A study of cryptography

Student No - 19016317

Contents

[A study of cryptography 1](#_Toc93341460)

[Verifying credit card numbers 3](#_Toc93341461)

[Introduction to Luhn 3](#_Toc93341462)

[Luhn algorithm summary 3](#_Toc93341463)

[BCH (10,6) Generating and Correcting 3](#_Toc93341464)

[Introduction to BCH (10,6) 3](#_Toc93341465)

[BCH (10,6) Generating 3](#_Toc93341466)

[Correcting BCH code 3](#_Toc93341467)

[Brute Force Password Cracking 4](#_Toc93341468)

[Cracking passwords with brute force 4](#_Toc93341469)

[Cracking BCH (10,6) codes with brute forcing algorithm 5](#_Toc93341470)

[Method One 5](#_Toc93341471)

[Method Two 6](#_Toc93341472)

[Brute forcing algorithms analysis 6](#_Toc93341473)

[Using Rainbow Tables to Break Passwords 6](#_Toc93341474)

[Rainbow table overview 6](#_Toc93341475)

[Building of the table 7](#_Toc93341476)

[Cracking passwords with rainbow tables, false alarms. 9](#_Toc93341477)

[Rainbow table cracking, and a comparison with brute forcing. 10](#_Toc93341478)

[Cracking BCH (10,6) hash with rainbow tables with method one 10](#_Toc93341479)

[Cracking 5-digit to 8-digit numerical passwords with rainbow tables, and a comparison with brute forcing 11](#_Toc93341480)

[References 12](#_Toc93341481)

# Verifying credit card numbers

## Introduction to Luhn

The Luhn algorithm can be used to check if a given credit card number is valid, meaning if the credit card number was mistyped, the Luhn algorithm can be used to check this (Tiwarekar, 2017).

## Luhn algorithm summary

To verify a 16-digit credit card number first the input string is split into individual characters. First loop though all digit in card number and take the modulus of that digit if it is zero then, double the digit, and then if the double of said digit is greater than 10, minus 9 from it as you can’t have a double-digit number. If the new digit was not greater than 10 set the new digit in the place of the old one in the credit card number. Finally, you must add up all digits and mod by 10, if this gives a number other than 0 it is not a Valid credit card number, if it does give zero then it is a valid credit card number.

# BCH (10,6) Generating and Correcting

## Introduction to BCH (10,6)

BCH codes are cyclic codes that can be created to correct errors in a given code (Ma, Feng, Liu, Li, 2018). To further illustrate what BCH is, we must discuss cyclic codes, a cyclic code is a group code wherein which has an added element that the cyclic shift of a given code vector is a code vector in itself (Peterson, Brown, 1961).

## BCH (10,6) Generating

Generating BCH (10,6) codes from 6-digit integers is achieved by doing the following, the 6 provided digits are split into an array containing each digit. Then the 4 extra digits are then generated using these 4 lines of code:

number.add((4\*number.get(0)+10\*number.get(1)+9\*number.get(2)+2\*number.get(3)+number.get(4)+7\*number.get(5)) % 11); number.add((7\*number.get(0)+8\*number.get(1)+7\*number.get(2)+number.get(3)+9\*number.get(4)+6\*number.get(5)) % 11); number.add((9\*number.get(0)+number.get(1)+7\*number.get(2)+8\*number.get(3)+7\*number.get(4)+7\*number.get(5)) % 11 );

number.add((number.get(0)+2\*number.get(1)+9\*number.get(2)+10\*number.get(3)+4\*number.get(4)+number.get(5)) % 11);

After the 4 numbers are generated, they are checked to see if any are above 10, if they are then the BCH (10,6) code is unusable. If not the newly generated numbers can be concatenated on the end of the previous 6 digits used to generate the new 4 giving a 10-digit BCH (10,6) code.

## Correcting BCH code

To correct BCH (10,6) codes first step is to produce 4 syndromes using the number in the BCH (10,6) code you want to correct. Next to check to see if the syndromes are zero, if they are then the BCH (10,6) code has more than 2 errors, if not continue to the next step which is to find the value of P, Q and R using these formulas:

P = ((s2 \* s2) - s1 \* s3) % 11;

Q = (s1 \* s4 - s2 \* s3) % 11;

R = ((s3 \* s3) - s2 \* s4) % 11;

Then it is needed to check if they are less than zero if they are then 11 must be added to them. If P, Q or R are all equal to 0 then there is only one error, the position of the error can then be calculated and the magnitude in this case is simply equal to syndrome 1, if the pos of the error generated by this formula:

pos = (s2 \* inverce(s1) % 11);

The inverse of s1 needs to be calculated and timed by s2 and mod by 11 in this case.

equals zero then there are more than 2 errors, if however, it is not then the error can be corrected.

