Analysis of CRC Methods and Potential Data Integrity Exploits

B. Maxwell, D.R. Thompson, G. Amerson, and L. Johnson
Computer Science and Computer Engineering Department
University of Arkansas, Fayetteville, AR, U.S.A.

Abstract

Cyclic redundancy checks (CRCs) are commonly used to detect errors from noise in networks. They also have been used to verify the integrity of files in a system to prevent tampering and suggested as a possible algorithm for manipulation detection codes. It has been known that a CRC will not detect all errors but with random noise it is unlikely. But a malevolent adversary can efficiently modify a file without changing its length to maintain the same CRC. In this paper, we present an efficient algorithm for modifying an individual byte of a file and other additional bytes to maintain the same CRC. A previous attack on CRC padded a file with additional bytes to maintain the same CRC. In this work, the original file length is maintained and it demonstrates that a CRC, although good for error detection, may not be the best choice for a hash function without additional modifications.

1. Introduction

Cyclic redundancy checks (CRCs) were originally used to detect errors in hardware and in communication links to detect bursts of errors [1]. But now they are being used to validate the integrity of data in software applications [2] and in the wired equivalent privacy (WEP) protocol for the 802.11 standard for wireless networks [3], because they appear to offer good error detection and are easy to implement. But, using a CRC for an integrity check instead of its intended use for detecting random errors can provide a false sense of security. In this paper, a technique is presented that can be used to modify an individual byte of a message and other additional bytes to maintain the same CRC as the original message while keeping the length of the message the same. It is known that padding a message with extra bytes can be used to maintain the same CRC but a method to maintain the same CRC with the same message length has not been found in the formal literature [4]. This is similar to the method found in [3] for modifying an existing message and its CRC so that the CRC remains valid.

There are three common ways of implementing CRC. The first is using polynomial division, the second uses a pre-computed table to increase the speed of the algorithm, and the third is the direct lookup table or fast table method. These three methods are described briefly in the next section. Then, an algorithm will be described that reverses the CRC table method; illustrating that it is possible to find a sequence of bytes that will change the CRC to any chosen value. In other words, the CRC algorithm becomes useless in detecting that the message was changed, because the CRC will be the same although data integrity has been compromised.

2. CRC methods

2.1. Polynomial division

Computing the CRC of a message uses polynomial division. The message is considered as a large binary number and the sender divides it by a predetermined polynomial divisor. This computation yields a remainder, which becomes the checksum of the message. This checksum is used to verify that the message has not been changed. The recipient of the message calculates the CRC using the same polynomial division and compares this checksum with the checksum associated with the original message.

The actual implementation of computing checksums does not use the division scheme. It uses the notion of polynomial modulo-2 arithmetic. Binary numbers are added without carries. Another property is that addition and subtraction are equivalent to the exclusive-or (XOR) operation. For division to take place, the greatest magnitude of two numbers is determined by the number with the highest 1 bit position [5].

CRC computation is polynomial division, but is equivalent to XORing particular values at shifted offsets. There is a divisor, which is called the generator polynomial. The message is divided by the generator polynomial. The quotient is irrelevant, but the remainder is relevant because it becomes the checksum. The width w of a polynomial is defined as N-1 for a polynomial of size N. There is always an assumed 1 at the beginning of the generator polynomial and is ignored in some of the algorithms. Before division, w zeros are appended to the message where N is the length of the polynomial. There

are two ways to verify the CRC. First, the receiver of the message can compute the CRC and compare the checksum to the newly calculated CRC. Or the entire message including the appended checksum can be divided by the generator polynomial to determine if it equals zero [5].

