Senior Seminar Proposal: Error Correcting Codes

John McCall
Adviser: Peter Dolan

December 11, 2013

1 Introduction

As technology progresses, we rely much more on electronic means of communication. This communication comes in many forms, such as sending emails or playing music from a CD. However, this communication is not perfect and some of the data may be lost or corrupted in transmission. There are many potential sources of this corruption, for instance a CD may become scratched. It is an unrealistic goal to try to prevent all corruption from occurring.

For this reason error correction codes have been developed. Error correction codes are designed to fix corruptions rather than prevent them. A CD with minor scratches can still play music perfectly. This is because of an error correction code, which extrapolates the data corrupted by the scratch based on the uncorrupted data.

One specific error correction code is the Reed-Solomon code. Reed-Solomon codes are used in many applications. From consumer electronics such as CDs and DVDs, to deep-space transmissions on the Voyager space probe. The goal of this senior seminar is to understand how and why Reed-Solomon codes work.

2 Current Work

Currently, we have formally defined the concept of a *ring*, a *group*, and a *field*. We have also explored the error correction methods of *cyclic redundancy checks* and the *Luhn algorithm*. In addition to this, we have started working towards understanding Reed-Solomon codes.

2.1 Cyclic Redundancy Check

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.

A CRC is called an n-bit CRC when its check value has n bits. Any n has multiple CRCs, each with a different generator 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.

CRCs are designed to give protection from 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.

2.1.1 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
```

2.2 Luhn Algorithm

The Luhn algorithm, which is also called the modulus 10 algorithm, implements a simple checksum formula and is used in validating identification numbers. Credit card numbers, IMEI numbers, National Provider Identifier numbers, and Canadian Social Insurance Numbers are a few examples of the types of ids that the Luhn algorithm is used with. It was created by Hans Peter Luhn, a scientist working for IBM, in 1954.

This algorithm is meant to detect accidental errors, as opposed to malicious attacks, so by design it is not a cryptographically secure function. It is mostly used as an easy way to distinguish valid numbers from a series of random digits. The Luhn algorithm will catch any single-digit error and most cases of switching adjacent digits. However, it will not detect 09 being switched to 90 (or vise versa). It will detect 7 of the 10 possible twin errors (it will not detect: $22 \leftrightarrow 55, 33 \leftrightarrow 66, 44 \leftrightarrow 77$).

2.2.1 The Algorithm

The Luhn Algorithm verifies a numbers using its check digit, which is usually added to a partial account number to generate the whole number. There is a test that the account number must pass:

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit; if the product of this doubling operation is greater that 9(e.g., $7 * 2 = 14$), then sum the digits of the products (e.g., 10: $1 + 0 = 1$, 14: $1 + 4 = 5$).
2. Take the sum of all the digits.
3. If the total modulo 10 is equal to 0 (if the total ends in zero) then the number is valid according to the Luhn formula; else it is not valid.

2.2.2 Example

Take the account number 7992739871, we will add a check digit, making the number: 7992739871x.

We obtain x by computing the sum of the digits and then multiplying that value by 9 modulo 10. In this case, the sum of the digits is 67. 67 times 9 equals 603, modulo 10 gives us 3, which is the check digit.

Using Luhn's Algorithm on the whole account number (79927398713) should result in a 0, meaning that the account number is valid. Alternately, if you already know the check digit, one can do the computation to determine the check digit and compare that result to the known digit. If they match the account number is valid, otherwise it's invalid.

3 Future Plans

The plan for the future is to continue studying Reed-Solomon codes. We hope to better understand how Reed-Solomon codes work by studying the generalized Reed-Solomon codes presented by J.I. Hall in *Notes on Coding Theory*. In the end we hope to have an in depth knowledge of how Reed-Solomon codes function, why they work, what potential weaknesses they have, and if there are any algorithms which are comparable to it.

4 References

1. I. S. Reed and G. Solomon, *Polynomial Codes Over Certain Finite Fields*, Journal of the Society for Industrial and Applied Mathematics Vol. 8, No. 2 (Jun., 1960), pp. 300-304 `http://www.jstor.org/discover/10.2307/2098968?uid=3739736&uid=2&uid=4&uid=3739256&sid=21103179919113`

2. J.I. Hall, *Notes on Coding Theory*, Department of Mathematics, Michigan State University, East Lansing, MI 48824 USA http://www.mth.msu.edu/~jhall/classes/codenotes/coding-notes.html

3. http://www.cs.cornell.edu/courses/cs722/2000sp/reedsolomon.pdf