1. Introduction

The process of encryption and decryption of data is referred to as ***coding***. Coding is divided into three categories: ***error correction***, ***cryptography***, and ***data compression***. Data integrity is served by error correction, security is provided by cryptography, and memory space is saved by data compression. This report will cover error correction and encryption, as well as provide several example applications and test results from the practical sheets. The report includes an additional examination of Cryptography's integration with other modules. It shows an example of an encoding/decoding mechanism used for AI chatbots, as well as ethical use and construction.

2. Error correcting

Error-correcting codes are utilised when sending messages. Changes in the original message could be caused by external factors or human error. The purpose of error-correction codes is to encode data in such a way that even if errors occur, the original message may be retrieved. These codes are particularly useful when the message can only be sent once.

The International Standard Book Number is one example (ISBN) of error correction. The ISBN is a 10-digit code in which the first three digits provide information about the book, the next six digits are assigned by the author, and the last digit is used to detect errors.

Example of ISBN: **0-19-964398-3**

The first digit represents the book's language (0 is for English).
The following two digits hold information about the publisher (19 is for Oxford Uni. Press).
The last digit is calculated as follows: $d10 = (d_1 + 2d_2 + 3d_3 + \ldots + 9d_9) \bmod 11$.
When the value of **10** is assigned to **d10**, it is written down as "**X**".

The ISBN program's checking results indicate whether the given ISBN number are accepted or failed verification, rather than the program's overall success. My program successfully authenticated all test cases:

Test Case	Result
99921-58-10-7	**Test Pass Successfully**
99921-58-10-2	**Test Pass Successfully**
960-425-059-0	**Test Pass Successfully**
960-455-059-0	**Test Pass Successfully**
1-84356-028-3	**Test Pass Successfully**
0-943396-04-2	**Test Pass Successfully**
0-9752298-0-X	**Test Pass Successfully**
5-9752298-0-X	**Test Pass Successfully**

Pseudo code for ISBN verification app:

1) Get input string of length 10.
2) Separate input into chars Array.
3) Sum = (d1+2*d2+3*d3+4*d4+5*d5+6*d6+7*d7+8*d8+9*d9+10*d10).
4) Sum = Sum mod 11.
5) Check if the result equals 0:
   - If yes, code is valid and display a validation message.
   - If no, code is invalid.

A similar example for error correction represents the ***Luhn's algorithm***, which is used to generate credit card numbers. Credit cards feature 16-digit numbers, with the first six numbers holding issuer information, the next nine containing the bank account number, and the final digit the error correction.

1 2 3 4 5 6 7 8

Example of Credit card number: 4552-7204-1234-5693

- The **first six digits** represents the **issuer identifier number**.
- The **next nine** digits are the **account number**.
- The last digit represents **error detection** and is generated by the **Luhn's algorithm**.

Luhn's Check

- Numbers at **even positions** are summed: 5 + 2 + 2 + 4 + 2 + 4 + 6 = **25.**
- Numbers at **odd position** are multiplied by 2. If the multiplication result equals or exceeds 10, the multiplied value is subtracted by 9. (4x2=**8**; 5x2=10 10-9=**1**; 7x2=14 14-9=**5;** 0x2=**0**; 1x2=**2**; 3x2=**6**; 5x2=**1**; 9x2=**9**) = 8 + 1 + 5 + 0 + 2 + 6 + 1 + 9 = **32**
  25 + 32 = 57 |=> the last digit should be 3, to get a total sum that's perfectly divisible by 10.
  57 + 3 = 60 mod 10 = **0** |=> This is a valid credit card number

The Credit Cards Verification application was checked to see if the verification of the provided numbers was successful, rather than the program's overall performance. My software application successfully authenticated all test scenarios.

Test Case	Result
4552-7204-1234-5693	**Test Pass Successfully (60%10 = 0)**
4552-7204-1234-5698	**Test Pass Successfully (65%10 != 0)**

8552-7204-1234-5693	**Test Pass Successfully (64%10 != 0)**
5169-7662-2129-0945	**Test Pass Successfully (70%10 = 0)**
6169-7662-2129-0945	**Test Pass Successfully (71%10 != 0)**
5169-7662-2129-9945	**Test Pass Successfully (79%10 != 0)**

The two examples provided can detect errors, but they can only correct them when they are at the last digit. More efficient error-correcting codes include the Hamming and BCH codes. Their names are derived from the names of their creators. Hamming codes detect and correct a single error, whereas BCH codes can detect and correct multiple errors. Hamming (7, 4) works with binary numbers, whereas Hamming (10, 8) works with decimal numbers. Hamming (8, 4) can identify double and single faults but can only correct one. BCH (6, 10) can discover and correct up to two errors in the code.

