# Cryptography Report

Christopher Thomas Edmunds - 17024395

Contents

[Cryptography Report 1](#_Toc94154116)

[Task 1 2](#_Toc94154117)

[Task 2 3](#_Toc94154118)

[Task 3 5](#_Toc94154119)

[Task 4 6](#_Toc94154120)

[Sources 11](#_Toc94154121)

## Task 1

The first task we were assigned on this module was verification of credit card numbers. Credit card numbers are generated based on an algorithm known as “Luhn’s Algorithm”

Luhns algorithm is a specific algorithmic formula used for the detection of accidental errors within credit card numbers, it is contingent on the generation of a “Check Digit” as the last digit of a credit card number, which the algorithm can use to produce a valid result, only if the check digit is correct relative to the rest of the numbers in the credit card. It is worth noting that the algorithm was specifically aimed at recognizing accidental errors and is not secure against malicious attacks

The check digit of a credit card can be calculated using a relatively simple method

Say for example we use the (sans check digit) credit card number

400360000000001

The first step is to take, starting from the first digit, every second digit and double it

**4**0**0**3**6**0**0**0**0**0**0**0**0**0**1**

X2

8-0-12-0-0-0-0-2

However, because we’re doubling following a modulo 10 method, “Mod 10 algorithm” being a colloquial name for this method, numbers that roll over 10 will begin counting from 1. So, where we have one number that equals 12, this will become 1+2 =3

8-0-3-0-0-0-0-2

We now add together these values

8-0-3-0-0-0-0-2 = 13

Now we take all the digits that **weren’t** doubled. And add them together

**4**0**0**3**6**0**0**0**0**0**0**0**0**0**1**4

0-3-0-0-0-0-0 = 3

And if we add together the sum total of these digits we get, 16, which we run through the following formula

((10 – 16 mod10) mod10) = 4

4 can now be used as the check digit at the end of our credit card number forming

4003600000000014

Any corruption of a single digit that happens during transmission, will cause the derived check digit and the transmitted check digit to be different from one another, indicating that corruption has occurred.

However, the Luhn’s algorithm, as valuable as it is, is limited by its inability to detect more than single digit transcription errors, or transposition errors wherein digits are moved left or right more than 1 space, or errors where two digits are swapped. It will detect some twin swap errors \*but not all of them\*

Ultimately I would argue this method is efficient for single small strings of numbers, but lacks the redundancy for more complicated strings of numbers as it is incapable of recognizing enough errors to be practical for large amounts, and since the likelihood of errors raises with the number of digits, and it can possibly fail if there are multiple errors, the larger the string of digits the more likely the algorithm is to fail making its application very specific.

Because of this, various other algorithms have been developed that I’ve read about, to handle multi digit errors, more consistently recognize transposition errors, and even handle alphanumeric. I will not go into greater detail about how these work but some examples include

Luhn mod N algorithm

Verhoeff algorithm

and the Damm algorithm

## Task 2

The second task we were assigned was to generate, then verify, BCH(10,6) codes.

BCH codes (short for Bose, Chaudhuri, and Hocquenghem) are another, more comprehensive, form of error correction, it is a category of multiple types of similar error correcting encryption. They are used in a wide variety of applications from basic communication, through to data storage on DVD’s, HDD’s and SSDs, and more experimentally Quantum Computing Resistant Cryptography (a very interesting field). Broadly speaking they are a generalization of “Hamming Codes”, another error correction method which we also studied.

BCH codes typically take 3 parameters, not including the original data itself. The block length, the number of parity checking digit, and the minimum distance

BCH codes specifically use a Vandermonde Matrix for parity checking, in the case of BCH 10,6 the matrix will be 10 entries wide, making it usable for MOD11 calculations

From the Vandermonde matrix we can derive the Generator Matrix, which is 10 entries wide and 6 entries tall, hence BCH 10,6

Ultimately this will allow us to support a string 6 digits long, and generate 4 parity codes that can be used to verify the integrity of these digits once they have been read or transmitted

Below is an example generator matrix

Text

Description automatically generated

We can generate our four parity codes by multiplying the digits of our original integer to each of the values vertically, then adding them together (digit1\*4 + digit2\*10 + digit3\*9 + digit4 \*2 + digit5\*1 + digit6\*7) and then modulo 11 for the end value. This will generate the value 0. We do this for the entire string, once for each column.

It is possible for this to generate the value “10” however, so in this situation we simply end up with an additional parity digit, which given the fixed length of the code (in this situation) is valid. However other methodologies will handle this differently, some will use a single character as a stand in such as X for ISBN for example.

