Author: Andrew Belcher
Student ID: 17010347

Cryptography Task Analysis and Implementation

Author: Andrew Belcher
Student ID: 17010347

Task 1: Verify Credit Card Numbers

The first task given was to deal with verification of two types of numbers. ISBN or international standard book number, and Credit Card numbers. The former, ISBN numbers, will target the pre 2007 version of the standard, which implements 10 digits instead of the latest using 13 **(Isbn-international.org, 2019).** In addition, for the credit card number we will not need to follow a standard such as **ISO/IEC 7812-1:2015** that closely since the information within the card number will not be verified, just if the number at the end can be divisible by 10. This was implemented using the Luhn Algorithm and the 16 number format for the credit card number. **(Ijcsmc.com, 2019)**

Starting with the ISBN number, we need to define the use of it. The key to verifying the full ISBN number is the last digit; this is the number we will check against at the end of processing all previous numbers. The number must be a ten digit number, however we can denote the error checking digit(final digit) as an asci 'x' to symbolize the number 10 as 1 character instead of 2, so that the ISBN remains 10 digits long. The use of 'x' will take care of the edge case of a valid ISBN number being reduced to a 10 after modulo arithmetic on the error-checking digit. This edge case is only present since the algorithm needed to verify the full ISBN number must use modulo 11 arithmetic leaving 10 as a possible output. The algorithm used to produce the error-checking digit is to take the current index of the number in the ISBN and multiply it by the value at that index. The result should be added to the previous held value, if we are at the start it is 0. At the end, we should have a checksum value that will need to be mod with 11 to produce a digit 0 to 10 where 10 can be represented by 'x'. We can write this algorithm in a c like fashion, where A is the result and D is the ISBN that is being iterated by the number I. **(math.ucdenver.edu, 2019)**

$$A = (D[I] * I) + A$$

The reason the index is used in this algorithm is to prevent transposition errors from occurring as well. Transposition errors occur when 2 or more items in a sequence have changed position, usually interchanging witch each other positions. **(dictionary.com, 2019)** If we think about the ISBN error checking number we won't be able to detect if the number is invalid if we were only calculating a sum based on the values without taking into consideration the position. So this algorithm will not only check for incorrect data, it will check for incorrect positioning of correct data.

Author: Andrew Belcher
Student ID: 17010347

The credit card half of the task is to verify a 16 digit credit card number using luhns algorithm which is a mod 10 implementation. We use mod 10 so that any number over 9 will be wrapped back around since the check digit at the end can only be 1 digit. The algorithm can work either way, going left to right or right to left in the 16 number sequence. As long as the check digit isn't counted, meaning if we start on the right we will be on the 15th digit, and if we start on the left we will be on the 1st. This aligns us to an even index, so it doesn't matter which way we start sequencing the number. The weighted sum this time doesn't come from the index, but if the index is odd or even. Every even index of the number will be checked, so that every evenly counted number is doubled, and the odd counted number remains the same. **(Ijcsmc.com, 2019)**

The rules for doubling a number are that if it goes over 10, we can store it in a single digit so it must be reduced with mod 10 to produce a single digit number. At the end we should have 16 numbers(check digit included) with each evenly counted number doubled and mod 10. To verify if the number is correct, we can take the sum of all the newly processed 16 numbers and mod 10 on them again, if this number mod 10 produces 0 then we know the credit card number is valid. if not then the number was corrupted or a transposition error has occurred. Since only every even number is doubled, this means that the weight of the final checksum is based on adjacent placement. This solves simple transposition errors where 2 numbers might swap, one in the evens place, and another in the odds. However if 2 numbers swap places in even places, the number will still be considered valid, if Luhn's algorithm implemented the index directing into the weighted checksum then it would detect all transposition errors. It is noted that this algorithm is meant to detect simple single errors, and not prevent against malicious attacks such as multi non-adjacent transposition errors. **(Ijcsmc.com, 2019)**

Author: Andrew Belcher
Student ID: 17010347

Task 2: BCH Generation and Error Correction

Previously we were tasked with validating data through error checking, for this task not only have to validate the data, but correct it as well if there are any transposition or corruption issues. The target of this task is a binary code, one from BCH or the (Bose, Chaudhuri, and Hocquenghem) code class. **(www-users.math.umn.edu, 2019)** Specifically BCH 10,6 which utilizes 10 decimal numbers with 6 of which being the message, and the 4 remaining having the purpose of error correction/detection. This code still uses a hamming code but rather than one parity check, it utilizes a matrix of parity checks, this is known as a q-ary hamming code. With this we can detect multiple existing errors in the 6 digit decimal message as well as the magnitude of the error. However since we are only using 4 decimal numbers to effectively produce a compressed encoding of the data we are limited to the correction of a maximum of 2 incorrect numbers in the message. If 3 or more numbers are incorrect, then the encoded data in the 4 numbers won't be enough to determine the magnitude of each of the errors. When dealing with more than 2 errors, we will at least know that there are more than two errors.

