Implementing the RC4 Algorithm

Brian Whitley
Computer Science 3 - CSI 3050
Term Paper

April 17, 2004

TABLE OF CONTENTS

Executive Summary

The students in Computer Science 3 are creating a B-Tree structure in the C++ programming language. This structure is commonly used for database systems because of its speed for data lookup, retrieval, and storage. The B-Tree structure uses a buffer class to store the lookup nodes, called BTreeNodes. These buffers are character arrays that save data in a plain text format when written to disk. Software security should be a primary design consideration for any application in our connected world. As such, I am recommending the RC4 algorithm as an encryption method for these buffers.

RC4 is in use today in web servers to secure content delivery over the Internet. RC4 is also used in wireless networking equipment to encrypt data transmissions over the airwaves. Its common usage is due to two factors, ease of implementation, and speed. RC4 is easy to implement because it only requires a few lines of code. It generates a cipher-stream using a private key. The cipher-stream is used to encrypt plain text messages and decrypt encrypted messages. The same algorithm performs both operations.

Implementing the RC4 Algorithm discusses the basics of encryption, explains the RC4 algorithm, and provides an implementation in C++. Lastly, a proposal for encrypting BTreeNode buffers is included. Students with a basic understanding of programming should be able to grasp the concepts presented here. With a little effort, the B-Tree structure can be encrypted.

Introduction

All good software design should consider security as part of the design process. Applications that transmit data over a network or save data to network servers should implement a security scheme. In today's world of Internet connected computers, every application should consider security the top priority even for stand-alone applications. I have been working on an implementation of a B-Tree using a character buffer for Dr. Hasz's CSI 3050[1] course at Metropolitan State College of Denver. In this paper I propose a method of securing the BTreeNode buffers as they are written to disk using the RC4 stream cipher algorithm. RC4 is the algorithm of choice because of its ease of implementation and its speed. Implementing the RC4 algorithm requires us to look at the algorithm in detail. This paper will provide a brief history about RC4, background information about encryption techniques, and a detailed description of the RC4 algorithm.

A Brief History of RC4

RC4 is a stream cipher designed by Ronald L Rivest[2] for RSA Security. Ron Rivest is a Professor of Computer Science at MIT's Department of Electrical Engineering and Computer Science. Ron is also the R in RSA. I mention Ron because RC is often refered to as "Ron's Code" or "Rivest Cipher" and his home page has numerous articles explaining his various ciphers.

RSA holds the trademark for RC4 and commercial use is permitted only with a license from them. RC4 was released in 1987 and its internal workings are only disclosed by RSA with the purchase of a license. According to Wikipedia, the online encyclopedia (MediaWiki), RC4 was anonymously leaked to the Internet Cypherpunks (Sterndark) mailing list in September 1994. Members of the Cypherpunks compared the "Alleged RC4" to RSA's version and

determined the results produced are compatible with each other. After the code was re-written it has been renamed to ARCFOUR to avoid trademark issues. Since I am not a legal expert, I will recommend you contact RSA for license information before using ARCFOUR in a commercial application.

Simple Cryptography

Cryptography is the science of keeping a message secret from those who are not supposed to see the message. In the world of electronics and computers there are numerous methods used to accomplish this goal. Most cryptographic algorithms can be boiled down into a fairly simple process that uses modular arithmetic. The book, “Codes Ciphers and Computers” (Bosworth) describes modulo 2 addition as binary addition without the carry. It is commonly called an exclusive OR (XOR). The symbol (⊕) is used to represent an exclusive OR. The rules for modulo 2 addition (⊕) are:

$1 \oplus 1 = 0$
$1 \oplus 0 = 1$
$0 \oplus 1 = 1$
$0 \oplus 0 = 0$

The process of using modulo 2 addition to create a cipher is to take a binary message and XOR a secret key to produce a cipher text message. The example below uses the ASCII character set (Prata) to represent a plain text message, “Hello” and a secret key, “Secret” in binary bit strings. XORing them together produces the cipher text message.

