Cryptography Report

Tom Goring

January 18, 2021

1 Abstract

The purpose of this report is to detail what has been implemented in order to complete the coursework for this module. The topics covered during the coursework include:

- Credit card verification
- ISBN verification
- BCH generation and correcting
- Password hashing and cracking
- Text ciphering and steganography

A GUI has been implemented in React which communicates with the Rust crypto library I have created over a REST API. The REST API can be used from any consumer - though I have not documented it so you would have to read the code to know what endpoints to use. The unit tests can be run by executing the command 'cargo test' from within the directory with 'cargo.toml' present. You will need OpenCL.lib within the target/debug/deps directory in order for the project to compile.

Contents

2 Credit card and ISBN verification

2.1 Credit Card Numbers

The requirement for this section of the coursework was to receive a 16 digit credit-card number and validate that it was numerically valid. The algorithm used to determine the correctness of credit card numbers is the Luhn Algorithm:

$$\left(\sum_{i=0}^{k} 2n_{2i} + \sum_{i=0}^{k} n_{2i+1}\right) \bmod 10 = 0$$

In plain English steps:

1. Starting from the rightmost digit (excluding the check digit) and moving left, double the value of every second digit.
2. For each doubled digit, if the result is greater than 9 then add the digits of the result or subtract 9. In the submitted code I modulo the digit by 9 as this has the same effect for the digits 10-19 ($x \bmod 9 \equiv x - 9$)
3. Take the sum of all new digits including the check.
4. If the modulo of this sum is equal to 0, the number is valid according to the algorithm.

I have implemented the provided test cases as unit tests with a 100% pass rate.

2.2 ISBN Numbers

The requirement for this section of the coursework was to receive a 10 digit ISBN code and validate that it was numerically valid. The algorithm used to determine the correctness of a 10-digit ISBN number is as follows:

$$\sum_{i=1}^{10} ix_i \equiv 0 \pmod{11}$$

In English: the sum of the digits multiplied by their position under modulo 11 must equal 0 for the ISBN to be valid.
Again, the provided test cases were converted to unit tests with a 100% pass rate.

3 BCH Generation and Correction

The task for this section was split into two parts:

1. Given six digits, generate 4 checking digits. If any of the checking digits are 10, reject the input and print a warning.
2. Given ten digits, perform one of three actions depending on its validity:
   (a) If there is no error, output this.
   (b) If there is a single or double error, correct them and output the corrected codeword
   (c) If there are more than two errors, output this.

3.1 Generating check digits

Using a generator matrix created via matrix operations on a Vandermonde Matrix (the derivation of which I will not detail here for brevity):

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 4 & 7 & 9 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 10 & 8 & 1 & 2 \\ 0 & 0 & 1 & 0 & 0 & 0 & 9 & 7 & 7 & 9 \\ 0 & 0 & 0 & 1 & 0 & 0 & 2 & 1 & 8 & 10 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 9 & 7 & 4 \\ 0 & 0 & 0 & 0 & 0 & 1 & 7 & 6 & 7 & 1 \end{bmatrix}$$

$$\begin{aligned} d_7 &= (4d_1 + 10d_2 + 9d_3 + 2d_4 + d_5 + 7d_6) \pmod{11} \\ d_8 &= (7d_1 + 8d_2 + 7d_3 + 1d_4 + 9d_5 + 6d_6) \pmod{11} \\ d_9 &= (9d_1 + 1d_2 + 7d_3 + 8d_4 + 7d_5 + 7d_6) \pmod{11} \\ d_{10} &= (d_1 + 2d_2 + 9d_3 + 10d_4 + 4d_5 + 1d_6) \pmod{11} \end{aligned}$$

All provided tests are implemented as unit tests and pass with a 100% accuracy.

3.2 Detecting and correcting errors

In order to decode a BCH(10, 6) code, one must first generate 4 syndrome digits:

$$s_1 = (d_1 + d_2 + d_3 + d_4 + d_5 + d_6 + d_7 + d_8 + d_9 + d_{10}) \pmod{11}$$
$$s_2 = (d_1 + 2d_2 + 3d_3 + 4d_4 + 5d_5 + 6d_6 + 7d_7 + 8d_8 + 9d_9 + 10d_{10}) \pmod{11}$$
$$s_3 = (d_1 + 2^2d_2 + 3^2d_3 + 4^2d_4 + 5^2d_5 + 6^2d_6 + 7^2d_7 + 8^2d_8 + 9^2d_9 + 10^2d_{10}) \pmod{11}$$
$$s_4 = (d_1 + d_2 + d_3 + d_4 + d_5 + d_6 + d_7 + d_8 + d_9) \pmod{11}$$