The Hamming and BCH codes have extremely similar algorithms. Both encodings are achieved by producing parity check digits from supplied formulas. Some BCH codes are unusable under certain conditions. The decoding process consists of generating syndrome values that indicate whether or not an error has occurred. There are no errors in either example when all of the syndromes equal "0." The difference between them is that Hamming codes only use the values of the syndromes to detect and correct errors. BCH codes, on the other hand, generate new values (P, Q, R) from the syndromes and use these values to determine whether they are dealing with one, two, or three errors. Their error correction is computationally heavier because more mathematical operations are performed, but they are also more efficient because of their larger scale.

Test Cases for BCH Encoding Program:

Test Case	Result
000001	**0000017671**
000002	**0000023132**
000010	**0000101974**
000011	**0000118435**
000003	**Unusable number**
000009	**Unusable number**

The BCH (6, 10) program successfully recognizes one, two, and more errors, according to testing. It can also correct up to two errors.

Input	Output	Results
3745195876	3745195876	**No errors**
3945195876	3745195876	Single error
3745995876	3745195876	Single error
3715195076	3745195876	Double error
0743195876	3745195876	Double error
3745195840	3745195876	Double error
2745795878	-	More than 2 errors
8745105876	3745195876	Double error
1145195876	3745195876	Double error
3745191976	3745195876	Double error
3745190872	3745195876	Double error
3745102876	3745195876	Double error
3742102896	-	More than 2 errors
1115195876	-	More than 2 errors
3121195876	-	More than 2 errors

Extra testing on test cases that contain more than two errors:

Test Case	Test Result
3121195876	**Pass**
1135694766	**Pass**
0888888074	**Pass**
5614216009	**Pass**
9990909923	**Pass**
1836703776	**Pass**
9885980731	**Pass**

3. Cryptography

Data must be correct as well as secure during transmission. Cryptography and encryption methods come in helpful here. Some of the more common ones are SHA-1, MD5, RSA, and SHA-256. SHA-1 encryption is used in all of the applications presented in this research. Hashing a given plain text until it is no longer identifiable or meaningful is how encryption is performed. To retrieve the original plain text, the hash must be decrypted.

The first two demonstration applications employ *Brute Force deciphering* or "*password cracking*". Program A's purpose is to decrypt passwords of up to six bits in length, which can include both **digits (0-9)** and lowercase characters **(a-z)**.The program generates random strings of varying lengths within the specified range, hashes each string, and compares it to the list of

targets. Program B's goal is to decipher three valid BCH codes. It generates random six-digit strings and checks their parity. Checks to see if the result is a valid BCH code, and if so, compares its hash to the list of targets.

Although the potential of randomly generating a particular password seemed unlikely, both algorithms accomplished their goals in a respectable length of time. Within a 4-hour run, Program A had cracked all twelve passwords. Program B breaches all passwords for around three seconds. The disadvantage is that the algorithm's iterations may repeat some of the random strings. The position of the target affects the execution time when producing successive strings (e.g., 000, 001) – "111" takes longer than "000".

Passwords and time required for cracking – Program A:

Original Password	Cracked Password	Required Time (minutes)
c2543fff3bfa6f144c2f06a7de6cd10c0b650cae	***this***	**0.0403**
b47f363e2b430c0647f14deea3eced9b0ef300ce	***is***	**0.0012**
e74295bfc2ed0b52d40073e8ebad555100df1380	***very***	**0.1562**
0f7d0d088b6ea936fb25b477722d734706fe8b40	***simple***	**16.0397**
77cfc481d3e76b543daf39e7f9bf86be2e664959	***fail7***	**0.1206**
5cc48a1da13ad8cef1f5fad70ead8362aabc68a1	***5you5***	**2.1039**
4bcc3a95bdd9a11b28883290b03086e82af90212	***3crack***	**56.2198**
7302ba343c5ef19004df7489794a0adaee68d285	***1you1***	**5.719**
21e7133508c40bbdf2be8a7bdc35b7de0b618ae4	***00if00***	**62.1882**
6ef80072f39071d4118a6e7890e209d4dd07e504	***cannot***	**19.2195**
02285af8f969dc5c7b12be72fbce858997afe80a	***4this4***	**67.705**
57864da96344366865dd7cade69467d811a7961b	***6will***	**11.2031**
TOTAL TIME		**4hrs 0 min 42 sec**

