation of a new *encoding/decoding* scheme to offer compression and security at the same time. For that, the scheme combines compression with encryption. To address the security issue, *encoding* is made through an *Encoding matrix* generated by a *Pseudo Random Number Generator*. This encoding already provides an initial compression. It then uses *Arithmetic Coding* to further compress the text. For further security, an *iterative encoding* scheme is proposed and implemented. The results obtained are encouraging in terms of both compression and security.

Key Words- Lossless compression, encoding/decoding algorithm, security.

1. INTRODUCTION

Technology today provides many means for data storage and transmission. People are still working to improve them. With so many ways to communicate, the amount of data transmitted is increasing drastically, no matter what the geographical distances are. That ease of transmission facilitates communication. But it may only seemingly increase the amount of data transferred, because lack of security frustrates people who are in need of secure data transfer. Whenever the channel capacity is limited and the data is so important that it should be recovered exactly as it was, a *lossless compression* algorithm is needed. This is almost always the case for text data.

The conventional *lossless compression algorithms* provide varying performance levels in terms of compression rate and runtime by exploiting redundancy in data [1]. One of the most common lossless compression algorithms is the *Arithmetic Coding* [2]. It represents symbol sequences as floating point numbers and is used in combination with many other compression algorithms.

The standard compression tools are mainly intended for providing compression, not security. The new *encoding/decoding* scheme we present intends to fill that gap: While compressing, it provides a reasonable level of security for the text being compressed.

The algorithm we have developed is mainly concerned with text type of data, but it can well be extended to other data types.

2. THEORY

The scheme consists of a *sender* side which does the *encoding* and a *receiver* side which performs *decoding*, as seen on Fig. 1. In the following section, operations of the *encoder* and *decoder* are explained in detail.

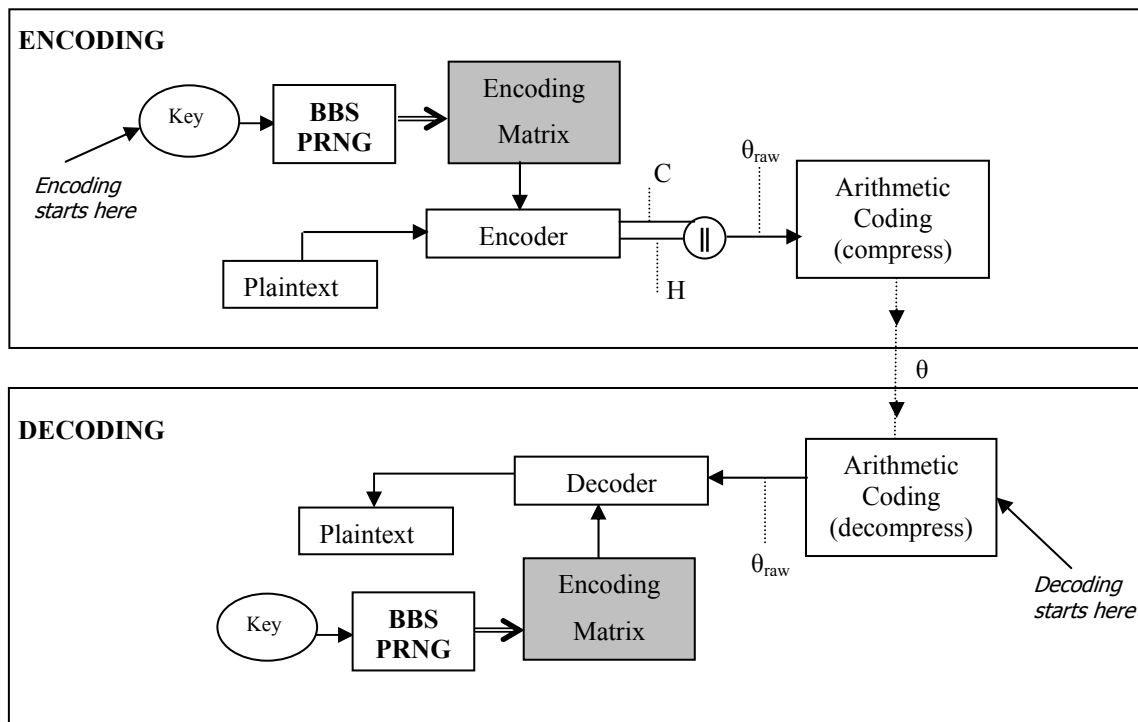


Fig. 1 A new *encoding/decoding* scheme for text compression with embedded security.