A problem arises quickly with any implementation of CRC because polynomial division is different than numerical division on processors. Also, processors don't have registers large enough to hold big dividends. So the CRC division is implemented using a register and shifting the message through it. Assume that the polynomial has width w and the division register also has width w. Therefore, the message will have w zeros appended to it before the CRC operation. The pseudo-code for this algorithm is shown in Fig. 1.

```
Zero out CRC register.
Append W zeros onto message.
While more message is left
  Test the top bit of the register,if equals 1
    then
      Shift message left 1 bit
      XOR polynomial with register and store
      contents back in register
    else
      Shift message left 1 bit
register now equals the remainder.
```

Figure 1. Pseudo-code for Bit-Oriented Method

By comparing this algorithm to the long division algorithm, it becomes clear that this algorithm is subtracting (XORing) shifted versions (powers) of the polynomial from the message. This is equivalent to the long division method. This method will be called the bit-oriented shift-register method in the rest of the paper.

2.2. Table method

The bit-oriented shift-register method is a good way to introduce the way the CRC algorithm works but is not efficient or practical for implementation. The algorithm must loop through each bit in a message, which can be very large, and is very difficult to code in most programming languages. The first improvement is to decrease the number of loops. The algorithm is modified to process bytes instead of bits. The following example will be using a 4-bit generator polynomial 1001 and take advantage of the fact that computers handle bits in groups of bytes instead of a serial bit stream [6]. The register looks the same as in the bit-oriented shift-register method, except the register processes bytes instead of bits. Let the top 8 bits of the 32-bit register (byte 3) be:

$x_7\ x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0$

and let the top 8 bits of the polynomial be:

$p_7\ p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0$

On the next iteration of the algorithm, x_7 is popped off and examined to determine if the next 8 bits (now the top 8 bits) should be XORed with the generator polynomial. If x_7 is a one, it will be XORed, otherwise it will not. A better way to look at this is to always XOR the polynomial with the top 8 bits of the register; however the polynomial is multiplied by x_7 first. If x_7 is 0, then you'll be XORing the top 8 bits with 0, which doesn't change the top 8 bits. However, if it's a one, you'll do the XOR as expected [7].

$$\begin{array}{lll} & x_6\ x_5\ x_4\ x_3\ x_2\ x_1\ x_0\ ?? & \\ + \quad x_7 * (p_7\ p_6\ p_5\ p_4\ p_3\ p_2\ p_1\ p_0) & & (+ \text{ is XOR}) \end{array}$$

The new top bit will control what happens in the next iteration of the loop; that value is $x_6 + x_7 * p_7$. The important thing is that the information required in calculating the new top bit was present in the top two bits of the original top byte. Taking this a step further, the next top bit could be calculated from the top three bits. In general, the value of the top bit in the register after k iterations can be calculated from the top k bits. Here's a clear example of how the method executes (the bold value is the second byte). The polynomial 1001 is used in the example below and it is assumed that the most significant bit in the register was a one.

```
0100010011101000              Register
1001                          XOR this
  0000 .                      XOR this
  . 1001 . .                  XOR this
      0000 . . .              XOR this
        0000 . . .            XOR this
          0000 . . .          XOR this
            1001 . . .        XOR this
              0000 . . .      XOR this
--------------------------
1111001010101000
```

What if instead of all doing all these individual XOR calculations, that a single value could be pre-calculated and then just perform one XOR and get the same result? Well this is exactly what the table method strives to achieve. It is able to look at the top byte and determine what the individual XORs should be and therefore create the individual value. The example above could be done in one XOR calculation as shown below:

```
0100010011101000      Register
10110110010           XOR this
--------------------------
1111001010101000
```

With the information shown in this example, pseudo code can be derived. The new algorithm for calculating the CRC is shown in Fig. 2.

```
While(more bytes to process in message )
  Begin
   Calculate the control byte from the top byte
     of the register
   Sum all the polynomials at various offsets
     that are to be XORed into the register in
     accordance with the control byte
   Shift the register left by one byte, reading
     a new message byte into the rightmost byte
     of the register
   XOR the summed polynomials to the register
  End
```