Passwords and time required for cracking – Program B:

Original Password	Cracked Password	Required Time (seconds)
902608824fae2a1918d54d569d20819a4288a4e4	***0000118435***	**0.1508**
88d0b34055b79644196fce25f876bc1a5ef654d3	***1111110565***	**1.6115**
5b8f495b7f02b62eb228c5dbece7c2f81b60b9a3	***8888880747***	**2.90912**
TOTAL TIME		**4.67142**

Pseudo Code for program B:

1) Generate a random 6-bit string of digits.
2) Encode the string, generating the four parity digit.
3) Check if the generated BCH code is valid.
   - If yes, hash it and go to step 4)
   - If no, go back to step 1)
4) Compare the produced hash with the targets list.

Another way for deciphering hashes is to create a dictionary. The dictionary records passwords and hashes, however it only supports a limited number of passwords. It is an inconvenient approach since it takes up a lot of hard disc space. The Rainbow Tables approach optimizes hard disc space use. Rainbow tables conserve storage space by creating password chains and saving just the first and last strings.

Pass 1	Pass 2	Pass 3	Pass 4	Pass 5	Pass 6	**Pass 7**

Chains are primarily built using two types of functions: hashing and reduction. The hashing function is straightforward: it produces hashes from provided plain text. To generate new plain text from hashes, the reduction functions employ a variety of mathematical operations. The name "Rainbow" is derived from the table's various reduction functions, which resemble a rainbow.

Start	->	Reduction1	->	Reduction2	->	Reduction3	->	Reduction4	->	End
Plain text	Hash	Plain text	Hash	Plain text	Hash	Plain text	Hash	Plain text	Hash	Plain text

The last task involves writing a text encryption application using a stream Cipher and steganographic techniques. This app combines cryptography and steganography. It conceals an encrypted message within a regular message using a pseudo random number generator for the stream cipher. Assuming the first message is *"How are you today? I had a very busy day! I travelled 400 miles returning to London. It was windy and rainy. The traffic was bad too. I managed to finish my job, ref No 3789. But I am really tired. If possible, can we cancel tonight's meeting?"* and the second one *"meet@9"*, after encrypting the second message with the stream cipher, we will obtain:

```
    0x6d6565744039        (meet@9)
⊕   0xa73e80e2b563        (a random key stream)
-------------------------------------------
    0xca5be596f55a        (encrypted message)
```

0xca5be596f55a in binary is 1100 1010 0101 1011 1110 0101 1001 0110 1111 0101 0101 1010.

Pseudo Code for the Stream Cipher:

1) Calculate m ← $p_1 \cdot p_2$
2) Initialize the seed state x_0
3) Go through the randomNums array until **length-1** :
    a. var ← randomNums[i]^2 mod m
    b. randomNums[i+1] ← *findParity(***var***)*

4) Append the randomNums values to the Key

Following encryption, we will use a steganographic technique to "hide" the encrypted message in the first message and obtain the cipher text. In the reverse order, we must obtain the initial message, message1, and the encrypted message, message2.

Pseudo Code for Hiding text:

5) Add full stop at the end of the original message
6) Go through the original message
    a. if the character from position *p* is "1", then we insert at a random position the character " " (white space).
    b. if the character from position *p* is "0", then we insert at a random position the character ".".
7) Repeat Step 2) until the end of the string is reached.

After the encrypted message has been hidden, another message containing the encrypted message will be displayed:

A possible cipher-text:

> How are you today? I had a very busy day! I travelled 400 miles returning to London.. It was windy and rainy.. The traffic was bad too.. I managed to finish my job, ref No 3789. But I am really tired.. If possible, can we cancel tonight's meeting?