2.1. Encoder

Given a *plaintext* of length $|P|$, the *encoder* first divides it into n parts, with length $|P|/n$ each, where n is the *text division index*. *Encoding* is performed step by step. At the i^{th} *encoding* step, an n -gram is constructed by concatenating the i^{th} symbols of each part. Both *encoding* and *decoding* use an n -dimensional *Encoding matrix* (E). This *matrix* is generated by a *BBS Pseudo Random Number Generator* with a shared *seed* between the two sides. The *seed* serves as the *key* to the system. Since *encoder* and *decoder* employ the same *key*, the system is *symmetrical* [3]. At the end of *encoding*, a *ciphertext* (C) of length $|P|/n$ is constructed. To let *decoder* uniquely decode that *ciphertext*, a special *bit stream* H (called the *help string*) is generated. The *help string* carries the information regarding to which n -gram each *cipher symbol* corresponds to. *Ciphertext* together with the *help string* comprises the *raw output* ($\theta_{\text{raw}}=C||H$), which is certainly longer than $|P|/n$

due to the *help string* cost. Finally, the *Arithmetic Coding* is applied to the *raw output* (θ_{raw}) to obtain compressed, hence shorter *final output* (θ).

In a 2-dimensional *Encoding matrix* (E), an n -gram ‘ xy ’ is encoded into a *cipher symbol* c as $E[x,y]=c$, where $x,y \in A$ and $c \in S$ (A is the *alphabet* and S is the *cipher symbol space*). The dimension of the *Encoding matrix* is the same as the *text division index*, n . This *matrix* is organized randomly to dissipate language statistics. The key to security in our scheme is to make it hard to determine to which n -gram each *cipher symbol* corresponds to. The most practical way of doing this is to hide the *Encoding matrix* itself. For extra security, the *matrix* is constructed out of a random bit sequence generated by *Blum Blum Shub (BBS) Pseudo Random Number Generator (PRNG)* [4]. The random bit sequence is generated via large primes in the *Groups, Algorithms and Programs (GAP)* [5] environment in *Unix*. For that, two large primes p and q (with 95-digits each) are created out of *Unix* system information. While $n=pxq$, a *seed* s is picked up where $gcd(n,s)=1$. The *BBS* algorithm below, then generates M random bits $\{B_1, B_2, \dots, B_M\}$.

procedure BBS (s, n, M)

```

X0 = s2 mod n
for i = 1 to M
    Xi = (Xi-1)2 mod n
    Bi = Xi mod 2

```

Table 1 is the *Encoding matrix* (E) generated out of 95-digit large primes p and q .

Table 1. *BBS* generated *Encoding matrix* (‘~’ denotes the space character).