The metadata regarding the error correction on the message are known as syndromes, which are parity digits encoded as the last 4 digits in our message. To detect and correct any errors in our message, we need to first decode these syndromes and validate the message. If all the syndromes decoded are 0 then the message was transmitted intact. However if any of the syndromes when decoded using the data from the message aren't 0 then one or more error exists. To determine how many errors exist in the transmitted message, we need 3 variables that pull out error position information from the data. In our case we use p, q, and r. If p, q, and are turn out to be all 0 due to the way they reveal each syndromes relationship to each other, then we know only one error exist in the message. If they aren't all 0 then the syndromes contain multiple error related data that needs to be separated to correct the message. If one error is detected, then we will need just 2 pieces of information to correct it. The position, and the magnitude of the error. The position is determined by how many times the first decoded syndrome can make up the 2^{nd} syndromes value, and the magnitude is determined by the value of the 1^{st} decoded syndrome. An important note to remember was that when the magnitude was applied to the correct digit now that the position of the error was known, then if it rolled under 0 during correction, we need to make sure it wraps around with regards to mod 11 arithmetic.

Error correction in this phase becomes a bit more difficult, since to correct only 2 errors we need all variables to be correct since the 2nd error information is entangled with the 1st errors. Getting one wrong means getting both wrong. To do so we needs to use p, q, and r again to determine both positions of the error(I, J) to separate both we need to do an addition in calculating I and a subtraction in J using the square root of a product of (p,q,r) under mod 11 arithmetic. For the magnitude of both errors, things become even more critical to get right, since the calculation of the 1st errors magnitude (A) relies on the correct calculation of the 2nd errors magnitude. B magnitude also relies on both positions being correct as well. In the implementation of this algorithm, it was common to get both A and B wrong due to their reliance on each other, but for I and J both or one could be correct without effecting the calculation of the other. This part of the implementation proved the trickiest, since there are so many places that calculation can be incorrect, the trick is to always check after some arithmetic to make sure the value is valid in mod 11 arithmetic and if negative to properly convert it so. **(Web.ntpu.edu.tw, 2019)**

To separate out double errors or more than two errors, it is critical to handle two edge cases. One case is to get the square root of (q^2-4*p*r) under mod 11 arithmetic. If there is not one then we have located a message with potentially more than two errors. Additionally to be sure, that the message has more than errors we need to evaluate the position/magnitudes calculated for the message. If both I/J target positions as well as both A/B magnitudes existing within the valid range, then we are sure that we have more than 2 errors in the message. For more than two errors, we can't pull out the required information to correct the message, so we are only required to alert that there are more than two errors. **(Web.ntpu.edu.tw, 2019)**

Author: Andrew Belcher
Student ID: 17010347

Task 3: Brute Force Password Cracking

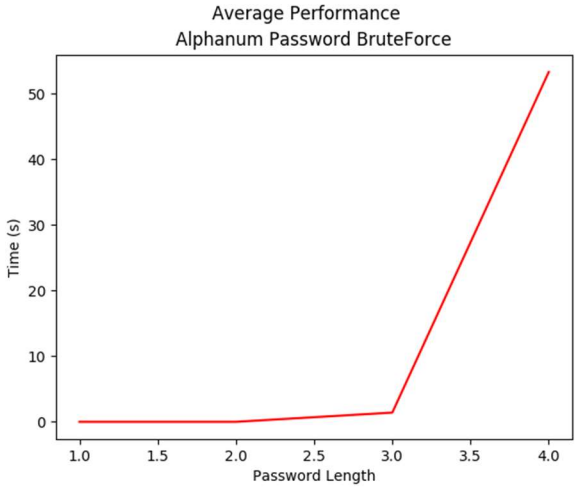
In Task 3 we shift focus from error detection and correction, to general cryptography. This task is rather simple in that it only requires the generation of plaintext and cipher text, and a comparison of this cipher text to another. In cryptography, plaintext is thought of as a non-encoded message, often in readable format. For this task it will be a 6 character long password. The cipher text will be a hash, one generated from a hash function such as SHA-1.