If P, Q and R are not equal to 0 then there is more than one error, this quadratic formula is applied to Q, P and R:

answer = (((Q \* Q) % 11) - ((4 \* P \* R) % 11)) % 11;

If the answer to the square root of this is -1 or 0 then the BCH (10,6) code has more than two errors. If not, then error correction can continue, using Q and P the positions of both errors can be found, and so can the magnitudes of both errors. If either of the position values are 0 then there are more than two errors, and we can’t continue to correct, however if there is not then then both errors can be corrected and if the corrected errors are not greater than 10, the errors can be corrected, and the BCH can be updated with the corrections.

The main issue with the algorithm implemented is it can’t correct more than 2 errors.

# Brute Force Password Cracking

## Cracking passwords with brute force

Brute forcing passwords can be achieved by running through every possible combination of a given character set for example [A-Z] at a given length for example a password length of 6, which means the program would produce every possible combination of A-Z with a length of 6 for example “ABZDED”. A brute force attack relies solely on the raw processing power of the computer it is being implemented on, rather than the skill of the hacker (Bosnjak, Sres, Brumen, 2018).

According to the conference paper (Bosnjak, Sres, Brumen, 2018), it took approximately 20 seconds to brute force a password with 2 lowercase letters followed by 4 digits and 20 hours to crack a password with a length of 10 with a charset consisting of lowercase English alphabet and numbers.

This gives an indication of the times we can expect when cracking passwords, the smaller the password and the smaller the char set the sorter the time it takes to brute force it.

Using a recursive method of generating possible combinations these are the cracking times for passwords up to a password length of 6. I.E starting from a, b, c, d etc then aa, ab, ac etc, ba, bb, bc, etc, aaa, aab, aac etc and slowly generating all combinations up to the length of 6.

|  |  |
| --- | --- |
| Password | Time to crack in seconds |
| this | 0.7236393 |
| is | 0.0008134 |
| very | 0.7234387 |
| simple | 674.4108448 |
| fail7 | 6.0306154 |
| 5you5 | 31.6836563 |
| 3crack | 1017.794178 |
| 1you1 | 27.6481319 |
| 00if00 | 1114.834087 |
| cannot | 104.6053995 |
| 4this4 | 1051.127791 |
| 6will | 31.975261 |

As can be seen from the table above the redder the number the longer it took. From this we can conclude that the longer it took the longer and more complex the password was.

From looking further into the results, we can see two very similar passwords, “00if00” and “cannot” taking significantly different amounts of time to complete even though they are of identical length, this is likely due to the char set and the position of the characters in the password within said char set, I.E the numbers are further in the charset used to break these passwords hash for example.

Table

Description automatically generatedHere is some output from the brute forcing code to illustrate this, as can be seen it does not go through 0-9 in the first position until it has gone through a-z in the first position, using this example, it is easy to see why “00if00” too much longer to crack than “cannot”. The numbers are last in the char array therefore they are the last to appear in each position of the password. The char array used in this example: "a", "b", "c", "d", "e", "f", "g", "h", "i", "j", "k", "l", "m", "n", "o", "p", "q", "r", "s", "t", "u", "v", "w", "x", "y", "z", "0", "1", "2", "3", "4", "5", "6", "7", "8", "9".

The main reason why longer passwords take more time in this case, is that the algorithm runs through every possible combination of the char set up to a given maximum password length, for example, if told to generate passwords up to the length of 8, it would go through passwords, 1-7 also. The reason this is done is the user is assumed to have minimal knowledge of the passwords they are trying to crack, therefore doing it this method it ensures that all lengths are checked up to the specified point, to give it the best chance of cracking the password hash. However, this does add cracking time as if the password hash is length of 9, it must go through password lengths of 1-8 also, assuming the maximum password length has been set to 9.

## Cracking BCH (10,6) codes with brute forcing algorithm

The next section consists of trying to crack hashed BCH codes, there are two ways of doing this.