Once we’ve completed this step, we’ve functionally created our BCH10,6 code. The four parity digits provide much more redundancy for error correction than the single digit present in Luhn’s algorithm. Where Luhn’s can only handle a single error, BCH10,6 can handle 1, 2, or recognize when there is 3 or more. While it can recognize the position and distance from the original value up to two errors are, it can only recognize when more than 3 errors have occurred, not their position or how much they are off by.

If we want to attempt to decode our BCH code, we need to generate something called syndromes using our parity checking matrix. For the sake of the markers sanity, I will refrain from regurgitating the entirety of these formulas, but suffice to say the steps can be summarized as such

Generate four syndromes, if all four syndromes == 0 then no errors present

Calculate the following values

P = S2squared - S1 S3

Q = S1 S4 - S2 S3

R = S3squared - S2 S4

If all of these values = 0 there is 1 error

If any of them are not 0, then there are 2 errors

The final check is this formula (*Q2-4\*P\*R)*

If this formula doesn’t produce a square root under mod11 then there are more than 2 errors.

I obviously excluded a lot in this, for the sake of brevity. But I feel this gives a broad sense of the required inputs, uses and limitations of BCH error codes.

There is a significant advantage to BCH codes, in that other codes can be written that can handle progressively more and more errors. This makes them practical for all sorts of applications from small amounts of data to validating extremely large amounts of data, although additional techniques are often applied to make these different data styles yet more practical and resilient.

Similar applications such as turbo codes and LDPC codes exist for use in other mediums with different requirements

## Task 3

The third task we were assigned, was to develop a brute force algorithm for cracking SHA1 encrypted passwords, and a similar brute force algorithm for cracking BCH10,6 codes.

In both applications the overarching design is broadly similar for a brute force algorithm. You generate every permutation possibly of potential decrypted results, encrypt them, and see if the encrypted result of your permutation and the encrypted input you are trying to crack match. If they match, then your random permutation is the value held in the encrypted input you’re trying to crack.

The only distinction between the two algorithms is, trying every permutation of normal passwords up to 6 characters, and trying every permutation of BCH10,6 codes

The reason this method needs to be used is because hashing functions such as SHA1 are typically one-way functions, they cannot be reversed. This means that there is functionally no way to find out their value without simply trying every permutation until you generate the same one. Although that said, many hashing algorithms have been cracked including SHA1, and are no longer securely one-way functions, hence its depreciation as a widely used hashing algorithm.

The downside of this method of course, is that the time constraint for attacking a password this way grows exponentially. To demonstrate this, I’ll show a formula below for a 3-char password

A: Password length

B: Dictionary Size

B^A + B^A-1 + B^A-2

As you can see, the increase per additional character in a password is exponential. On a 6-character alphanumeric password, using a fast optimized program that can try one hundred billion attempts a second (totally viable on modern hardware), this can be cracked in 0.003 seconds.

Comparatively an 11-char alphanumeric would take 2.24 weeks.

12 = 1.55 years

13 = 55.79 years

And 14 = 20.08 centuries.

Within big o notation this would be described as having O(mn) time complexity.

From this we can see clearly how rapidly the time to brute force a password balloons outside of the realms of practicality for solving in this way. My own code was able to crack a 6 char alphanumeric (in theory alphanumeric, the actual string I used was purely lowercase alphabetical but it makes no difference in this case) Text

Description automatically generated with medium confidence

In the above amount of time.

This is interesting when compared to the time taken by a rainbow table, which I will show later. But on an individual level, this is relatively quick compared to the time to generate a rainbow table, a lookup table, and use them both to attempt to look up a single password. However, for a rainbow table the time to generate a rainbow table is singular, and the only requirement for repeated use of it is to generate a new lookup table, and use it, each time.

## Task 4

Our fourth and final task, was to generate a rainbow table and use it to crack passwords.

Rainbow tables are functionally, based on the same general idea as brute force password cracking algorithm. Producing every permutation of a password, hashing it, then comparing the hashed value to the hash value you’re trying to decrypt. However, Rainbow tables do many things differently to a simple brute force algorithm.

There have been multiple attempts to solve the issues inherit to brute force algorithms, each having their own problems themselves. A fun way to examine these issues that are inherit to brute force algorithms, is to examine the attempted solutions and the problems that these themselves created

The first most obvious solution is to attempt a brute force style algorithm once and store every possible iteration of string with their associated hash. There are two issues with this solution, first it doesn’t reduce time to execute as much as is desirable. While lookup times are faster than hashing algorithms, the time to execute is still very long. Secondly the storage space required to store this combination of hashes and strings grows exponentially, rapidly reaching into the terabytes by the time you have reached the 8-character numeric we were dealing with.