Message:	H	e	l	l	o	!
	01001000	01100101	01101100	01101100	01101111	00100001
⊕						
Secret Key:	S	e	c	r	e	t
	01010011	01100101	01100011	01110010	01100101	01110100
=						
Cipher text:	00011011	00000000	00001111	00011110	00001010	01010101

The cipher text can then be transmitted over the Internet or saved to disk and the only the person that has the secret key can retrieve the message. To undo the cipher text, XOR the cipher text with the same secret key and the original message is reproduced.

Secret Key:	S	e	c	r	e	t
	01010011	01100101	01100011	01110010	01100101	01110100
⊕						
Cipher text:	00011011	00000000	00001111	00011110	00001010	01010101
=						
Message:	01001000	01100101	01101100	01101100	01101111	00100001
	H	e	l	l	o	!

This is fairly simplistic and anyone could write a program that could crack the cipher text that is produced. To make it harder to crack, an algorithm such as RC4 is used to generate the secret key.

Secret Key Selection

Ciphering is done with many differing strategies for creating and managing secret keys. A complex system that has a public key and a private key can be used. The public key is shared with anyone who wants to send an encrypted message. The private key is used to decrypt the

message. To send a reply the public key for the receiving party must be used. They would use their private key to decrypt the message.

Bi-directional communications between two parties requires 4 keys, a set of public, and a set of private keys. As the number of parties that need to communicate securely increases, it becomes impractical to remember all the keys involved. To make the public – private key system work requires sophisticated algorithms and substantial infrastructure developed to track and manages all the keys.

RC4 uses a much simpler method, a shared secret where both (or multiple) parties agree upon a secret key. The same key is used to encrypt and decrypt the message. Every party involved uses the same key. Since the same key is used everywhere, the algorithm should follow some simple design guidelines (Meyer). To make the cipher unbreakable:

1. The key must be randomly selected.
2. The key must be used only once.
3. The key must be of equal or greater length than the message to encrypt.

There are two basic algorithm categories that produce a secret key, block ciphers and stream ciphers. Block ciphers are used when the message to be encrypted is a known set size (Meyer, p.23). Stream ciphers are used when the size of the message is not known until it is encrypted. A typical use for a stream cipher is data transmitted over a network. Since our buffer class can be any size we will use a stream cipher that meets the guidelines above.

Stream Ciphers

Both sides of the cipher process must know and use the same key to encrypt and decrypt the message. Assuming the message was really large, it is not practical to use a fixed key of equal size. Following the design guidelines, stream ciphers solve this basic problem by generating an infinite key. This means the secret key is of endless or infinite length, is non-repeating, and randomly selected.

A stream cipher (Meyer p. 53-61) starts with a small secret key that is permutated into a cryptographic bit-stream. This bit-stream generator (C) is used to XOR a plaintext message (M) into a cipher text message (P). The same algorithm is used on the other end to XOR the cipher text message back into plaintext.

$M \oplus C = P$ Encryption
$P \oplus C = M$ Decryption

In order to synchronize the bit-streams an initialization vector (IV) is sent at the start of each new message. The IV tells the stream cipher how many permutations have occurred since it started. You cannot backtrack in a stream cipher, instead if the two side become out of sync one side must start from the beginning and perform the number of permutations the IV states. The IV is included at the start of a message and is in the clear. That means anyone can read the IV, but if they don't have the shared secret it is very difficult to crack the stream cipher.

This leads to an interesting feature of stream ciphers. They will produce the exact same bit stream when started from the beginning provided the same secret key is used. In order to achieve our design guidelines we must never repeat an IV or we must restart the stream cipher using a new key.

RC4 Internals

RC4 is a stream cipher (Schneier). It has two functions, stream setup and byte generator. The stream setup is accomplished by creating two 256-element arrays. Both arrays have elements of size byte.