SHA or Secure Hash Algorithms are cryptographic functions that transform data through bitwise, modular, and compression operations. **(Brilliant.org, 2019)** A hash function is a cryptographic function that can produce a digital signature of a variable sized data set, it also is a function that requires very low computation power to produce an output. To reverse such output to the original input can be exponentially harder. Due to this factor, for this task the least efficient way to get to the original plaintext is to try to reverse the hash function using the cipher text output. Instead, we will use the efficiency of the design of the hash function against itself by supplying any possible plaintext we can generate within 6 alphanumeric characters and see if we have produced the cipher text output that we are looking for.

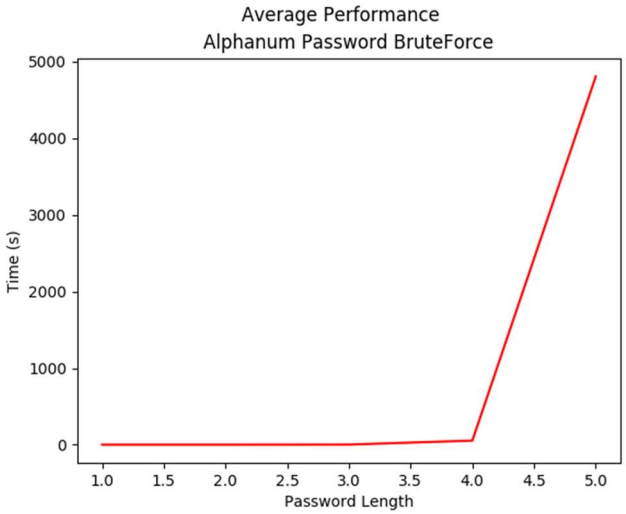
SHA-1 produces a 160-bit message digest so we will interpret this as a 40-character long 20-byte hexadecimal string. **(Brilliant.org, 2019)** To produce every possible character in an alphanumeric six character long password, will need to iterate on the character set. In our case, we are only using the lower case alphabet charset leaving us with 36 possible characters to be attempted for each character. Each time we iterate through the set, the next character in the password guess will be incremented until it reaches 36 where we will wrap it under again. Doing so will require 36^6 combinations, which around 2 billion possible passwords we need to search through. Because the search space with each password character extended is exponential, we have restricted the search to six characters long as to perform several password cracks within a practical period. If we average out the timings for each password cracked depending on the password length, we end up with a typical logarithmic scale graphing of the search times. It's hard to see on just one graph featuring 1 to 6 char long password strings, so we can generate a few graphs for 4,5, and an estimated timing of 6 password long brute force timings. Doing so reveals the logarithmic scale of the algorithms performance.

Author: Andrew Belcher
Student ID: 17010347

Four char long sample range

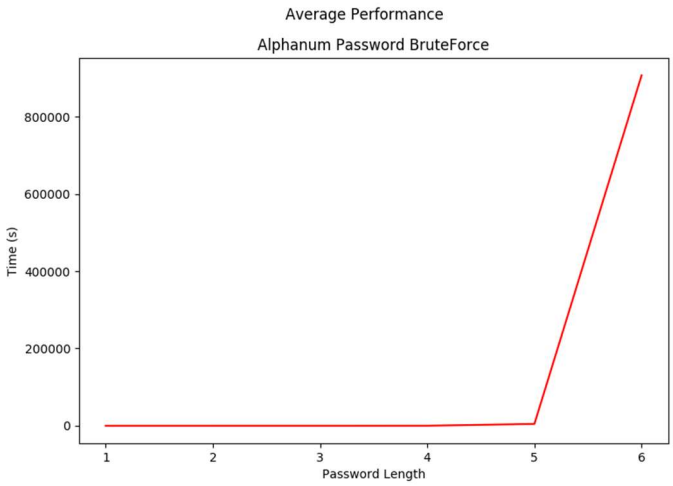


Five char long sample range



Author: Andrew Belcher
Student ID: 17010347

Six char long sample range



Author: Andrew Belcher
Student ID: 17010347

To complete the Task we were also required to crack BCH 10, 6 codes as well, this was a lot simpler since only 6^{10} was the search space due to the syndromes being calculated each time from the preceding message. For each required password to crack, the time in milliseconds was recorded, and each set depending on password length was averaged out to produce a timing for a password of that length.