Figure 2 Pseudo-code for byte-oriented method

The algorithm has been changed a little, but the improvement isn't really there yet, because the algorithm is still having to look at each individual bit for summing the polynomials at different levels. However, the calculation of summed polynomials can be pre-computed and placed in a lookup table. This leads to the algorithm shown in Fig. 3.

```
While( more bytes to process in augmented
message )
  Begin
    Top = top_byte(Register);
    Register = (Register << 8)
      | next_augmessage_byte;
    Register = Register XOR
      precomputed_table[Top]
End
```

Figure 3 Table method

This new algorithm only requires a shift, OR, and XOR for each byte. Fig. 4 is an actual implementation of this in C.

```
theRegister = 0;
while(messagelength-->0)
{
  byte temp = (theRegister >> 24) & 0xFF;
  theRegister   =   (theRegister   <<   8)   |
*augMessagePtr++;
  theRegister ^= table[temp];
}
```

Figure 4 Table method in C

2.3. Fast table method

This algorithm in section 2.2 is referred to as the table algorithm. The algorithm is efficient. However, there is some room for improvement. Adding on those zeros at the end seems is inefficient. Notice that w/4 of the zero bytes are pulled in, however they have no effect in this algorithm. The zeros do not change the values after an XOR. The only purpose of the zero bytes is to push the real bytes through. In addition, there is some wasted time initially because it takes four iterations to get the real data started processing. The first four iterations do not do anything, because they are just zeros. Even if it's not a zero, all that would happen is to have multiple XORs with different offsets of a constant value. The initial value of the register would just be what is left after four iterations of whatever the default value is for the register. If it started with a zero the final value would be a zero also. This leads to an improved algorithm called the direct table algorithm shown in Fig. 5.

```
While( more bytes to process in message )
  Begin
    Top = top_byte(Register);
    Register = (Register << 8)
      | next_augmessage_byte;
    Byte tempValue = Top XOR message byte just
      ORed into the register;
    Register = Register XOR
      precomputed_table[tempValue];
  End
```

Figure 5 Fast table algorithm pseudo-code

An implementation of this CRC32 algorithm in C is shown in Fig. 6.

```
register = 0;
while(messagelength-->0)
{
  theRegister = (theRegister << 8) ^
    table[(theRegister >> 24) ^ *messageptr++];
}
```

Figure 6 Fast table algorithm in C

Most modern implementations use this version, because it is faster than the two previous versions shown. It is called the direct lookup table or fast table method. According to [Barr03], the bit by bit CRC algorithm requires 185 instructions per byte of message data, whereas the faster byte by byte algorithm only requires 36 instructions per byte of message data.

3. How to reverse engineer the CRC

A CRC-16 will be used to illustrate the method to reverse engineer the CRC. Using CRC-16, there are a possibility of 216 = 65,536 different codes. One way to create messages that have the same CRC but different data is to change the data, and then find a sequence of bytes that will force the CRC to be equal to the CRC of the original data. It requires n bits to change a CRC to any value, where n is the size of the generator polynomial. So to change a CRC-16 value, two bytes or 16 bits are required. The basic steps to change a message but maintaining the same CRC are as follows [4]:

1. Calculate the CRC of the original message. Save this value as (CRC_Before).
2. Calculate the CRC of the new message. Save this value as (CRC_New)
3. Reverse the CRC algorithm using CRC_Before & CRC_New to calculate the 2 bytes that will change CRC_New to CRC_Before.

Using this method, two extra bytes will have to be appended to the end of the new message to make the new message have the same CRC as the old message. If only part of the message is different between the original and new message and there are two bytes that can be used that are before the end of the message, they can be modified to maintain the same CRC with the same message length. Below is an example on how to modify bytes to maintain the same CRC.

3.1. Padding

CRC-16 will be used to illustrate how to calculate the modified bytes. To calculate the two bytes that change CRC_New to CRC_Before, suppose the two unknown bytes are X and Y. Using the direct CRC table method previously described, the bytes X and Y are processed with the CRC starting at CRC_New.