	~	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
~	C	I	D	K	H	J	G	E	Z	V	R	Q	N	~	O	U	F	P	S	M	T	W	B	L	A	Y	X
a	Y	C	I	L	O	G	F	V	J	Q	W	S	R	K	T	~	D	M	Z	X	E	A	B	U	H	P	N
b	K	Y	P	~	E	M	L	X	U	J	B	D	V	S	G	Z	H	I	A	C	F	O	N	R	W	T	Q
c	X	Z	B	F	U	O	I	L	M	W	A	V	Y	D	~	E	C	H	K	S	T	R	G	J	P	N	Q
d	Q	Z	F	I	H	Y	O	M	T	N	P	L	R	X	D	A	E	S	G	C	U	K	J	W	V	B	~
e	E	F	D	N	J	M	W	I	V	S	P	B	Y	A	R	T	L	K	H	Q	~	O	C	G	X	U	Z
f	P	J	V	K	H	R	N	Z	F	W	S	U	X	M	A	D	L	G	T	B	~	O	Y	Q	C	E	I
g	K	X	U	D	Z	I	Q	L	O	E	S	J	Y	T	H	~	A	C	M	N	V	F	W	G	P	B	R
h	G	V	M	F	Z	Q	I	W	~	J	B	P	C	U	Y	D	L	N	K	H	R	T	A	X	S	O	E
i	I	F	~	B	Z	V	D	K	R	L	E	P	H	T	J	G	M	A	O	U	C	W	N	Y	Q	X	S
j	G	B	U	W	K	O	X	S	R	M	N	F	Q	V	Y	T	H	~	Z	C	A	E	P	J	I	D	L
k	J	Q	K	C	B	I	M	O	R	Y	E	D	V	W	X	S	H	T	U	G	A	L	F	P	~	Z	N
l	C	Z	Y	F	H	O	N	W	U	~	S	J	E	V	M	A	D	I	P	G	X	T	K	Q	L	B	R
m	P	O	H	R	Y	K	Z	M	I	B	F	D	X	W	J	L	V	~	Q	E	N	U	S	G	A	C	T
n	~	S	V	E	M	L	U	W	T	Z	A	J	K	P	D	C	G	X	R	H	B	Y	O	F	N	I	Q
o	M	O	I	K	D	R	W	Y	U	V	T	A	L	~	Z	P	F	S	N	G	C	H	E	B	Q	J	X
p	E	Y	R	J	~	A	L	D	B	C	G	T	U	F	Q	N	W	O	P	X	M	I	K	S	Z	V	H
q	B	H	Q	S	F	P	K	R	N	G	A	D	~	V	Y	Z	U	I	L	W	O	M	X	E	J	C	T
r	Q	L	G	H	O	P	T	K	A	Z	V	B	D	S	C	J	U	I	F	Y	N	R	X	M	~	E	W
s	I	U	K	D	B	J	V	M	Y	A	E	R	~	S	X	C	G	W	L	N	P	F	Q	O	T	H	Z
t	Q	O	B	J	I	P	S	L	Z	D	X	C	N	M	T	~	G	V	H	A	F	R	W	E	K	Y	U
u	C	A	I	Z	X	H	M	N	W	V	O	B	F	K	P	~	Y	J	S	Q	R	U	L	G	D	T	E
v	R	~	P	C	X	J	H	N	V	F	Q	W	Y	B	T	Z	G	I	U	K	A	D	M	L	O	E	S
w	Q	U	I	~	X	O	F	P	K	N	G	M	W	Z	C	A	H	L	T	Y	B	V	R	E	S	J	D
x	A	T	I	K	L	~	Y	O	X	M	J	N	S	W	Q	H	R	F	P	Z	G	D	E	C	V	U	B
y	I	C	F	T	X	W	M	O	S	R	P	B	Q	U	G	A	K	N	E	D	H	V	~	J	L	Z	Y
z	P	D	J	Y	R	B	M	N	T	S	U	V	O	I	C	K	A	W	H	E	Z	X	F	G	Q	L	~

```

procedure Encode (P,E,n)
repeat
  i=1;  read(nGrami);           // nGrami=P[(2*i-1), (2*i)]
  Ci=E[nGrami[1],nGrami[2], ...,nGrami[n]];  // Issue Ci
  ListIndex=0;  FlagFound=0;  Context='';
  repeat
    if (nGrami=SuccessorsListContext[ListIndex] then FlagFound=1;
    else ListIndex++;
  until (FlagFound=1 or EndOfList)
  // D=Number of elements in the SuccessorListContext
  if (FlagFound=1) then Hi=EncodeWithLogDBits(ListIndex);
  else Hesc=000...0;           // Issue Hesc (escape seq.) of log2[D] bits
    HoldIndex=1;               // HoldIndex = number of Cis in E
    for j=1 to |A|n-1         // rows in E
      for k=1 to |A|           // columns in E
        if E[j,k]=Ci then HoldIndex++; break;
    // R=HoldIndex - number of Ci encoded n-grams following Context
    Hi=EncodeWithLogRBits(HoldIndex); // Issue Hi
    AppendToList(nGrami, SuccessorsListContext);
  Context=nGrami;  i++;
until EOF