To combat this, the concept of precomputed hash chains was implemented. Hash chains are an intelligent way of compressing the storage space required to store all these hash string combinations. They work by taking a string, hashing it, then using a reduction function to turn that hash into a different deterministic password. This is looped for as long as you want the chain to be. It is important that this is deterministic as this process needs to be repeatable. Now that you have your chain you can discard everything but the starting chain link and the end chain link. You then generate enough chains of sufficient length to cover the approximate number of permutations that your password may consist of.

Now that you have your hash table. You can follow a similar chaining process with the hash you are attempting to decrypt. Eventually, your decryption chain will produce a chain that is identical to one of the end chains of your table. You can then repeat the chaining process on the starting chain, of the end chain that is identical to your decryption chains end chain. A screenshot of a game

Description automatically generated with low confidence

Eventually through expanding this chain. You will find the point where the two chains begin to intersect, and find the value stored within the chain that corresponds to the hash you’re trying to decrypt like so

A screenshot of a game

Description automatically generated with low confidence

This method fixes one of the fundamental problems with a table of hashes. Which is compresses the storage requirement of the table by however many links are contained within a chain. This is an example of a time/space tradeoff in password cracking.

There does however exist a problem with this method. Which is the problem of collisions.

During the creation of these chains, it is possible for you to generate an identical password to that contained in another chain, in fact the likelihood of this happening increases exponentially the more of the search space you cover. For example, A picture containing chart

Description automatically generated

To quote the article from which these graphics were derived

“As you can see, if we want to crack 95% of 4 digit md5 hashes, we need about 25,000 total hash computations. If we wanted to crack 100%, we need about 100,000 hash computations.”

An example image showing what a chain collision might look like is below

A picture containing text, sign

Description automatically generated

For every collision that occurs, the following chains are effectively useless, requiring we either balloon the storage space to account for them, or implement an extremely time intensive method of checking for these duplicates. Some applications of this technique, referred to as “imperfect hash chains” make the trade off and only cover a portion of the search space but this is less than ideal.

It is to combat this last problem, that our final solution was implemented. The reduced rainbow table.

The Rainbow Table operates broadly the same as a hash table with one key difference. Its reduction function takes an additional modifier, the current link within the chain. This modifier, due to the nature of a hashing algorithm, will mean that once the password has been reduced, depending on where in the chain it is, it will create a different hash value.

The result of this is that say, if the string 1337 exists in chain 1 at link 5, and in chain 2 at link 6, they will produce entirely different hashes. In the case of my code, once I had converted the hash value to an integer, I simply added the current chain onto it before using the reduction function on the integer. Example below

A picture containing text, sign

Description automatically generated

Now it is still technically possible for a collision to occur, but because they must occur at the same point in the chain, from that point onwards in the chain the rest of the two collided chains will be identical. This will consequentially produce an identical end chain. Which makes it trivial to remove these duplicates by discarding results with duplicate end chains.

Ultimately this means we are left with a very space efficient table, containing a minimum of duplicate results.

This does unfortunately have the side effect of meaning that we can’t simply create one long chain when looking up the password in our table. We are required to first create a lookup table, with the chain being created starting from different beginning chain points, as the end chain will be different depending on where in the chain the matching password and hash for our hash started. However, the time to create this is relatively small. An example of how this would look is below.

Diagram

Description automatically generated

The comparison of the efficiency of this solution is easy to do.

When deciding whether it is preferable to use a brute force algorithm or a rainbow table, the decision is not based on whether one system is better than the other. The choice is dependent on the volume of passwords you want to crack. Many password cracking attempts are operated on entire databases that have leaked; others are targeted towards individuals.

In the event that you wanted to crack a single password. The 1075 seconds my brute force attempt took is obviously preferable to the 24,905 seconds my rainbow table took to generate plus the 135 seconds it took to generate a lookup table and reference it.

However, a rainbow table generation and associated lookups only takes the same time to execute as 28 password attempts, after that point each individual lookup only takes 1/10th (roughly) the time to execute as a brute force algorithm. This is especially impressive when you consider that the password space the rainbow table was working with here was 2 characters larger than the brute force algorithm (I didn’t have time to take another measurement for the table sorry Rong)

As many password databases contain upwards of millions of passwords, this significant time saving becomes more than noteworthy, it becomes essential to being able to decrypt these databases.

Text

Description automatically generated

Text

Description automatically generated

## Sources

Rainbow table sources:

<https://kestas.kuliukas.com/RainbowTables/>

<https://stichintime.me/2009/04/09/rainbow-tables-part-5-chains-and-rainbow-tables/>

<https://rsheasby.medium.com/rainbow-tables-probably-arent-what-you-think-part-2-probability-efficiency-and-chain-9e2fb5e8cdc9>

Brute force sources:

<https://www.grc.com/haystack.htm>