3.1.1. Definitions

```
a = starting CRC (CRC_New) (16 bits)
b = value from CRC table (16 bits)
c = value from CRC table (16 bits)
d = ending CRC (CRC_Before) (16 bits)

aH = Most Significant Byte of a
aL = Least Significant Byte of a
```

The same format is used for b, c, and d.

```
table = the precomputed CRC lookup table as
        described in the table CRC algorithms
index = an index into the table, that is
        table[index] = precomputed value

a << 8 means  a is shifted left by 8 bits
```

Curly braces are used to show the most significant bit on the left (a1) and the least significant bit on the right.

```
a = {aH} {aL}
```

3.1.2. Method

Remember the direct table lookup method for calculating CRCs is as follows:

```
n = number of bytes in the CRC

          Iterate for n
Step 1: top = MSB(CRC)
Step 2: index = top XOR message[i]
Step 3: CRC = ( CRC << 8 ) XOR table[index]
```

Now lets process two unknown bytes X and Y using the direct table method described above. The three steps of the method are done for each byte.

```
Step 1: top = MSB(a) = aH
Step 2: index = top XOR X = a1 XOR X = index(b)
        table[index] = b
Step 3: CRC = (a<<8) XOR table[index] =
          {aL XOR bH} {00 XOR bL}
Step 4: top = MSB(CRC) = (aL XOR bH)
Step 5: index = top XOR Y = (aL XOR bH) XOR Y =
          (aL XOR bH XOR Y) = index(c)
        table[index] = c
Step 6: CRC = {bL XOR cH} {00 XOR cL} = d
```

Now solve the equations backwards from step 6 to step 1.

```
From Step 6:   cL = dL
               bL = dH XOR cH
From Step 5:   Y = aL XOR bH XOR index(c)
From Step 2:   X = aH XOR index(b)
```

So d0 is the LSB(CRC_Before), and cL = dH. Now, index(c) and cH can be found by searching the table looking for the entry whose LSB = cL. Next, bL is found by solving the second equation. So, bL = dH XOR cH, which are both known. Again to find bH and index(b) are found by searching the table until the entry that has its LSB = bL. Now, Y is found by the equation Y = aL XOR bH XOR index(c). Finally, X is found by X = aH XOR index(b). So now that X and Y are known, if a message has a CRC = CRC_New and the bytes X and Y are added to the end of the message, the CRC will now be the same as CRC_Before.

To be able to do the algorithm described above, the exact implementation of the CRC must be known. Some CRC implementations initialize the CRC register with a value other than zero. The polynomial used by the CRC must also be known. If everything is known about the algorithm, then the data of the message can be changed and still produce the same CRC value as long as there are n bytes after the significant data that can be set to correct the CRC value, where n is the width of the polynomial.

3.2. Maintaining the same message length

The location of the modifier bytes does not have to be at the end of the message. The bytes can be placed anywhere after the last position where the message was changed and the end of the file. Therefore, if the modifier bytes are placed before the end of the file, the file length is maintained. In order to place the modifier bytes at a particular place in the message, the method shown in Fig. 7 is performed.

```
1. Calculate the CRC up to the position where
the data will be changed, save as CRC_A.
2. Continue calculating until the position where
the modifier bytes will be placed, save as
CRC_B.
3. Start with CRC_A and calculate the CRC of the
changed bytes up to but not including the
position of the modifier bytes, save as CRC_C.
4. Now calculate the modifier bytes as described
before with the original CRC being CRC_B and the
new CRC being CRC_C.
```

Figure 7 Method to maintain message length

Since many integrity and authentication methods verify that the length of the sent message is equal to the length of the received message, the method described above still permits an attacker to modify a specified byte and place the modifier bytes somewhere in the message to maintain the same CRC and message length. The method above was implemented and can easily change a specified byte and put the additional modifier bytes between the modified byte and the end of the file. The implemented code uses the popular CRC-32.

