Development of Secure LZW Technique for Secret Compressed Data Transmission

Ali M. Sagheer

***Abstract*— Compression algorithms reduce the redundancy in data representation to decrease the storage required for that data. Data compression offers an attractive approach to reducing communication costs by using available bandwidth effectively. Over the last decade there has been an unprecedented explosion in the amount of digital data transmitted via the Internet, representing text, images, video, sound, computer programs, etc. With this trend expected to continue, it makes sense to pursue research on developing algorithms that can most effectively use available network bandwidth by maximally compressing data. It is also important to consider the security aspects of the data being transmitted while compressing it, as most of the text data transmitted over the Internet is very much vulnerable to a multitude of attacks. This paper is focused on addressing the problem of lossless compression of text files wit an added security. Lossless compression researchers have developed highly sophisticated approaches, such as Huffman encoding, arithmetic encoding, the Lempel-Ziv (LZ) family. However, none of these methods has been able to reach the theoretical best-case compression ratio consistently, which suggests that better algorithms may be possible. We trying to attain better compression ratios is to develop new compression algorithms. An alternative approach, however, is to develop, reversible transformations that can be applied to a source text that improve LZW algorithm ability to compress and also offer a sufficient level of security of the transmitted information.**

***Index Terms*—Data Compression, Data Transmission, LZW, Information Security, Multimedia.**

I. INTRODUCTION

WITH the emergence of wireless sensor networks and distributed video applications, distributed source compression has become an important research area. A significant amount of effort has been devoted to understanding the information theoretic limits of distributed lossless and lossy compression and developing codes to achieve these limits. However, in many real-life applications involving distributed compression, such as distributed video surveillance or monitoring of some private information, secure compression and communication while meeting the end-to-end quality of service requirements becomes important. In this paper we consider the information theoretic limits of secure lossless source compression in the presence of an adversary who has access to some of the links in the network as well as its own correlated observation of the data to be compressed. We consider information theoretic secrecy, that is, we want to limit the information leakage to a computationally unbounded eavesdropper who has the full knowledge of the compression algorithms used.

Manuscript received Oct, 02, 2010.

Ali Makki Sagheer is with the University of Anbar, Anbar,Iraq. He is now with the Information System Department, College of Computer (Phone No.: 009647901705213; e-mail: ali_makki_sagheer@yahoo.com).

II. BACKGROUND

A. *Data Compression*

Data compression is the process of converting an input data stream (the source stream or the original raw data) into another data stream (the output, or the compressed, stream) that has a smaller size. A stream is either a file or a buffer in memory. Data compression is popular for two reasons: (1) People like to accumulate data and hate to throw anything away. No matter how big a storage device one has, sooner or later it is going to overflow. Data compression seems useful because it delays this inevitability. (2) People hate to wait a long time for data transfers. When sitting at the computer, waiting for a Web page to come in or for a file to download, we naturally feel that anything longer than a few seconds is a long time to wait [1].

B. *Lossless and Lossy Algorithms*

Compression algorithms can be either lossy or lossless. Lossy algorithms can give substantially better compression ratios in some cases because they perform nonreversible changes. One scenario in which this is acceptable could be digital sound compression. Lossless algorithms restore the original data perfectly. A general purpose algorithm which can not depend on any knowledge on the input data must be lossless. This is obviously necessary when compressing software, because one single bit of error can lead to a disastrous run-time failure.

Lossy methods are viable in some circumstances when compressing software. The requirement in such a case is that the compressed program operates exactly the same as if it had not been compressed. An example of a lossy compression that is possible for software is a compiler that optimizes for space by transforming certain operations into equivalent operations that use less storage, or removes unused functions [2].

C. Dictionary Methods

The dictionary methods, as the name implies, maintain a *dictionary or codebook* of words or text strings previously encountered in the text input and data compression is achieved by replacing strings in the text by a reference to the string in the dictionary. The dictionary is *dynamic or adaptive* in the sense that it is constructed by adding new strings being read and it allows deletion of less frequently used strings if the size of the dictionary exceeds some limit. It is also possible to use a *static* dictionary like the word dictionary to compress the text. The most widely used compression algorithms (Gzip and Gif) are based on Ziv-Lempel or LZ77 coding [3] in which the text prior to the current symbol constitute the dictionary and a greedy search is initiated to determine whether the characters following the current character have already been encountered in the text before, and if yes, they are replaced by a reference giving its relative starting position in the text. Because of the pattern matching operation the encoding takes longer time but the process has been fine tuned with the use of hashing techniques and special data structures. The decoding process is straightforward and fast because it involves a random access of an array to retrieve the character string.