### Method One

The first is to go through each combination of numbers [0-9] with a password length of 10. Like the first section, cracking passwords with brute force, going through each possible combination with a password length of 10 takes a considerable amount of time as can be seen from the results bellow.

|  |  |
| --- | --- |
| Password cracked | Time it took in seconds |
| 0000118435 | 3.2858832 |
| 1111110565 | 735.7151244 |
| 8888880747 | 5747.7130476 |

From these results it is obvious that in this case the larger the number the longer it took.

### Method Two

However, the second method of cracking BCH code is only generating the first six digits I.E generating all passwords with a length of 6 with the char set [1-9] then generating the last 4 digits of the BCH code using the first 6 numbers, checking to see if it is a valid BCH code and then ignoring it if it is not, if it is then hash and comparing it to the hash you are trying to crack.

|  |  |
| --- | --- |
| Password cracked | Time it took in seconds |
| 0000118435 | 0.007755499 |
| 1111110565 | 0.1738313 |
| 8888880747 | 0.771245799 |

As can be seen from the results above this is a far more efficient way of doing it compared to, generating all combinations of [0-9] with a password length of 10 takes considerably longer than only generating a password with 6 characters and then generating the last 4 then comparing hashes.

## Brute forcing algorithms analysis

The brute force method of cracking password hash does have its merits, its relatively simple to implement, and very easy to use once parameter like the char set, and password length have been decided. Of course, this is perhaps a disadvantage to brute forcing algorithms, you must know or have an idea of what passwords you are trying to crack, such as knowing the length of the passwords, and what characters reside within them. Not including a selection of characters in your char set, like symbols will result with passwords that contain these characters from being able to be cracked. Likewise, if a brute forcing algorithm is set to only crack passwords up to a certain length, in the case of the tested algorithm, six, trying to crack passwords with a length of 8 for example would be impossible. Failing to account for these factors would result in wasted time and resources.

# Using Rainbow Tables to Break Passwords

## Rainbow table overview

Unlike brute force methods of hash cracking, rainbow table has a time memory trade off (Horalek, Holık, Horak, Petr, Sobeslav, 2017). This means that the rainbow table sacrifices memory to reduce the time it takes to crack password hash (Horalek, Holık, Horak, Petr, Sobeslav, 2017). For a rainbow table to be effective you need to know the characteristics of the passwords you are trying to crack (Horalek, Holık, Horak, Petr, Sobeslav, 2017). For example, to generate the passwords, having a character set is necessary to determine what characters will appear in the password in the rainbow table, for example:

public ArrayList<String> alpha = new ArrayList<String>(Arrays.asList("0","1","2","3","4","5","6","7","8","9","a","b","c","d","e","f","g","h","i","j","k","l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"));

Also, other characteristics like the length of the passwords are needed to produce a rainbow table.

## Building of the table

Generating rainbow tables is simple, it consists of repeated hashing of passwords and reduction of hash to generate a chain. The first step in this process is to hash a given input password then reducing it to produce an alpha numerical output, or an outputted password with characters from the char set (In the case of this program uses a random number from 0 to 2147483647 as the beginning of the chain), these steps are then repeated until you have a chain the length of your chosen chain length. Only the first password and the last password are stored in the chain to save space. After a chain has been completed, the end of the completed chain needs to be checked and compared to the previous ends of the chains inside the rainbow table. This is to avoid something called collisions, which is where two or more chains hold the same last password, it is essentially a duplicate chain. Allowing these duplicate chains, would simply result in wasted memory. The chains that are duplicate, are simply discarded.

The example bellow contains in total 109,132 chains, this is accounting for the chain losses when checking for collisions. The number of chains it aims to generate is 6,000,000. The chain length for each chain is 2000. It contains passwords generated from a [0-9] numeric char set, and the passwords within can contain passwords from a length of 0 to 8.

Using this formula, we should be able to see how many chains we will need:

**Number of chains \* chain len = covered passwords**

6000000 \* 2000 = 12,000,000,000 (number of passwords covered)

And so:

Times by 1.5 to account for collisions:

1.3 \* 11111111111 = 14,444,444,444 (number of passwords we need to cover)

Is 12,000,000,000 > 14,444,444,444 = false

There for in this test example we may not be able to cover all passwords, depending on how many collisions we get.

Graphical user interface

Description automatically generated

The rainbow table was constructed in 26175297145200 ns (Nano seconds) or 436 Minutes (7.2 hours). This may seem like a significant amount of time; however, the table only needs to be generated once and then can be used to crack as many passwords hash as needed.