If $(s_1, s_2, s_3, s_4) \equiv (0, 0, 0, 0)$, then there are no errors present in the given code, else we can calculate P, Q, and R values:

$$\begin{aligned} P &= S_2^2 - S_1 - S_3 \\ Q &= S_1S_4 - S_2S_3 \\ R &= S_3^2 - S_2S_4 \end{aligned}$$

If $(P, Q, R) \equiv (0, 0, 0)$, there is a single error present, the magnitude and location of which is locatable with:

$$error_{mag} = s_1$$
$$error_{pos} = \frac{s_2}{error_{mag}}$$

If $(P, Q, R) \not\equiv (0, 0, 0)$, there are two or more errors present. To calculate the positions and magnitudes of the errors:

$$i = err1_{pos} \qquad j = err2_{pos}$$
$$a = err1_{mag} \qquad b = err2_{mag}$$

$$i = \frac{-Q + \sqrt{Q^2 - 4PR}}{2P} \qquad j = \frac{-Q - \sqrt{Q^2 - 4PR}}{2P}$$

$$b = \frac{i * s_1 - s_2}{i - j} \qquad a = s_1 - b$$

If $\sqrt{Q^2 - 4PR}$ doesn't have a valid root under mod 11, or i or j is 0, or a digit is corrected into 10 there are at least three errors.
All test cases were implemented as unit tests with a 100% pass rate.

4 Password Hashing and Cracking

In this section the requirement was to first hash (using a library) a single string with the SHA1 algorithm, and then create a brute force password cracker to crack a six digit length password using an alphabet of [0-9, a-z].

4.1 Hashing

I utilised the Rust implementation of the SHA1 algorithm to do the hashing of passwords in the CPU version. The following code hashes a string:

```rust
use sha1::Sha1;

pub fn sha1(input: &str) -> String {
    let mut hasher = Sha1::new();
    hasher.update(input.as_bytes());
    hasher.digest().to_string()
}
```

4.2 Cracking

For this section, I wrote two implementations of a password cracker, one utilising CPU hashing, and one utilising GPU hashing.

In order to enumerate all possible passwords up to the given length, I am using a form of permutation ranking. Every password is given a list of indices, where each element represents an index into the alphabet. An example of this would be:

$$\begin{aligned}
alphabet &= a, b, c \\
a &= [-1, -1, 0] \\
b &= [-1, -1, 1] \\
c &= [-1, -1, 2] \\
aa &= [-1, 0, 0]
\end{aligned}$$

This allows me to iterate through all possible permutations (with repetitions) by doing a binary addition with carry on these indices.

I implemented two methods of cracking: one using CPU threads, and one using GPU threads (via OpenCL).

4.2.1 CPU Performance

On the CPU, the program utilizes all available cores to permute through passwords. Each thread has its own indices which it steps through at num-cpu-cores at a time. This means on my 32 logical thread processor, the program starts with 32 sets of indices, and each is stepped upwards by 32 at a time.

The performance here is not particularly good, the worst case running time of this program is if the given hash is of the last possible password, which in this case is '999999'. The formula for calculating the total number of passwords up to a given length can be given by:

$$\sum_{i=1}^{k} n^i$$

Where k = password length and n = alphabet size.

Plugged into our requirements, this means a total of 2,238,976,116 passwords - over two billion in total!.

When attempting to crack the hash of '999999', the CPU version of my code achieves this in approximately 1100 seconds:

```
Testing started at 21:51 ...
    Finished test [unoptimized + debuginfo] target(s) in 0.25s
     Running target\debug\deps\crypto-ab5f994e8d50b6c3.exe
Time to enumerate all passwords on CPU: 1106.8557s
```

This represents a speed of approximately 2 million hashes per second across 32 threads - about sixty thousand per second on a single core. I felt almost 20 minutes to crack all possible passwords was slow, so I implemented a GPU based cracker as well:

4.2.2 GPU Performance

In order to run code on a GPU, I used a combination of OpenCL[4] and a Rust crate allowing the control of GPU kernels from a Rust program. I copied an implementation of the SHA1 algorithm[1] and made some changes to make it work in an OpenCL context.