The first called S[256] is filled with the values 0 to 255. In binary that represents every bit combination in a byte. The second array called K[256], is filled with the shared secret key. The key can be any length, 40 bit, 128 bit, 256 bit. It is broken into byte segments and copied byte by byte into K. If we use a 128-bit key, the first 16 array cells would be filled with the key. Then the key is repeated to fill the next 16 array cells. This is repeated until all of the array cells are filled.

To complete the stream setup the array must be randomized. This is done using two counters, A and B. Both counters are initialized to zero. Then inside a loop the following is performed to pseudo-randomly generate the starting point for the stream:

```
for (A = 0; A < 256; A++)
        B=(B + S[A] + K[A]) mod 256
        swap S[A] and S[B]
```

This sets up the stream generator. The stream is the bytes contained in the S[256] array. Every element in S[256] is swapped with a random element. The random element selection is made by adding the last element pointer (B) to the contents of the current element S[A] and to the key value in K[A]. This creates a large number that is modulus 256. This will produce a number between 0 and 255 that is used to find the random swap location. Since both sides of the communication channel use the same key to perform stream setup, both stream generators will have the same contents in the S[256] array.

Stream setup initializes two counters used for the byte generator, I and J. Both are initialized to zero. The byte generator is then used to get a single byte from the stream. The following process is performed:

I = (I + 1) mod 256
J = (J + S[I]) mod 256
swap S[I] and S[J]
K = (S[I] + S [J]) mod 256
return S[K]

The byte stream increments I and performs modulus 256. Then it adds J to S[I] and performs modulus 256. This sets I and J to a number between 0 and 255. In effect this is a double pseudo-random number generation and the contents of S[256] at elements I and J are swapped. This randomizes the array every time the byte generator is called. Taking the sum of two random elements and then mod 256 provides the element location to return. One pseudo-random byte is returned on every call.

The initialization vector is variable I. It counts the number of times through the encryption process to synchronize the cipher stream. It is left as an implementation problem in the documentation about RC4 that I have read. For our buffer encryption we will use the IV as described in the section “Encrypting the Buffer Class”.

Implementing RC4 in C++

Using the RC4 definition in this paper and examples from the open source project AirSnort[3], I derived the source files to implement RC4 in C++. The source is located in Appendix A. The simple header file describes the methods RC4 will use. There is a constructor and three methods, stream_setup, getStream, and rc4Crypt.

The constructor takes no parameters. When the user instantiates the constructor an RC4 object is returned. It must be setup before it can be used.

Stream_setup takes the users KEY and copies it into the K[256] array. Each key byte is copied into one cell of the K[256] array until all of the key has been placed into the array. If the array is not full the key repeats until the array is full. It then initializes S[256] and randomizes it using a couple of loops. In RC4 the key is not used to cipher, only to randomize the S[256] array. Choosing a different key will produce a different cipher stream. The S[256] array is used to cipher.

The getStream method performs the randomize function once and then returns a randomly selected byte from S[256].

Lastly, rc4Crypt takes a character array and either encrypts or decrypts it by performing an XOR with the cipher stream provided by getStream. When it has performed its operations on each byte it then returns the character array.

To test the implementation of RC4 a test suite was developed. Two files comprise the RC4 stream cipher, rc4.h and rc4.cpp. A third file, rc4-driver.cpp is used to test the implementation. The test creates two RC4 objects. One is used to cipher a plain text message into a cipher text message. The second object is initialized with an identical key. It is used to decipher the encrypted message back into plain text. The initialization vector for this test suite is

zero. That means that the number of permutations each RC4 object is put through prior to the encryption / decryption of the test message is zero. Compiling and running the test case on Linux using the GNU C++ compiler produced the following results:

```
[bwhitley@FLY rc4-cpp]$ g++ rc4.cpp rc4-driver.cpp
[bwhitley@FLY rc4-cpp]$ a.out
************************************************************
* RC4 Test - one RC4 object encrypts another decrypts      *
************************************************************
Using the key abcdefg
The original data is hello world
The encrypted stream is "Vï¿½"â–’Bï¿½
The decrypted stream is hello world
[bwhitley@FLY rc4-cpp]$
```

The results from the test case prove the implementation of RC4 works. The plain text is ciphered into encrypted text and back to plain text. The original message is identical to the deciphered message.