After generating the table, we can see that there were 5,890,868 collisions therefore:

**Number of rows needed to generate – Collision amount = Number of rows generated**

6,000,000 – 5,890,868 = 109,132 (Number of rows in table).

**Number of actual rows in table \* chain len = Number of passwords it covers.**

109,132 \* 2000 = 218,264,000 (Passwords covered)

**Password space – Passwords Covered = Number of passwords not in table**

11111111111 - 218,264,000 = 10,892,847,111 passwords not in table.

Unfortunately, in this case, the table generated does not cover all passwords. This may be in part due to the number of collisions.

## Cracking passwords with rainbow tables, false alarms.

Table

Description automatically generated with medium confidence  
  
When testing found an example of a false alarm, where a chain was thought to contain a password however it did not. There was a merging issue in my program where two chains collided and merged and mapped a password to the same position, as can be seen by the example above where the two chains merge at position 76 in both examples of the chain.

This however does not happen if the password is in the rainbow table itself, however it consistently happens when the password is not in the rainbow table, this however is only the case when dealing with smaller “Test” rainbow tables, when creating and using a much larger rainbow table this issue is present, but it is possible to get around it, without the need to completely fix it, observe: Graphical user interface, text, application

Description automatically generated

This work around involves allowing the rainbow table cracking program to continue searching even after it finds a chain that it thinks a password is in and does not find the password. The example above had 113 false alarms (Every “Recompiling chain” indicates that it has found another chain with the password in it) before it finally found the password. Meaning it thought the password was in 112 of these when this simply was not the case.

There are many disadvantages to this approach to the problem however, as this is going to affect the password hash cracking time to varying degrees, depending on multiple factors, where the password is in the rainbow table, if it is in the very last chain this issue could make the time it takes to find it quite substantial, based on the second factor, how many false alarms occur on the journey to the password. If there are many, and like with our example the password is actually located right at the bottom of the table, it will take a substantial amount of time to reach it, likewise this factor also depends on the size of the table in terms of both chain length and row amount, both of these factors can significantly affect the time it takes to crack a given password, and of course the purpose of a rainbow table is to store in memory all possible passwords in a given password space, knowing this, it is likely that this issue could void the whole point of a rainbow table as of course the point of a rainbow table is to trade memory for a decrease in cracking time, if the issue manifests itself severe enough with all of these factors being in there worst possible state, I.E large amounts of chains and rows, the password is at the very bottom of the table, and there are many false alarms, cracking a single password could possibly be as slow as something like brute forcing.

This issue could be solved however, by increasing the diversity of chains by editing the reduction function so that it is more unique, in the example above to account for collisions, the current position in the chain is added to the converted big integer (prior to the conversion it was a sha1 hash) to try and reduce the number of collisions and false alarms caused. Of course, doing so would make the cracking section of the rainbow table more complex as consistency is of paramount importance when dealing with rainbow tables.

## Rainbow table cracking, and a comparison with brute forcing.

### Cracking BCH (10,6) hash with rainbow tables with method one

In the brute forcing section of the report under the “Cracking BCH (10,6) codes with brute forcing algorithm”, there were two methods applied to BCH (10,6) cracking the first brute forcing all 10 digits at once and brute forcing 6 digits and then generating the next 4 and checking if it was a valid BCH code. In this comparison the time taken to brute force all 10 digits will be compared to the time it takes to do the same but with rainbow tables.

##### Rainbow table results

Graphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generatedGraphical user interface, text, application

Description automatically generated

|  |  |  |  |
| --- | --- | --- | --- |
| BCH Code | Brute forcing Time ns/sec | Rainbow table Cracking time ns/sec | Rainbow table Cracked |
| 0000118435 | 3285883200 ns / 3.2858832 sec | 77194480 ns /  0.0771 sec | Yes |
| 1111110565 | 735715124400 | N/A | No |
| 8888880747 | 5747713047600 | N/A | No |

As can be seen from the above times, the rainbow table failed to crack two BCH code hash, likely because the rainbow table generated, did not have enough coverage. However, the BCH code that rainbow table method did crack was much faster at 0.0771 seconds compared with brute forcings 3285883200 ns indicating even though, the issue described in the above section “Cracking passwords with rainbow tables “rainbow tables, in this case was much faster than brute forcing the same hash.

Due to the issues displayed above with only one of the hashes being cracked by rainbow tables, more testing is required to conclude.

