Cyclic Redundancy Check

John McCall

October 14, 2013

1 Introduction

A *cyclic redundancy check (CRC)* is a type of error-detecting code used to detect accidental changes in data. To detect an error, a fixed length *check value* is appended to the end of the message. The check value is based on the remainder of a polynomial division of the messages contents. When the message is received the check value is recomputed and the appropriate measures to fix the corruption can be taken if the two values do not match.

CRCs are commonly used because they are easy to implement in binary hardware, easy to analyze, and are quite good at detecting burst errors, continuous series of incorrect data in a message. This is particularly important because burst errors are common in several types of communication channels. Usually, an n-bit CRC used on an arbitrary length message will detect any single burst error less than or equal to n bits and will detect a fraction $1 - 2^{-n}$ of all longer burst errors.

The specification of a CRC code requires a *generator polynomial.* This polynomial is the divisor in polynomial long division, the dividend is the message, the quotient is thrown away, and the remainder becomes the result. It is important to note however, that the coefficients of the polynomial are computed using the carry-less arithmetic of a finite field. Most CRCs use the finite field GF(2) in practice as this field only has two elements, 0 and 1, which meshes nicely with the architecture of computers.

A CRC is called an n-bit CRC when its check value has n bits. Any n has multiple CRCs, each with a different polynomial. Any such polynomial has highest degree n, which means that it has $n + 1$ terms, and that its encoding requires $n + 1$ bits. A CRC and associated polynomial are given a name of the form CRC-n-XXX. The simplest CRC is CRC-1 and it uses the generator polynomial $x + 1$. This is also called a parity bit, which is the simplest form of error detection.

2 Application and Integrity

A device that implements a CRC calculates the check value for each data block to be sent or stored and appends it to the end of the block. This forms a *codeword.*

When a codeword is read or received the device does one of two things. It either recalculates the check value of the data block and compares it to the check value from the codeword, or it performs a CRC on the entire codeword and compares the resulting check value with an expected *residue* constant. If the check values do not match the block contains some errors in its data and the device may take corrective actions. However, because of the nature or error-checking, it is possible that some errors may remain undetected, even if the check values match.

CRCs are designed to protect accidental errors incurred from communication channels, and can give assurance of the integrity of delivered messages. CRCs are not suitable when it comes to protecting against intentional modification of data for several reasons. There is no authentication so an attacker can edit the message and recompute the CRC without being detected. CRC is easily reversible which makes it unsuitable for use in digital signatures. Finally, CRC is a linear function with a property that $crc(x \oplus y) = crc(x) \oplus crc(y)$. Because of this, even if the CRC is encrypted with a stream cipher, the message and its CRC can be altered without knowledge of the encryption key.

3 Computing a CRC

To compute an n-bit binary CRC, start with the message encoded in binary:

```
11010011101100
```

This is padded with n zeroes, corresponding to the bit length n of the CRC. The first calculation for a 3-bit CRC is as follows:

```
11010011101100 000 <--- input padded with 3 bits
1011               <--- divisor (4 bits)
------------------
01100011101100 000
```

If the input bit above the leftmost divisor bit is 0, do nothing. If the bit is 1, the divisor is XORed with the input. The divisor is then shifted one bit to the right. This process is repeated until the divisor reaches the right most bit of the input. The calculation in its entirety is:

```
11010011101100 000 <--- input padded with 3 bits
1011               <--- divisor
01100011101100 000 <--- result
 1011              <--- divisor ...
00111011101100 000
  1011
00010111101100 000
   1011
00000001101100 000
       1011
```

```
00000000110100 000
        1011
00000000011000 000
         1011
00000000001110 000
          1011
00000000000101 000
           101 1
------------------
00000000000000 100 <--- remainder (3 bits)
```

In this case the leftmost bit in the divisor zeroes out all the bits in the input so all that's left is the remainder. To check the validity of the received message simply perform the computation again, but instead of padding the input with zeroes, add the check value instead. If there are no errors the remainder after this computation should be zero.

```
11010011101100 100 <--- input with check value
1011               <--- divisor
01100011101100 100 <--- result
 1011              <--- divisor ...
00111011101100 100

......

00000000001110 100
          1011
00000000000101 100
           101 1
------------------
                 0 <--- remainder
```