Due to an unresolved bug, the GPU version can only crack up to a maximum string of '8rshel', however this is close enough to the maximum that it should not change the benchmarks too much.

When run on a GPU, the results are far more desirable:

```
Testing started at 17:16 ...
    Finished test [unoptimized + debuginfo] target(s) in 0.24s
     Running target\debug\deps\crypto-ab5f994e8d50b6c3.exe
Time to enumerate all passwords on GPU: 4.8137026s
```

The results from the given hashes are as follows:

```
Hash c2543fff3bfa6f144c2f06a7de6cd10c0b650cae = this
Hash b47f363e2b430c0647f14deea3eced9b0ef300ce = is
Hash e74295bfc2ed0b52d40073e8ebad555100df1380 = very
Hash 0f7d0d088b6ea936fb25b477722d734706fe8b40 = simple
Hash 77cfc481d3e76b543daf39e7f9bf86be2e664959 = fail7
Hash 5cc48a1da13ad8cef1f5fad70ead8362aabc68a1 = 5you5
Hash 4bcc3a95bdd9a11b28883290b03086e82af90212 = 3crack
Hash 7302ba343c5ef19004df7489794a0adaee68d285 = 1you1
Hash 21e7133508c40bbdf2be8a7bdc35b7de0b618ae4 = 00if00
Hash 6ef80072f39071d4118a6e7890e209d4dd07e504 = cannot
Hash 02285af8f969dc5c7b12be72fbce858997afe80a = 4this4
Hash 57864da96344366865dd7cade69467d811a7961b = 6will
Time to crack all given hashes: 4.323484s
```

These results indicate the GPU version is cracking around 520 million hashes a second - 260 times as many as the CPU version!

5 Stream Cipher and Steganography

The requirement for the final task was to implement a messaging system capable of hiding a stream-cipher encrypted message inside another message using steganography. This section can therefore be broken down into two parts - the encryption using a stream cipher, and the hiding of the encrypted using message steganography.

5.1 Stream Cipher

I have implemented a simple stream cipher using the Rust 'rand' crate. A PRNG is seeded using the given key (pre-known by both parties), and a byte array is filled using the created values. This array is equal in length to the input byte array to be encrypted. Each byte of the target array is then XOR'd with each byte in the keystream until all bytes are encrypted. To unencrypt, one must simply run the outputted byte array back through the cipher with the same key used to encrypt it.

5.2 Steganography

There are a few different kinds of text steganography, most of which involve somehow editing the actual carrier message content to encode the hidden message. I have chosen a method of hiding the message that does not involve visibly changing the content of the message: zero-width characters. Under unicode encoding, there are certain characters belonging to non-english alphabets that are not visible when the text is rendered normally. I have selected three characters to utilise for this purpose - zero width spaces (U+200B), zero width non-joiners (U+200C), and zero width joiners (U+200D).

Using (U+200B) and (U+200C) as equivalents of binary 0 and 1, the algorithm encodes each byte of the encrypted message into its binary format. This means that a number like 1 becomes:

(U+200C) (U+200C) (U+200C) (U+200C) (U+200C) (U+200C) (U+200C) (U+200B)

Or in binary form: 0b00000001

These 'binary' strings are appended after every character in the carrier message; if there are not enough characters in the carrier, they are added using (U+200D) as a replacement for null string terminators.

When rendered normally in a unicode compliant format - these characters are invisible. Of course, when opened in a more basic editor, such as one using ASCII only, the characters will render as broken boxes, however the purpose of this camouflage is merely to divert a casual eye (who is probably looking at it using a browser based technology), not a determined investigator - the hidden message is still encrypted until someone with the right key decrypts it.

References

[1] Conte, Brad. *crypto-algorithms.* Available from: `https://github.com/B-Con/crypto-algorithms` [Accessed January 18, 2021]

[2] Misawa Kei. *Zero-Width Unicode Character Steganography.* Available from: `https://330k.github.io/misc_tools/unicode_steganography.html` [Accessed January 18, 2021]

[3] Hoid. *Use Zero-Width Characters to Hide Secret Messages in Text.* Available from: `https://null-byte.wonderhowto.com/how-to/use-zero-width-characters-hide-secret-messages-text-even-reveal-leaks-0198692/` [Accessed January 18, 2021]

[4] Khronos Group. *OpenCL.* Available from: `https://www.khronos.org/opencl/` [Accessed January 18, 2021]