In real world use, the implementation of the methods could be in a cpp file or in the header itself. In this test implementation I placed the methods in a separate cpp file.
In the constructor the S and K arrays can be new'd off, however GNU C++ on Linux did not require it, so the lines are commented out. Also the stream_setup method could be part of the constructor. It would require the constructor to take a character array as a parameter. I didn't implement RC4 this way because I thought it would be useful to have the ability to call the stream_setup method without having to create a new RC4 object each time.

Encrypting the Buffer Class for BTreeNodes

The Buffer class uses a character array whose size differs based upon the users instantiation of the buffer. The user can create several different buffers each with a different size and save them to a single file, buffer after buffer. This unknown, varying size, is the ideal data for a stream cipher.

The Buffer class can be extended by a CryptoBuffer class that over-rides the iostream methods, readFrom and writeTo. The writeTo method can make a temporary character pointer that contains the encrypted buffer character array. After creating the encrypted array the writeTo method can then write the encrypted buffer to the file stream.

```
ostream& CryptoBuffer::writeTo(ostream& stream) {
        IVSetup(IV);
        char * tempbuffer = rc4Crypt(buffer);
        return stream.write(tempbuffer, size);
}
```

The readFrom method is a little trickier to implement. As the stream is read it needs to be decrypted in parts starting with the first four bytes. This is the Buffer's size field that tells you how big the buffer is on disk. After reading and decrypting the size, the readFrom should set the size field for the buffer and read the next 4 bytes. This is the field count. Again, it needs to be decrypted and the field count variable set. The remaining buffer can then be read from disk, decrypted and placed into the real buffer. The last step would be to read the delimiter character out of the buffer and set the delimiter variable. The code would look something like this:

```
istream& CryptoBuffer::readFrom(istream & stream) {
        // read the first 4 bytes, convert to int,
        IVSetup(IV);

        char * temp = new char[4];
        stream.read(temp, 4);         // read the four bytes
        temp = rc4Crypt(temp); // decrypt the four bytes
        size =*( reinterpret_cast<int *> (temp));    // cast into int

        buffer = new char [size];            // create a new buffer
        char * temp = (char *) (&size);
        memcpy(buffer, temp, 4);             // put size into the buffer

        // get the field size
        stream.read(temp, 4);                // read next four bytes
        temp = rc4Crypt(temp);
```

```
        fieldCount =*( reinterpret_cast<int *> (temp)); // cast 2 int
        memcpy(buffer + 4, temp, 4);          // put fieldCount

        // get the rest of the stream
        temp = new char [size - 8];
        stream.read(temp, (size - 8));
        temp = rc4Crypt(temp);

        memcpy(buffer + 8, temp, (size - 8)); // put remainder

        // get the delimiter
        memcpy(temp, buffer + 8 + (4 * fieldCount), 1);
        delimiter = temp[0];

        return stream;
}
```

The CryptoBuffer class would contain the source methods for RC4. Its constructor would set up the S[256] array with a key that is passed in using another method. The BTreeNode class would then extend the CryptoBuffer instead of extending the plain Buffer.

To cipher the Buffers as they are written and read from a file the CryptoBuffer acts as an inline stream-ciphering object. The CryptoBuffer would have to track the buffers position in the file. This is the initialization vector for the cipher. A buffer in the first zero position of the file is ciphered with the first RC4 permutation of S[256]. A buffer in position 28 is ciphered with the 28^{th} permutation of S[256]. To handle this we will need to do some work in both the buffer and the B-Tree. The BTreeNode object keeps track of what number location in the file it is written to. It also knows what number location in the file its children are located at. Using these numbers the CryptoBuffer calls an IVSetup method that performs the appropriate number of permutations to the RC4 S[256] array. This sets up the BTreeNode object to be ready to read from the file or write to the file.