4. Significance

So why is the ability to create different sets of data with the same CRC significant? Because an attacker could intentionally alter data and it would be undetectable if a CRC is used as the detection method. Many programs use a CRC to check for errors such as bios updates, pkzip (Compression Software), png (Portable Network Graphics) and other software available on the Internet. Also, CRC is commonly used for error detection in communication protocols, such as Ethernet and ATM [8]. The CRC is popular for hardware implementations, because it has good random error detection properties and is easily implemented [2]. An attacker could edit a program and then fix the CRC so it appears to be a valid archive and use it to spread computer viruses or Trojan programs that allow the attacker to break into computer systems. Suppose a bank uses a CRC to verify the integrity of their database transaction and the transaction below is executed.

```
       UPDATE Account SET Balance='100' WHERE
Name=Customer AND ...;
```

Suppose someone changes the 100 to 100000 as shown below, and then fixes the CRC.

```
       UPDATE Account SET Balance='100000' WHERE
Name=Customer AND ...XY;
```

As long as the extra bytes X and Y can be added to the message, the CRC will be the same. Thus, the use of a CRC with a known generator polynomial and initial value in the register is not suggested when an adversary can intercept and alter the data.

5. Conclusion

It has been shown that cyclic redundancy checks with known generator polynomials should not be used to verify the integrity when someone can alter the data between its source and destination and can obtain the original CRC values. Two solutions are to keep the polynomial used to create the CRC secret [9] or to incorporate some secret value into the CRC that is unknown to the adversary as in HMAC-SHA1 or HMAC-MD5 [10]. A hash function such as MD5 or SHA1 may be a better choice to verify integrity of data. One-way hashes are mathematical functions that produce a relatively unique value for a particular message. So unlike the CRC algorithm, the one-way hash algorithms are designed such they cannot be mathematically reversed. A brute force method would be needed to attempt to attack these hashes, so they are currently the better choice to ensure data integrity unless a secret value is incorporated into the CRC. Many Unix archives are now providing MD5 sums to validate the integrity of archives downloaded off of the Internet. But currently Windows machines do not include an MD5 program.

6. Acknowledgments

The authors would like to thank Logical Dynamics, Inc., for the opportunity to collaborate on the message authentication research. The material is based upon work supported by the National Science Foundation under Grant No. 0090596.

7. References

[1] S. Lin, D. J. Costello, Jr., *Error Control Coding: Fundamentals and Applications*, Prentice Hall, 1982.

[2] D. C. Feldmeier, "Fast Software Implementation of Error Detection Codes", *IEEE/ACM Transactions on Networking*, vol. 3, no. 6, December 1995, pp. 640-651.

[3] N. Borisov, I. Goldberg, and D. Wagner, "Intercepting mobile communications: the insecurity of 802.11", *in Proceedings of the Seventh International Conference on Mobile Computing and Networking*, pp. 180-189, Rome, Italy, 2001.

[4] Anarchriz (anarchriz@hotmail.com), "CRC and how to reverse it," April 30, 1999, http://its.mine.nu/html/re/essays/CRC.html.

[5] R. Williams, Rocksoft Pty Ltd of Austrailia, "A Painless Guide To CRC Error Detection Algorithms", August 19, 1993, ftp://ftp.rocksoft.com/papers/crc_v3.txt.

[6] Dilip V. Sarwate, "Computation of Cyclic Redundancy Checks via Table Look-Up", *Communications of the ACM*, August 1988, vol. 31, no. 8, pp. 1008-1013.
[7] T.V. Ramabadran, S.S. Gaitonde, "A Tutorial on CRC Computations", *Micro, IEEE*, vol. 8, no. 4 , Aug 1988, pp. 62-75.
[8] T. Henriksson, D. Liu, "Implementation of Fast CRC Calculation", *in Proceedings of Design Automation Conference*, 2003, pp. 563-564.
[9] H. Krawczyk, "LFSR-based hashing and authentication," *in Proceedings of CRYPTO '94*, Lecture notes in Computer Science, vol. 839, Aug. 1994, New York: Springer-Verlag, pp. 129-139.
[10] H. Krawczyk M. Bellare, and R. Canetti, "HMAC: keyed-hashing for message authentication," RFC 2104, Feb. 1997.