A variation of the LZ77 theme, called the LZ78 coding, includes one extra character to a previously coded string in the encoding scheme. A more popular variant of LZ78 family is the so-called LZW algorithm which leads to widely used *Compress* utility, developed by Terry Welch in 1984 [4] and [5]. This method uses a suffix tree to store the strings previously encountered and the text is encoded as a sequence of node numbers in this tree. To encode a string the algorithm will traverse the existing tree as far as possible and a new node is created when the last character in the string fails to traverse a path any more. At this point the last encountered node number is used to compress the string up to that node and a new node is created appending the character that did not lead to a valid path to traverse. In other words, at every step of the process the length of the recognizable strings in the dictionary gets incrementally stretched and is made available to future steps [6].

D. Stream Ciphers

Stream ciphers have extensive applications, many of them are in the area of wireless communication. As an example, they are part of the security framework in GSM networks, Bluetooth or WLANs. There is an ongoing effort to analyze existing algorithms and to design new ones in the academic community as well as in companies or governmental organizations. The fact that well established techniques for passive side channel attacks such as Differential Power Analysis (DPA) do not work with stream ciphers increases the interest in them even more. The stream cipher is one of important secret-key cryptosystem. Several researches on stream ciphers was mainly focused on the development of efficient stream ciphers in hardware.

Linear Feedback Shift Registers (LFSRs) are the main building blocks of these ciphers, because of their good statistical properties and their efficiency in hardware. To make these LFSRs cryptographically secure, the two most common practices are the use of a Boolean function (sometimes complemented with some nonlinear memory bits) and the irregular clocking of LFSRs. Two widely used stream ciphers based on this research are A5 used in GSM mobile phones and E0 used in the Bluetooth standard [7].

E. RC4 Stream Cipher

RC4 was designed by Ron Rivest in 1987 in an attempt to make a stream cipher which is more suitable for software implementations. He did not use LFSRs at all, but used a dynamic permutation instead. The design was a trade secret of RSA Inc. but leaked 1994 when someone anonymously posted the source code to the Cypherpunks mailing list. The security of RC4 was of course not affected as it issolely based on the used key. Even though this alleged version was never officially confirmed to be equivalent to the original version by RSA Inc., there is strong evidence to assume this. Subsequently, the notation RC4 refers to the alleged version of the algorithm. RC4 is one of the most popular stream ciphers, it is heavily used in SSL/TLS or IEEE 802.11 and is integrated into many widely used open-source libraries or applications of Microsoft or Oracle. Hardware implementations [8] have been considered as well.

III. Development of Secure LZW Technique

Many applications require data to be both secured and compressed. Stream ciphers are extremely fast cryptologics but rarely used today because of improved computer speed and the ciphers lack of diffusion. Lossless compression algorithms add a bit of diffusion, with the statistics of previous plaintext effecting future output, but make it only slightly more difficult for an adversary to read traffic. Using a stream cipher to dictate the direction of the compression algorithm eliminates the need for two algorithms, improves the speed performance while leaving the compression ratio unchanged.

I combined stream ciphers with dictionary methods such as LZW, eventually; the stream cipher based on Pseudo Random Number Generator (PRNG) from input.

The LZW method starts by initializing the dictionary to all the symbols in the alphabet. In the common case of 8-bit symbols, the first 256 entries of the dictionary (entries 0 through 255) are occupied before any data is input. Because the dictionary is initialized, the next input character will always be found in the dictionary. This is why an LZW token can consist of just a pointer. In the encoding stage the pointer is XORed with the random number that is generated from PRNG of RC4, and then is stored in the output secret compressed file. The decoding stage is opposite operation, the secret number is read from the secret compressed file, then XORed with PRNG from RC4 to return the pointer of the write token in dictionary.

A. Secure LZW Encoding

The output of this improved algorithm is fixed-length references to the dictionary (indexes). Of course, we can't just remove all symbols from the output and add nothing elsewhere. Therefore the dictionary has to be initialized with all the symbols of the input alphabet and this initial dictionary needs to be made known to the decoder. Also the initialization of the PRNG from seed secret input key must be done before encoding.