The IVSetup method has to perform a new stream setup to read and write. This insures an identical cipher stream is created for the read and write process. The IVSetup method calls the getStream method in a loop and discards the results. The loop is processed according to the IV passed in. When it is done the S[256] array is set to the appropriate permutation for this buffer. As long as no two IV's are used the results should be unique.

Conclusion

Creating a CryptoBuffer places the responsibility for encryption in the buffer. The user of the B-Tree doesn't need to know or understand RC4. RC4 is simple enough that a programmer can remember how to use and implement it in only a few lines of code. If the algorithm is implemented correctly it should be fast enough to perform its cipher operations without slowing down the file system access. The suggested RC4 implementation for B-Tree applications should work. However, there may be some fine-tuning required to keep the encryption algorithm synchronized so it properly encrypts and decrypts.

Appendix A – The RC4 C++ source code.

rc4.h – the header file:

```
#ifndef RC4_H         //  To 'wrap the Stack class so it is
#define RC4_H         //  never included more than once in a project

class RC4 {
       public:
              RC4(); // Constructor.  It takes no arguments.

              void stream_setup(char* key); // initializes the RC4 algorithm
              char getStream();             // returns a byte of cipertext
              char* rc4Crypt(char* data);   // takes a char * and encrypts it
                                            // returning char *

              //======== THE CLASS INSTANCE VARIABLES ========
       private:
              int S[256];    // holds the cipher text stream
              int K[256];    // holds the cipher key
              int i;         // both ints used to maintain the state
              int j;         // in the cipher text stream array

};
#endif  // ============= END OF WRAPPER CONSTRUCT =============
```

rc4.cpp – the method implementation:

```
#include "rc4.h"
#include <cstdlib>
#include <string>
using namespace std;

	// Constructor.  It takes no arguments.
	RC4::RC4(){
		//S = new int[256];    // some C++ implementations require
		//K = new int[256];    // the arrays be new'd off
		i = 0;                 // set the I and j variables to zero
		j = 0;
	}

	// initializes the RC4 algorithm
	void RC4::stream_setup(char* key){
		int key_length = strlen(key);

		int a = 0;
		int b = 0;
		int temp = 0;

		for (a = 0; a < 255; a++){
			S[a] = a;              // initialize S[i] = i 0 ... 255
			K[a] = (int)key[b];    // init K[i] by coping Key byte
			b++;
			if (b > key_length){
				b = 0;
			}
		}

		for (a = 0; a < 255; a++){
			b = (b + S[a] + (int)K[a]) % 256;
			temp = S[a];
			S[a] = S[b];
			S[b] = temp;
		}
	}

	// returns a byte of cipertext
	char RC4::getStream(){
		int temp = 0;
		i = (i + 1) % 256;
		j = (j + S[i]) % 256;

		temp = S[i];
		S[i] = S[j];
		S[j] = temp;
		int t = (S[i] + S[j]) % 256;
		char* stream = (char *)(&S[t]);
		return stream[0];
	}

	// take a char * and encrypts it returning char *
	char* RC4::rc4Crypt(char* data){
		char crypto;
		int data_length = strlen(data);
		for (int index = 0; index <= data_length; index++){
			crypto = getStream();
			data[index] = data[index] ^ crypto;
		}
		return data;
	}
```