4. Combination of Cryptography and Error correction

Cryptography and error correction commonly interact. Maiorana and Ercole's Biometric Authentication System is one example (2007). The approach relies on user-adaptive error-correcting codes to secure the saved templates and cancel them if required, such as when a malware attack is detected. The integrity of the saved templates is ensured through error correction. Distributed cryptography enables hierarchical key management while also improving system fault tolerance. The system's testing yields positive results, notably for signature and iris recognition.

5. Combining Cryptography with different modules

Most aspects of computing and computer science may be coupled with cryptography. Many applications demand the security that encryption provides. To be ethical, applications that

use personal data must encrypt it in their database. Decryption aids application maintenance by letting developers to decode and evaluate data collected.

The following example integrates modules like Cryptography, Artificial Intelligence, and Ethics. In recent years, there has been a substantial increase in the use of AI chatbots. They help app developers and customers communicate better. Chatbots must never store customer chats in their database in order to be ethical. How can developers know if the chatbot is performing successfully in that case? Encryption is a straightforward answer to this problem. As long as the conversations are encrypted before storage, the chatbot may store customer chats while remaining ethical.

Algorithm for encryption of conversations:

Crypto → c + r + y + p + t + o ⟶ sha1(c) + sha1(r) + sha1(y) + sha1(p) + sha1(t) + sha1(o)

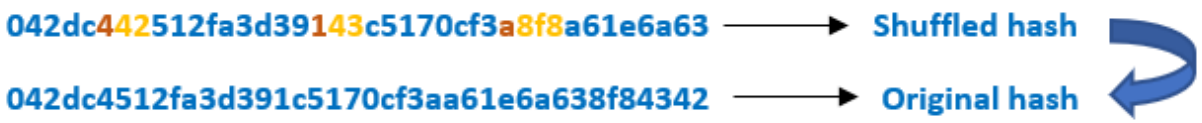
Every word in the conversation has been reduced to chars. The complete length of the word is recorded as a passive key, and each letter is encrypted independently. These keys have no influence on security and are simply used for reassembling the text after decoding. The necessity for quicker brute force decryption drove the idea to hash individual characters.

All hashes generated by the same encryption techniques have the same length, regardless of their plain content. SHA-1 has a length of 40 bits. Anyone with this information could interpret the overall hash and break it into individual hashes. That would be very straightforward because the hashed password space in this case is 1. (individual chars). It is intended to be rapid. To solve this problem, hash bits are relocated to specified key places before being merged into the big hash.

This is what a big hash looks like:

7cf184f475c67a68d5828329ecb19349720b0cae58e6b3a4ba14a112e090df7fc6029add0f3555cc0
7c342be856e56f00e7f4347842e2e21b774e61d07c342be856e56f00e7f4347842e2e21b774e61d7a
81af3e7d591a83c713f8761ea1efe93dcf36150ab8318a1bcaf639e678dd3d02e2b5c343ed4111ca7
3ab6…

The decryption technique in this application pulls the big hash from the database and divides it into individual hashes of 40 bits each. The hashes had previously been shuffled using a specified key pattern. Each bit is returned to its original position using the same procedure.



The concept of hashing individual characters arose from the requirement for rapid decryption of big texts (such as chatbot conversations). Even on big text files, brute forcing 1bit strings provides an exceptionally quick and easy decoding technique. The algorithm decrypts letters while maintaining their original arrangement. After converting all of the letters to plain text, words are constructed using the passive key values - the length of each word in the original text. The system understands all punctuation marks and will not be damaged if unfamiliar symbols are typed with the text. It also correctly restores all capital letters and punctuation, ensuring the data's integrity.

6. Conclusion

We already use cryptography and error correction software in our daily lives. The integrity of our data is ensured through error correction, and it is protected by cryptography. Their applications' convergence is tremendously useful, particularly in automated identification utilizing biometrics. The knowledge from this course was successfully applied to a large-scale project, which also included elements from two other modules.

(Total words count: 2415)

7. References

- Jain, A.K., (2004) *An introduction to biometric recognition, IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 14, No. 1

- Maiorana, E. and Ercole, C., (2007). Secure biometric authentication system architecture using error correcting codes and distributed cryptography. *Proceedings of Gruppo nazionale Telecomunicazioni e Teoria dell'Informazione*, pp.1-12.