### Cracking 5-digit to 8-digit numerical passwords with rainbow tables, and a comparison with brute forcing

**The hash used to test is contained in the “Test data for rainbow tables 1” document on blackboard.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Hash | Password | Brute force Time to crack ns/sec | Rainbow table Time to crack ns/sec | Rainbow Table Cracked |
| fe635ae88967693bc7e7eead87906e62e472c52f | 187494 | 290879801 ns /  0.290879801 sec | 255148100 ns /  0.2551481 sec | Yes |
| 3ac2d907663deccd843f9bbcf0c63bd3ad885a0e | 940376 | 743803200 ns /  0.7438032 sec | 11940300 ns /  0.0119403 sec | Yes |
| 3557c095ed6c16a90febda48d6b3a4490107b0d9 | 1098368 | 1483904600 ns /  1.4839046 sec | 933376900 ns / 0.9333769 sec | Yes |
| 85e04129ed328d4a2b3eedabca74d08b3e6badc1 | 0987593 | 1413957300 ns /  1.4139573 sec | 855116800 ns /  0.8551168 sec | Yes |
| 70352f41061eda4ff3c322094af068ba70c3b38b | 00000000 | 7502602400 ns /  7.5026024 sec | 1658254300 ns /  1.6582543 sec | Yes |
| 052bd5b02559d1270866c5626538e720cec0c135 | 93020840 | 70561425800 ns /  70.5614258 sec | 1907287300 ns /  1.9072873 sec | Yes |
| 3e71f65d56cb29521ac16ff1f92ecace156b1db5 | 87657890 | 66712830100 ns /  66.7128301 sec | 3959030900 ns /  3.9590309 sec | Yes |
| bfc52d4e36cb45cb667749982755e63630f3bc93 | 09680243 | 14004581600 ns /  14.0045816 sec | 5962254300 ns /  5.9622543 sec | Yes |
| 8cb2237d0679ca88db6464eac60da96345513964 | 12345 | 16419700 ns /  0.0164197 sec | 235980500 ns /  0.2359805 sec | Yes |
| 38bbc0a1ca7e9b3e9f6ab33782e0f780f009db1f | 99887766 | 75027175300 ns /  75.0271753 sec | N/A | No |

Slower times are marked in red. As can be seen from the results it is very clear that rainbow tables are much faster at cracking passwords, this makes sense as rainbow tables trade memory for faster cracking times. Unlike brute forcing which takes a lot of time to complete. If the false alarm issue was not present as discussed in the section “Cracking passwords with rainbow tables, false alarms” it is very likely that rainbow tables cracking times would be even faster, however, even with this issue rainbow tables have proven to be far more efficient at cracking passwords, except in the instance of the password “12345” where the brute forcing algorithm was faster, this was likely because of the false alarm issue present in the rainbow tables.

# UI Design for each task:

## ISBN and Luhn:

Graphical user interface, application

Description automatically generated

## BCH:

Graphical user interface, application

Description automatically generated  
Brute Forcing and SHA1:

Graphical user interface, text, application

Description automatically generated

## Rainbow Tables:

Graphical user interface, application

Description automatically generated

All UIs have methods for checking user inputs, the given screen shots are taken within NetBeans editor view not whilst the program is running.

# References

Tiwarekar, P.T. (2017) A Novel Idea for Credit Card Fraud Detection using  
Decision Tree. International Journal of Computer Applications [online]. Volume 161. [Accessed 24 December 2021].

Ma, Feng, Liu, Li, Y.M, X.F, Y.L, S.L. (2018) BCH–LSH: a new scheme of locality-sensitive hashing. IET Image Processing[online]. Volume 12. [Accessed 24 December 2021].

Peterson, Brown, W.W.P, D.T.B. (1961) Cyclic Codes for Error Detection. Proceedings of the IRE[online]. Volume 49. Pages 228-235 [Accessed 24 December 2021].

Bosnjak, Sres, Brumen, L.B, J.S ,B.B. (2018) Brute force and dictionary attack on hashed real-world passwords. In: Anon. Proceedings of the International Convention MIPRO. Opatija, Croatia, 2018,IEEE [Accessed 24 December 2021].

Horalek, Holık, Horak, Petr, Sobeslav, J.H, F.H, O.H, L.P, V.S. (2017) Analysis of the use of Rainbow Tables to break hash. Journal of intelligent & fuzzy systems [online]. Volume 32, page 1523-1534. [Accessed24 December 2021].