The encoding algorithm consists of two phases which are executed sequentially.

An initialization phase:

- KSA (Key Scheduling Algorithm).
- Dictionary Construction.

A secure LZW encoding phase:

- PRGA (Pseudo Random Generation Algorithm)
- LZW Encoding.

Both parts access an internal table of size $n * 2^n$ bits, which can be viewed as an array containing 2^n words of size n. Let's look at the two parts of the algorithm in more detail. Even if the internal table has a size of $n * 2^n$ bits, the total number of states is not 2^{n*2n}, (which would be 2^{2048} in the case of $n = 8$). Only permutations of 2^n words are possible, thus leading to a total number of $(2^n)!$, different table-states.

Using a word-size of 8, the number of different table-states is $(2^8)! \approx 2^{1684}$.

At first, the KSA is initializing the internal state to the identity permutation. Using the words in the variable sized key, each word in the internal state is swapped with another one. The identity permutation is permuted 256 times (in the case of n=8) depending on the key. If the key is smaller than 2048 bits (again in the case of n=8), it is reused.

```
Algorithm 1: Initialization Phase
Input:   $key_0, key_1, key_2, ….., key_m$
Output:  $State_0, State_1, State_2, ….., State_{255}$,
         $Dictionary_0, Dictionary_1, ….. , Dictionary_{255}$
  1: set  $Dictionary_i := Char_i$ ;  for $i$:=0,1 , …. , 255 ;
  2: set  $State_i := i$ ;  for $i$:=0,1 , …. , 255 ;
  3: let  $j := 0$ ;
  4: for $i := 0$ to 255 do the following:
     4.1: $j := j + key_i + State_i$ ;
     4.2: swap( $State_i$ , $State_j$ );
  5: Output State and Dictionary.
```

The generation of the output operates similar to the key scheduling. The internal state is evolving (again, using the swap operation) with every generated output symbol. An outline is depicted in the following algorithm. The actual algorithm of the secure LZW encoding is as follows:

```
Algorithm 2: Secure LZW Encoding Processing Phase
Input:   Data file,
         $State_0, State_1, State_2, ….., State_{255}$
Output:  Secret compressed Data file.
  1: Set initial values
     1.1: set  $i := 0$;
     1.2: set  $j := 0$;
     1.3: set  $w :=$ NIL;
  2: While not end of input Data file do the following:
     2.1: $s$ := next symbol from input;
     2.2: if ($ws$ exists in the dictionary)
          2.1.1: $w := ws$;
     2.3: else
          2.3.1: $m$ := index($w$);
          2.3.2: add $ws$ to the dictionary;
          2.3.3: $w := s$;
     2.4: $i := i + 1$;
     2.5: $j := j + State_i$ ;
     2.6: swap( $State_i$ , $State_j$ );
     2.7: $State_k := State_{(State_i + State_j)}$;
     2.8: $c := m \oplus State_k$
     2.9: write $c$ in output Data file
     2.10: End if
  3: End While
  4: Output: Encoded Data file
```

B. Secure LZW Decoding

The decoder starts with the first entries of its dictionary initialized to all the symbols of the alphabet (normally 256 symbols). It then reads its input stream (which consists of encrypted pointers- to the dictionary-) and uses decryption of each pointer to retrieve uncompressed symbols from its dictionary and write them on its output stream. It also builds its dictionary in the same way as the encoder (this fact is usually expressed by saying that the encoder and decoder are *synchronized* or that they work in *lockstep*).

The decoding algorithm consists of two phases also which are executed sequentially.

An initialization phase:

- KSA (Key Scheduling Algorithm).
- Dictionary Construction.

A secure LZW decoding phase:

- PRGA (Pseudo Random Generation Algorithm)
- LZW Decoding.

The initial phase is the same as in the encoding.