```

While encoding, the i^{th} n -gram (n -gram _{i}) constructs the index for a table lookup in *Encoding matrix*, resulting in the i^{th} ciphertext (C_i). The *encoder* provides an initial compression ratio of $100 \times (n-1)/n$ percent (Eq. 1):

$$100 \times \left(1 - \frac{(|P|/n)}{|P|} \right) = 100 \times \left(\frac{n-1}{n} \right) \% \quad (1)$$

Let's encode P ='spacecraft~to~mars' with $|P|=18$ and $n=2$: First we divide P into 2 parts and obtain $18/2=9$ digrams (Table 2). Then, we determine *cipher symbols* corresponding to each *digram* with simple table lookup through the *Encoding matrix* shown in Table 1. The generated *ciphertext* is C ='GLNL~TMOY' (*cipher symbols* are denoted by capital letters).

Table 2. Encoding with the novel scheme on a sample English text.

	1	2	3	4	5	6	7	8	9
P₁	s	a	e	r	f	~	o	m	r
P₂	p	c	c	a	t	t	~	a	s
C	G	L	N	L	~	T	M	O	Y
H	10100	01	000110	11	010	1	100	1000	101

For *receiver* to uniquely decode the *plaintext*, a varying length special bitstream called *help string* (H) is generated for each n -gram encoded. *Help string encoding* is carried

out in a dynamic fashion to minimize its length. While *encoding* n -gram $_i$, the corresponding *help string* bits (H_i) are constructed based on a length- m *context*. Since no *context* exists for the very first n -gram (i.e., n -gram $_1$), *help string* for it (H_1) needs to be encoded differently: It is encoded as n -gram $_1$'s *row index* in the *Encoding matrix*. For the ongoing example, H_1 for digram $_1$ ='sp' is encoded as '10100', since 's' occurs in the 20th row of the *Encoding matrix*. The reason why H_1 is encoded with 5 bits is that, each *cipher symbol* occurs 27 times within the *Encoding matrix* in Table 1. So, minimum of $\lceil \log_2(27) \rceil = 5$ bits are needed to uniquely identify it.

For generating H_i ($i > 1$), a length- m *context* is used. If $m = n$, then *context* becomes n -gram $_{i-1}$. n -gram $_i$ that is mapped into C_i is first searched within the n -gram $_{i-1}$'s *successors-list*, which is the list of n -grams that follow the *context* and are mapped into the same symbol as C_i . This list is constructed by using the q -gram statistics of the source language, where $q = m + n$. If $m = n = 2$ then, we need to use *fourgram* statistics. For example, the full English *fourgram* list contains $27^4 = 531.441$ entries. Since not each *fourgram* occurs in English texts, we only keep a list of the most frequent (e.g. 24.688) *fourgrams* in our current implementation.

Following the *context* n -gram $_{i-1}$, the number of n -grams (denoted by d_i) that are mapped into the same *cipher symbol* as C_i determines H_i 's length. If n -gram $_i$ is found within the *context*'s *successors-list*, then its index in that list is encoded as H_i with $\lceil \log_2(d_i) \rceil$ bits. In the example, to encode digram $_2$ ='ac' (with C_2 ='L'), we search 'ac' within its *context*'s ('sp') *successors-list*. There are two *digrams* ('ac' and 'ra') that are mapped into C_2 , i.e. 'L'. The encoding consisting of all zeros for each *help string* length is named as the *escape sequence* and is reserved for a special case when the searched n -gram is not found within the current *context*'s *successors-list*. So, we need to consider that special case as a member of each *successors-list*. Therefore, we have 3 cases that follow the current *context* ('sp') in our example, leading to need for $\lceil \log_2(3) \rceil = 2$ bits. Since 'ac' is the first *digram* in that list, then $H_2 = '01'$.

