UFCFT4-15-3 - Cryptography: Assignment

Harri Renney

December 12, 2017

Introduction

In this assignment the goal was to complete a series of given tasks in error detection/correction and cryptography.

Task 1. Verifying credit card numbers

The first task was entirely focused on error detection in well known number codes ISBN and credit card numbers.

ISBN numbers work by using the 10th digit as a check digit for error detection.
It takes each digit, multiplies it by its index then adds it to total. If all 9 digits summed is divisible by 11 then code is valid otherwise at least one error present.
In ISBN, because modulus by 11 used, 10 is a possible value and therefore needs representing by some symbol, in this case an 'X'.

Credit card numbers work by using the last as a check digit to detect error as well. Similar to ISBN but last digit generated to satisfy Luhn condition or mod 10 check.
Every alternate number doubled and all values summed (Any that exceed 10 are reduced by 9 until below) and then if the total is divisible by 10 it passes Luhn check and is valid.

Task 2. BCH Generating and Correcting

The second task was focused on not only detecting but then correcting errors using BCH codes for multi-error correction. To begin with the program takes an original six digits and using some formulas generates 4 extra parity digits used for BCH.
The generation process requires modulus by 11. This means 10 is a possible computed value, rather than using a symbol to represent 10 like in ISBN it is simply termed an unusable number and a new one must be entered.

Now when the code is to be checked for errors. To begin with using some formulas 4 syndrome values are calculated.
If all syndrome values computed are equal to zero, then no error is present.

No Error: $S_1 = S_2 = S_3 = S_4 = 0$

If at least one is present then 3 new values (P,Q,R) are computed, if they are equal to zero then there is one error at a position and magnitude given by further equations using these three values.

$P = (S_2)^2 - S_1 * S_3$
$Q = S_1 * S_4 - S_2 * S_3$
$R = (S_3)^2 - S_2 * S_4$
Single Error: $P = Q = R = 0$

If they are not all zero, then quadratic equations used to find two positions and two magnitudes of the two errors.

$i = \frac{(-Q+\sqrt{Q^2-4*P*R}}{2*P}$

$j = \frac{(-Q-\sqrt{Q^2-4*P*R}}{2*P}$

If either position of error given is zero or there is no square root possible for quadratic equation, then there are more than two errors.

Task 3. Brute Force Password Cracking

The third task is to brute force a SHA-1 hash with password of original length 0-6 and with an alphabet of all lower case letters and all numbers.
Brute forcing is simply trying the hash of every combination of plain text in the search space of the alphabet and length and seeing if it matches the hash searching for. If the plain text that when hashed equals the hash searched for it is known that is the plain text.
The reason it can't be done the other way is because hashing, in particular SHA-1 hash is a one way function so doesn't map to exactly one the other way.

It was learnt from this task that there are two main ways of trying every combination in code.
Iteration, where loops are used to go through a range. Starting from some point like zero then to the maximum point, like the maximum length 6 at which point it leaves loop and

terminates.
Recursion, where a function within calls itself, within it can keep calling itself recursively. This requires some stopping condition to be met to not call itself again and leave the function.

SHA-1 Hashes tasked to crack

SHA-1 Hash	Plain Text	Time Taken(ms)
c2543fff3bfa6f144c2f06a7de6cd10c0b650cae	this	2,250
b47f363e2b430c0647f14deea3eced9b0ef300ce	is	309
e74295bfc2ed0b52d40073e8ebad555100df1380	very	3,436
0f7d0d088b6ea936fb25b477722d734706fe8b40	simple	3,712,365
77cfc481d3e76b543daf39e7f9bf86be2e664959	fail7	31,849
5cc48a1da13ad8cef1f5fad70ead8362aabc68a1	5you5	183,356
4bcc3a95bdd9a11b28883290b03086e82af90212	3crack	4,411,754
21e7133508c40bbdf2be8a7bdc35b7de0b618ae4	00if00	4,223,347
6ef80072f39071d4118a6e7890e209d4dd07e504	cannot	672,780
02285af8f969dc5c7b12be72fbce858997afe80a	4this4	4,689,254
57864da96344366865dd7cade69467d811a7961b	6will	186,226