PASSWORD	TIME(ms)	AVERAGE
is	1	
this	1310	
very	1506	1408
6will	22669	
1you1	59115	
5you5	63973	
fail7	67632	53347
simple	320768	
3crack	761606	
cannot	1408631	
00if00	1874348	
4this4	2210701	4807493
0000118435	6	
1111110565	1266	
8888880747	2251	

Task 4: Using Rainbow Tables to Crack Passwords

Finally, Task 4, which is just an extension of the concepts explored in Task 3, but this time focusing on data compression. In terms of time, guessing each password incrementally and comparing the hash is very expensive. Another way to tackle cracking the password based on its hash is to utilize a wordlist say of the most common or most likely used passwords. Depending on how you generate a wordlist, your size will vary, but if it is near the actual full search space for the password then it can be massive, introducing another expensive method of cracking the password. Rainbow tables deal with both these problems, they don't take up a ton of space nor do they require as much time to find the password than a traditional brute force method.

The way that a rainbow table works is that they compromise storage space by generating chains. Chains are sequences of hash and reduction strings that start and end with plaintext. The start will be some randomly generated string and the end will be the reduction of this randomly generated string after continual reduction/hashing, which is statically set as the chain length. At the core of chain generation is the reduction function. A reduction function is a way of producing plaintext from a hash, the opposite of what a hash function does. However, a reduction function is not a hash function inverter since it can produce the original plaintext using the hash of some plaintext. Instead, it will create a new plaintext that we can use to eventually match with the end of a chain. If we say start with the hash we want to crack and keep reducing it, we can keep checking if the reduction string of this target hash matches the key(or end) of one of the stored hash chains. If we find a match we know that somewhere in that chain can be the plaintext of the original hash. This is because we are still using our hash function to hash the reduced strings to a hash and then reduce again. **(Kestas.kuliukas.com, 2019)**

Doing so this way, we effectively can exhaust a search space without having to store all the points in that space. Instead, we are storing what can be thought as nodes or points along the way of that search space as a kind of short cut. During brute forcing, most of the time was lost in switching between comparison of the output hash to the target hash, and then regenerating a new password guess. This way ends up being a lot quicker because we will not have to alternate between checking and generation, but just focus on generating chains, and for cracking simple loop through and compare.

However, this method is not full proof, during implementation we will run into what is known as collision within our hash table. Collisions are when the hash function being used produces a non-unique hash; in fact collisions are always a possibility due to the fact that a hash function can take in an infinite input length and produce a finite output. The smaller the output size the more likely that the hash function will produce the same hash for two different inputs. To mitigate collisions in our rainbow table we will only add a chain if it has a unique key, or end reduction. This does introduce another issue though, where if we provide poor values for the chain length or chain number, then we may never be able to satisfy completing the table since we might not be able to create a new chain that does not exist in the table. Currently the user needs beforehand knowledge of this before using the tool; else, the program will never complete and need to be killed. **(Learncryptography.com, 2019)**

Bibliography

Brilliant.org. (2019). *Secure Hash Algorithms | Brilliant Math & Science Wiki*. [online] Available at: https://brilliant.org/wiki/secure-hashing-algorithms/ [Accessed 27 Nov. 2019].

Ijcsmc.com. (2019). [online] Available at: https://ijcsmc.com/docs/papers/July2013/V2I7201373.pdf [Accessed 27 Nov. 2019].

Isbn-international.org. (2019). *What is an ISBN? | International ISBN Agency*. [online] Available at: https://www.isbn-international.org/content/what-isbn [Accessed 27 Nov. 2019].

Kestas.kuliukas.com. (2019). *How Rainbow Tables work*. [online] Available at: http://kestas.kuliukas.com/RainbowTables/ [Accessed 27 Nov. 2019].

Learncryptography.com. (2019). *Learn Cryptography - Hash Collision Attack*. [online] Available at: https://learncryptography.com/hash-functions/hash-collision-attack [Accessed 27 Nov. 2019].

Web.ntpu.edu.tw. (2019). [online] Available at: http://web.ntpu.edu.tw/~yshan/BCH_code.pdf [Accessed 27 Nov. 2019].

www.dictionary.com. (2019). *Definition of transpose | Dictionary.com*. [online] Available at: https://www.dictionary.com/browse/transpose [Accessed 27 Nov. 2019].

Www-math.ucdenver.edu. (2019). *ISBN - Code*. [online] Available at: http://www-math.ucdenver.edu/~wcherowi/jcorner/isbn.html [Accessed 27 Nov. 2019].

www-users.math.umn.edu. (2019). [online] Available at: http://www-users.math.umn.edu/~garrett/coding/Overheads/19_hamming_bch.pdf [Accessed 27 Nov. 2019].