If the i^{th} n -gram is not found in the i^{th} *context*'s *successors-list*, first an *escape sequence* (H_{esc}), which is a special bit stream consisting of all-zeros, is issued using as many number of bits as $\lceil \log_2(d_i) \rceil$. Second, the *Encoding matrix* (E) *row index* of the not found n -gram is emitted as H_i with $\lceil \log_2(r_i) \rceil$ bits, where r_i is the number of times C_i occurs in E . In fact, we subtract the number of C_i -encoded n -grams occurring in the *context*'s *successors-list* from r_i . Finally, the not found n -gram is appended to the end of the *context*'s *successors-list*. Thus, next time we encode the *help string* for the same n -gram following the same *context*, no *escape sequence* would be needed, meaning fewer bits in the *help string* (H). This is why *help string* encoding is called *dynamic*. While encoding digram $_3$ ='ec' which maps to C_3 ='N' in the *Encoding matrix* of Table 1, since 'ec' does not exist within the *context* (digram $_2$)='ac's *successors-list*, we first need to issue an *escape sequence* as '0', because $d=2$ and $\lceil \log_2(2) \rceil = 1$. Then, we encode the non-existing *digram*'s *Encoding matrix row index* as '00110'. H_3 is the combination of *escape sequence* with the *index*, i.e. '000110'.

At the end of $|P|/n$ encoding steps, *ciphertext* (C) together with *help string* (H) forms the *raw output* (θ_{raw}). The *final output* (θ) is generated by compressing *raw output* (θ_{raw}) with *Arithmetic Coding*.

2.2. Decoder

The receiver first decompresses the *final output* (θ) sent by the *encoder* to obtain the *raw output* (θ_{raw}). If the *Encoding matrix* is at hand, *decoding* is performed step by step in the manner that *encoding* was done before. At each *decoding* step i , the *decoder* needs to determine to which n -gram the i^{th} *cipher symbol* (C_i) corresponds to. To recover the first n -gram, the first *cipher symbol* (C_1) is treated as a special case. The n -gram corresponding to C_1 is recovered using H_1 as a *row index* in E . For the following n -grams, *decoder* uses the previous n -gram (called as the *context*) with corresponding *help string* bits. Below, the *decoding* algorithm for the scheme developed is given:

```

procedure Decode (C,H,E,n)
repeat
   $i=1$ ;  $C_0=''$ ;  $\text{Context}=C_{i-1}$ ;
  read( $C_i$ );  $H_i=\text{read}(\text{LogDBitsFromH})$ ;
  if ( $H_i \neq 0\dots 0$ ) then  $n\text{Gram}_i=\text{SuccessorsList}_{\text{Context}}[H_i]$ ;
  else  $\text{HoldIndex}=1$ ; //  $\text{HoldIndex}$  = number of  $C_i$ s in  $E$ 
    for  $j=1$  to  $|A|^{n-1}$  // rows in  $E$ 
      for  $k=1$  to  $|A|$  // columns in  $E$ 
        if  $E[j,k]=C_i$  then  $\text{HoldIndex}++$ ; break;
    //  $R=\text{HoldIndex}$  - number of  $C_i$  encoded  $n$ -grams following  $\text{Context}$ 
     $H_i=\text{read}(\text{LogRBitsFromH})$ ; // Processing escape seq.
     $n\text{Gram}_i=E[(H_i + \text{number of } C_i \text{ encoded } n\text{-grams following } \text{Context}),k]$ ;
    //where  $k$  is the column  $C_i$  resides
    AppendToList( $n\text{Gram}_i$ ,  $\text{SuccessorsList}_{\text{Context}}$ );
   $i++$ ;
// Issue  $n\text{Gram}_i$  to construct  $P=[n\text{Gram}_1 || n\text{Gram}_2 || \dots || n\text{Gram}_{|P|/n}]$ 
until EOF

```

Let us decode the *ciphertext* $C='GLNL\sim TMOY'$ of the example where $H='10100010001101101011001000101'$. Suppose E is given and $n=2$. We take the first *cipher symbol* $C_1='G'$ and need to determine to which *digram* it corresponds to. Since the very first n -gram encoding is done specially, decoding for it should also be carried out accordingly. Knowing that each *cipher symbol* occurs 27 times throughout the *Encoding matrix*, we need to read the first $\lceil \log_2(27) \rceil = 5$ bits of the *help string* as H_1 . Hence, $H_1='10100'$. We also know that H_1 denotes the *row index* for the very first *digram*. Localizing the 20th row, we here search for the column where C_1 resides. This is the column corresponding to letter 'p'. Therefore, we decode C_1 as 'sp'.