It can be seen from the time taken that there is an exponential increase in relation to the length of the plain text.
Length two is in hundreds milliseconds, length four is in thousands, length five is in ten thousands then a huge increase to length six in millions.
From this it can be learnt that the processing time can get out of hand as password length increases and brute forcing looses its usefulness after about passwords of length 7 and other techniques are needed.

Task 4. Using Rainbow Tables to Break Passwords

There are two programs used to complete this task that both share a lot of the same functionality.
It was invented as a alternative attack to have a trade off between memory space and processing time.

Step 1 - Building a rainbow table

To begin with the search space size is calculated to decide on a decent rainbow table size and to find a prime value bigger than search space used in reduce function. Numbers so large handled here that the BigInteger API used to hold values.
For the table size a randomly generated starting value for the chain is made and then hashed, then the hashed value is reduced back to some plain text according to a reduce function. This process is repeated for however long the chain is chosen to be.
Once it stops, the last plain text value is taken along with the first at the beginning of the chain and stored as a pair in a table.

To avoid chain collisions a different reduce function for each link in the chain must be used, a trick that is employed for this task is to pass the position into the reduce function and use the position to change the reduce function in some way, making each act as if it is a different function. This is where it gets the name rainbow table from.
Also by adding checks if the beginning and end values in chains are unique it avoids collisions and so speed up later processing time when cracking (avoiding false alarms), but means at this stage table takes longer to build as chains can be thrown away if not unique.

It was learnt that using a hash map provided from some API to store the chain begin/end pairs is much better for program performance, having constant time complexity for searching a value compared to say an array which would require some longer search processing.

Step 2 - Using the rainbow table to break passwords

To find the plain text of the hash the hash code is taken and reduced, if it does not match any of the end chain values then it moves further up the chain from the end, reducing and hashing and checking if equal to any chain ends. If it is, then going from first value through chain until it finds the hash, then it knows the plain text before is the answer. If it gets to the end then a false alarm.

SHA-1 Hashes tasked to crack

SHA-1 Hash	Plain Text	Time Taken(ms)
fe635ae88967693bc7e7eead87906e62e472c52f	Not Found	N/A
3ac2d907663deccd843f9bbcf0c63bd3ad885a0e	940376	13,409
3557c095ed6c16a90febda48d6b3a4490107b0d9	1098368	10,593
85e04129ed328d4a2b3eedabca74d08b3e6badc1	0987593	129,060
70352f41061eda4ff3c322094af068ba70c3b38b	00000000	517
052bd5b02559d1270866c5626538e720cec0c135	93020840	24,435
3e71f65d56cb29521ac16ff1f92ecace156b1db5	87657890	80,225
bfc52d4e36cb45cb667749982755e63630f3bc93	09680243	1,930
8cb2237d0679ca88db6464eac60da96345513964	12345	4,750
38bbc0a1ca7e9b3e9f6ab33782e0f780f009db1f	Not Found	N/A

Two out of the ten passwords could not be found, this is because the plain text simply wasn't generated to be in the chains the table stored. Even though it is large enough to cover the search space, collisions mean there is redundancy and it needs to be either avoided even further or a bigger table generated to cover all search space.
However it can be seen although the search time isn't really dependant on the length like a brute force method, considering the length is 8 it outperforms brute forcing and what processing it does do is done to avoid huge amounts of memory like a dictionary attack might use.

Conclusion

All the tasks were complete to a satisfactory level, however it would be interesting in the future to see how the rainbow table can be improved to avoid further collisions and therefore reduce redundancy in chains and cover the entire search space for more efficient password hacking.