The actual algorithm of the secure LZW encoding is as follows:

```
Algorithm 3: Secure LZW Decoding Processing Phase
Input:   Secret compressed Data file,
         $State_0, State_1, State_2, ….., State_{255}$
Output:  Data file.
  1: Set initial values
     1.1: set  $i := 0$
     1.2: set  $j := 0$
     1.3: set  $w :=$ NIL
  2: While not end of input Data file do the following:
     2.1: $c$ := next symbol from input
     2.2: $i := i + 1$
     2.3: $j := j + State_i$
```

```
        2.4: swap( $State_i$ , $State_j$ )
        2.5: $State_k \leftarrow State_{(State_i + State_j)}$
        2.6: $m := c \oplus State_k$
        2.7: s:= $Dictionary_m$;
        2.8: if (ws exists in the dictionary)
            2.1.1: w := ws;
        2.9: else
            2.3.1: write s in output Data file;
            2.3.2: add ws to the dictionary;
            2.3.3: w := s;
        2.10: End if
    3: End While
    4: Output: Decoded Data file.
```

IV. Implementation

The proposed technique is implemented in C++, the input key is variable size as the level of security is needed. The security of any cryptographic technique is the security of the used key. The key be secure as long as is large size. The input key represents the seed of the PRNG, which used as an expanded key to XOR with output code from LZW. This technique is implemented for various Data files with different size and value using several key lengths. Which give a secure code with efficient compression ratio.

Example:-

- Let the input key is [56, 24, 59, 29, 93, 102, 41, 199] with size of 64-bit.
- Let the input Data file contains:

"**sir sid eastman easily teases seas sick seals**"

- The secure LZW code file is contain:

"113 108 123 46 276 127 3 73 87 50 57 55 9 25 385 400 195 209 168 198 136 276 88 33 45 378 384 433 185 140 100 297 51 25 234",

the secure LZW code con not be decoded without the generation the PRNG from the input key which used the encode the Data file.

- While the original LZW code file is contain:

"115 105 114 32 256 100 32 101 97 115 116 109 97 110 262 264 105 108 121 32 116 263 115 101 115 259 277 259 105 99 107 281 97 108 115",

which is change according to changing of the input key.

V. Conclusion

The simplicity of the round algorithm also makes for ease of implementation, reducing the likelihood of errors. As well, the number of operations required is quite small, making it efficient. Further, only byte operations are required making it efficient when implemented on small processors, and the memory requirements are reasonable. On the other hand, it is not easy or efficient to the byte operations do not take advantage of the wider busses available on newer processors. As a result there has been interest recently in adapting the algorithm to make use of wider busses.

In the proposal secure LZW technique the difficulty of knowing where any value is in the table and difficulty of knowing which location in the table is used to select each value in the sequence. Also the compressed code has less redundancy -high randomness-, the encryption of the compressed code became more secure and randomness than encryption of the classical plaintext. The proposed technique appears more strength and secure with the same of the compression ratio of the original LZW.

References

[1] David Salomon, *Data Compression*, 3rd ed. Springer-Verlag New York, Inc., 2004, pp. 1,–64.

[2] Nicolas Tsiftes, "Using Data Compression for Energy-Efficient Reprogramming of Wireless Sensor Networks", M.S. thesis,Computer Science, Stockholm University, Sweden 2007.

[3] J. Ziv and A. Lempel. "A universal algorithm for sequential data compression", *IEEE Transactions on Information Theory*, IT-23:337{343, 1977.

[4] Welch, T. A. "A Technique for High-Performance Data Compression," *IEEE Computer* **17**(6):8–19, June,1984.

[5] Phillips, Dwayne, "LZW Data Compression," *The Computer Application Journal* Circuit Cellar Inc., **27**:36–48, June/July,1992.

[6] Nan Zhang, "Transform Based and Search Aware Text Compression Schemes and Compressed Domain Text Retrieval", Ph.D. dissertation, School of Computer Science, College of Engineering and Computer Science, University of Central Florida, Orlando, Florida, 2005.

[7] Golomb, S. W., "Shift register sequences", Aegean Park press, 1982.

[8] P. Kitsos, G. Kostopoulos, N. Sklavos, and O. Koufopavlou. Hardware Implementation of the RC4 Stream Cipher. In *46th IEEE Midwest Symposium on Circuits & Systems '03*, 2003.

Ali Makki Sagheer was born in Basrah-1979. He got on B.Sc. in Computer Science Department at the University of Technology (2001)-Iraq, M.Sc. in Data Security from the University of Technology (2004)-Iraq and Ph.D. in Computer Science from the University of Technology (2007)-Iraq.

He is interesting in the following Fields (Cryptology, Information Security, Number Theory, Multimedia Compression, Image Processing and Coding Systems). He is published many papers in different conferences and scientific magazines.