For decoding the next *cipher symbol* $C_2='L'$, we now have the *context digram*₁='sp', at hand. We first check the current *context's successors-list* to determine the number of bits for H_2 . There are 3 members in the *list*: the *escape sequence*, 'ac' and 'ra'. Then, we would need $\lceil \log_2(3) \rceil = 2$ bits for H_2 . Therefore, $H_2='01'$. Since H_2 is not all-zeros (*escape sequence*) then it denotes the index of the *digram* to be decoded in the current *context's successors-list*. The 1st *digram* in that list is 'ac'. Then, *digram*₂ is decoded as 'ac'.

Decoding the 3rd *cipher symbol* $C_3='N'$ is performed likewise: This time *digram*₂='ac' is the *context* and its *successors-list* contains two members as the *escape sequence* and 't1', leading to need for $\lceil \log_2(2) \rceil = 1$ bit for H_3 . When we read the next 1 bit from the *help string*, which is '0', we see that it is an *escape sequence*. It means that the searched *digram* does not exist within the current *context's successors-list*. So, we need to read extra *help string* bits with as many number of bits as $\lceil \log_2(27-2) \rceil = 5$. H_{extra} for H_3 is then '00110'. It is the *row index* of the letter 'e' and when we search for $C_3='N'$ in that row, we find it under column 'c'. Hence, *digram*₃='ec'. When decoding is completed, the *plaintext* is recovered as 'spacecraft~to~mars'.

2.3. Mathematical Analysis of the Scheme

The scheme introduced provides an initial compression by reducing the *input (plaintext)* into the *raw output* ($\theta_{raw}=C||H$) as explained in section 2.1. That *raw output* is then compressed via *Arithmetic Coding* to provide further compression and *output* (θ) is obtained. Hence, the overall performance of the scheme depends on how large *ciphertext* (C) and *help string* (H) sizes are, as well as how good the *Arithmetic Coding* compresses the *raw output* (θ_{raw}). Below, (C) and (H) sizes (costs) are formulated. If C_i is the number of bits required to encode the *ciphertext symbol* corresponding to the i^{th} *n-gram* in the *plaintext*, then the *ciphertext* size $|C|$ becomes (Eq. 2):

$$|C| = \sum_{i=1}^{|P|} C_i \leq \sum_{i=1}^{|P|} \lceil \log_2 |S| \rceil = \frac{|P|}{n} \lceil \log_2 |S| \rceil \tag{2}$$

The maximum number of bits needed to encode a *ciphertext symbol* is $\lceil \log_2 |S| \rceil$. Whenever all *n-grams* are encoded with the same *cipher symbol* $|S| = 1$ and $|C| = 0$. On the other hand, if each *n-gram* is encoded uniquely, then we get $|S|=|A|^n$ and $|C| = |P| \times \lceil \log_2 |A| \rceil$. Thus, Eq. 3 holds for these extreme cases:

$$0 \leq |C| \leq |P| \lceil \log_2 |A| \rceil \tag{3}$$

The formula for the *help string* size $|H|$ is given in Eq. 4. If H_i is the number of bits necessary to encode the correct choice number for the i^{th} *n-gram* in *plaintext*, then:

$$|H| = \sum_{i=1}^{|P|} H_i \leq \sum_{i=1}^{|P|} \lceil \log_2 |S_i| \rceil \leq \frac{|P|}{n} \left\lceil \log_2 \frac{|A|^n}{|S|} \right\rceil \quad (4)$$

S_i is the number of n -grams represented by the *ciphertext symbol* which represents the i^{th} n -gram of the *plaintext* in Eq. 4. S_i should also satisfy that sum of all S_i ($1 \leq i \leq |S|$) must be equal to the $|A|^n$ (Eq. 5).