rc4-driver.cpp – the test suite:

```
#include "rc4.h"
#include <cstdlib>
#include <iostream>
#include <string>

using namespace std;  //introduces namespace std

int main( void )
{

	cout << "*****************************************************************\n";
	cout << "* RC4 Test - one RC4 object encrypts another decrypts          *\n";
	cout << "*****************************************************************\n";

	char key[8] ={'a', 'b', 'c', 'd', 'e', 'f', 'g', '\0'};

	char data[12] ={'h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '\0'};

	cout << "Using the key " << key << "\n";
	RC4 encrypt;           // This is the encryption object
	RC4 decrypt;           // This is the decryption object

	encrypt.stream_setup(key);           // set both RC4 objects up with the same key
	decrypt.stream_setup(key);

	cout << "The original data is " << data << "\n";
	char* data_encrypted = encrypt.rc4Crypt(data);      // encrypt the data

	cout << "The encrypted stream is ";                 // show it is encrypted
	int max = strlen(data);
	for (int loop = 0; loop <= max; loop++){
		cout << data_encrypted[loop];
	}
	cout << "\n";

	char* data_decrypted = decrypt.rc4Crypt(data);      // decrypt with different object
	cout << "The decrypted stream is " << data_decrypted << "\n";      // prove it!
};
```

End Notes

[1] For more information about B-Tree's, see Hasz.

[2] Ronald Rivest maintains a FAQ page that has references and information on his encryption algorithms, see Rivest.

[3] The IEEE 802.11 wireless networking protocol implements RC4 for the Wired Equivalent Privacy (WEP) protocol. The implementation is flawed allowing tools such as AirSnort to decipher the WEP encrypted packets. Research on WEP lead me to write this paper. For more information on the flaws see the papers written by, (Borisov), (Fluhrer), and (Walker). For more information on Airsnort see, Hergele.

References

Borisov, Goldberg, Wagner. (2001). *Intercepting Mobile Communications: The Insecurity of 802.11*. Proceedings of the 7th annual international conference on Mobile computing and networking NewYork: ACM Press. P. 189-199. International Conference on Mobile Computing and Networking [Online Archive] http://portal.acm.org/citation.cfm?id=381677.381696&dl=GUIDE&dl=ACM&type=series&idx=381677&part=Proceedings&WantType=Proceedings&title=International%20Conference%20on%20Mobile%20Computing%20and%20Networking&CFID=20252987&CFTOKEN=24016103

Bosworth, B. (1990). *Codes Ciphers and Computers: An Introduction to Information Security.* (9th ed.). New Jersey: Hayden. p. 149-150.

Fluhrer, Mantin, Shamir. Weakness in the Key Scheduling Algorithm of RC4. http://www.drizzle.com/~aboba/IEEE/rc4_ksaproc.pdf

Hasz, E. (2004, March 25). *CSI3050 Computer Science 3: Class Homepage.* http://clem.mscd.edu/csi3050/

Hergele, et al. (2004). AirSnort (Version 0.2.4a) [Computer Software] http://airsnort.shmoo.com/ http://sourceforge.net/projects/airsnort

MediaWiki (2004, April 17). *RC4 (Cipher).* http://en.wikipedia.org/wiki/RC4_cipher

Meyer, C. (1982). *Cryptography: A New Dimension in Computer Data Security.* New York: John Wiley. p. 21-23 p. 53-61.

Prata, S. (2002). *C++ Primer Plus.* (4th ed.). Indianapolis: SAMS. p. Appendix C.

Rivest, R. (2004, March 16). *Ronald L. Rivest: Homepage.* http://theory.lcs.mit.edu/~rivest/

Schneier, B. (1996). *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* (2nd. ed.). New York: John Wiley p. 203-205 p. 397-398.

Sterndark, D. (1994, September 14). *RC4 Algorithm Revealed.* [Msg 1]. Message posted to news://sci.crypt http://groups.google.com/groups?selm=sternCvKL4B.Hyy%40netcom.com

Walker, J. (2000). *Unsafe at any key size; An analysis of the WEP encapsulation.* IEEE Working Group P802.11. Document Number IEE802.11-00/362 [online] http://grouper.ieee.org/groups/802/11/Documents/DT351-400.html