$$\sum_{i=1}^{|S|} S_i = |A|^n \quad (5)$$

To encode the correct one among $|A|^n/|S|$ possibilities, at most $\lceil \log_2 |A|^n / |S| \rceil$ bits are needed. If, on one extreme, all n -grams are encoded with the same symbol, then $|S| = 1$ and $|H| = |P| \times \lceil \log_2 |A| \rceil$. If, on the other extreme, each n -gram is encoded uniquely, then $|S| = |A|^n$ and $|H| = 0$. Hence, Eq. 6 holds for the *help string* size:

$$0 \leq |H| \leq |P| \lceil \log_2 |A| \rceil \quad (6)$$

As expected, there is a tradeoff between *ciphertext* size and the *help string* size: The best case for the *ciphertext* size is the worst case for the *help string* size and vice versa. Since each *plaintext* symbol can be encoded using at most $\lceil \log_2 |A| \rceil$ bits, then Eq. 7 gives the upper bound for the sum of the *ciphertext* and the *help string*.

$$|C| + |H| \leq |P| \lceil \log_2 |A| \rceil \quad (7)$$

3. RESULTS

To observe the rate of change in the level of compression and security, the presented scheme has been implemented for $n=2$ and $n=3$ on English texts *bib* and *book1* from *Calgary Corpus* [6]. In Fig. 2, average *bpc* rates for these files are shown. Implementing the scheme for both n values yields different *ciphertexts*, hence varying levels of compression. For *bib*, the compression rates are measured as 3.79 *bpc* for $n=2$ and 3.61 *bpc* for $n=3$, while for *book1* they become 3.69 *bpc* for $n=2$ and 3.32 *bpc* for $n=3$. These results show that, increasing the *text division index* (n) from 2 to 3 improves the *compression rate*. It also increases the level of security because, it is harder to determine to which *trigram* a *cipher symbol* corresponds to (for $n=3$), than determining to which *digram* a *cipher symbol* corresponds to (for $n=2$).

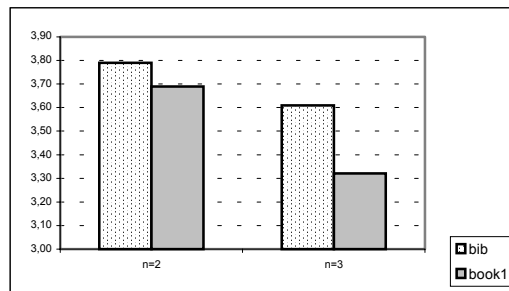


Fig. 2 Compression performances for $n=2$ and $n=3$.

We also compared the compression performances (*bpc*) yielded by our scheme for $n=2$ and $n=3$ with that of the conventional tools *gzip* [7] and *pkzip* [8]. The base algorithm used in *gzip* and *pkzip* is *Lempel-Ziv 77* [9], which is a *dictionary* based compression technique: It searches the longest pattern of the upcoming text (*look ahead buffer*) in a *dictionary* of predetermined length. If a match is found, then the matching pattern is represented with a reference to the *dictionary* to achieve compression. As seen in Table 3, standard compression tools achieve better compression rates (24.5% better in average for $n=3$). Nevertheless, our approach do provide security which lacks in conventional tools. In addition, there is a tradeoff between the degree of security and the level of compression, which means we can achieve better compression by giving up security to some degree. Our earlier work [10] shows that if you settle down for lesser security, than better compression is achievable by either employing *Encoding matrices* generated using language statistics (instead of *BBS*) or by using slower but more efficient algorithms (e.g. *PPM** [11] or *BWMN* [12]) in place of *Arithmetic Coding*.

Table 3. *Bpc* comparisons with conventional tools.

filename	new scheme n=2	new scheme n=3	Unix gzip	Windows pkzip
bib	3.79	3.61	2.32	2.31
book1	3.69	3.32	2.93	2.91
average	3.74	3.47	2.63	2.61

3.1. Iterative Encoding

The level of security provided by our system can be increased by passing the *output* (say θ_1) through the *Encoding matrix* (E) once more and obtain another *output* (say θ_2), then pass θ_2 through E to obtain θ_3 , and so on. As the iterations roll on, reflection of dissipated language statistics in θ yields harder-to-compress *ciphertexts*. This increases the overall *bpc* rate of the *output* (θ).

In Fig. 3, the *output* (θ) *bpc* rates obtained by implementing our scheme with 8 iterations on the sample *bib* file for $n=2$ and $n=3$ are given. As expected, the overall (θ) *bpc* rate for each iteration increases and converges to a stable value for both $n=2$ and $n=3$, because good cryptosystem outputs tend not to be compressible.

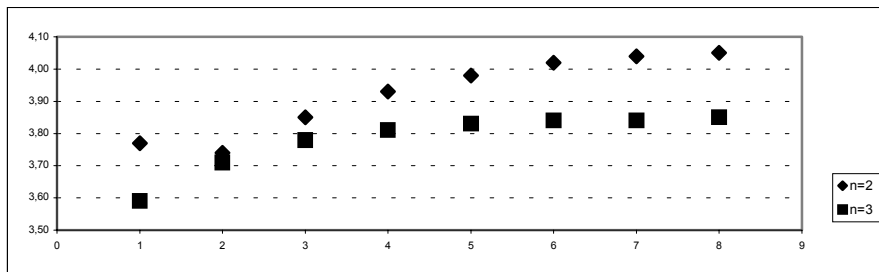


Fig. 3 Iterative Encoding *bpc* performances for $n=2$ and $n=3$.

While $n=2$, for each of the 8 iterations, the compressed (*final*) *help string* (H_{out}) over the uncompressed (*raw*) *help string* (H_{in}) ratios (H_{out}/H_{in}) are given in Table 4. These values point out that H is poorly compressed, even expands indicated by values > 1 in the last

column of Table 4, as more *iterations* are applied. This is an indication of the increasing strength of our *encoding* method.

Table 4. *Help String* Compressibility Changes in *Iterative Encoding* for $n=2$.

iteration number	C_{in} (bytes)	H_{in} (bytes)	C_{out} (bytes)	H_{out} (bytes)	H_{out}/H_{in}
1	46.118	16.075	27.199	16.312	1,0147
2	23.059	12.919	13.931	12.920	1,0001
3	11.530	8.059	7.089	8.094	1,0043
4	5.765	4.264	3.626	4.396	1,0310
5	2.883	2.158	1.877	2.336	1,0825
6	1.442	1.079	989	1.261	1,1687
7	721	540	533	707	1,3093
8	361	271	293	416	1,5372
AVERAGE					1,1435

4. CONCLUSION

Addressing the need for a compression algorithm which not only provides compression but also security, we have developed a new *encoding/decoding* scheme. Experimental results reveal that the designed system provides reasonable level of compression and security. Further security is achieved through *iterative encoding* which yields hard to break *ciphertexts* and *help strings*, reflected by higher *bpc* rates, as was expected. The results also show that increasing the *text division index* (n) from 2 to 3 leads to better compression rates. As future work, we plan on experimenting with larger n values to examine if increasing n continues to provide better compression.

5. REFERENCES

1. M. Nelson, *The Data Compression Book*, M&T Publishing, New York, USA, 1996.
2. I. Witten, A. Moffat and T. C. Bell, *Managing Gigabytes – Compressing and Indexing Documents and Images*, San Fransisco, CA, USA, 1999.
3. W. Stallings, *Network&Internetwork Security*, Prentice Hall, New Jersey, USA, 1995.
4. A. Menezes et. al, *Handbook of Cryptography*, CRC Press, USA, 1997.
5. *GAP Manual*, <http://www.gap-system.org>.
6. *Calgary Corpus*, <ftp://ftp.cpsc.ucalgary.ca/pub/projects/text.compression.corpus>.
7. *GZIP Manual*, <http://www.gzip.org/#intro>
8. D. Hankerson, G.A. Harris and P.D.Jr. Johnson, *Introduction to Information Theory and Data Compression*, CRC Press, FL, USA, 1998.
9. J. Ziv and A. Lempel, A Universal Algorithm for Sequential Data Compression, *IEEE Transactions on Information Theory*, Vol. 23, No. 3, pp. 337-343, 1977.
10. M. E. Dalkilic and E. Celikel, Concept of and Experiments on Combining Compression with Encryption, *ISCIS XVII*, Orlando, USA, 2002.
11. *PPM**, <http://ftp.cs.waikato.ac.nz/pub/compression.ppm>
12. *BWMN*, <http://www.dogma.net/markn/articles/bwt/bwt.htm>