**Introduction**

Coding is the encoding and decoding of data. There are three areas in coding – error correction, cryptography, and data compression. Error correction serves the data’s integrity, cryptography brings security, and data compression saves memory space. This report will go over error correction and encryption, providing several example applications and their testing. The report contains an additional review of Cryptography merging with other modules. It showcases an example application of encryption/decryption in AI chatbots for their Ethical use and construction.

**Error correcting**

Error-correcting codes get used when transmitting messages. External conditions or human errors could cause changes in the original message. The objective of error-correcting codes is to encode data in such a way that, even if errors occur, the original message can get recovered. These codes are especially useful if the message can only be transmitted once.

An example is the International Standard Book Number (ISBN). The ISBN is a 10-digit code where the first three digits give information about the book, the next six digits get assigned by the author, and the last digit is for error detecting.

**0-19-853287-3 ISBN**

**The first digit stands for the language of the book (0 is for English)**

**The next two digits hold information about the publisher (19 is for Oxford Uni. Press)**

**The last digit is worked out by: d10 = (d1 + 2d2 + 3d3 + …+ 9d9) mod 11**

When the value of **10** is assigned to **d10**, it is written down as “**X**”

The ISBN application’s testing results indicate whether the given ISBN **number** passed or failed verification and not the overall success of the program. The program successfully authenticated all test cases.

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

*Pseudo code for ISBN verification*:

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 = mod 11
5. Check if the result equals 0:

* If yes, code is valid
* If no, code is invalid

A similar example is the algorithm for generating credit cards numbers – the Luhn algorithm. Credit cards have 16-digit numbers – the first six numbers hold information about the issuer, the next nine make up the bank account number, and the final digit is for error detection.

Luhn Algorithm

1 2 3 4 5 6 7 8

4552-7204-1234-5693

The first six digits are the issuer identifier number.

The **next nine** are the account number.

The last digit is generated by the Luhn check and stands for error detection.

**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 result of the multiplication is equal to or greater than 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

Testing done on the Credit Cards Verification application show whether the verification of the given numbers was successful, not the overall success of the program. The program successfully authenticated all test cases.

|  |  |
| --- | --- |
| **Test Case** | **Result** |
| 4552-7204-1234-5693 | Verification Successful (60%10 = 0) |
| 4552-7204-1234-5698 | Verification Failed (65%10 != 0) |
| 8552-7204-1234-5693 | Verification Failed (64%10 != 0) |
| 5169-7662-2129-0945 | Verification Successful (70%10 = 0) |
| 6169-7662-2129-0945 | Verification Failed (71%10 != 0) |
| 5169-7662-2129-9945 | Verification Failed (79%10 != 0) |

The two example codes given can detect an error, but they can only correct it when it's at the last digit. More efficient error-correcting codes examples are the Hamming and the BCH codes. Both their names come from the names of their inventors. Hamming codes are single error detection and correction algorithms, whereas BCH codes can deal with multiple errors. Hamming (7, 4) works with binary numbers, and Hamming (10, 8) works with decimal ones. Hamming (8, 4) is expanded to detect double errors as well as single but can only correct one error. BCH (6, 10) can detect multiple errors in the code but can only correct up to two.

Hamming and BCH codes have very similar algorithms. Both of their encodings get done by generating parity checks digits derived from given formulas. Under certain conditions, some BCH codes are unusable. The decoding consists of generating syndrome values, which will show whether there is an error occurring. In both examples, when all the syndromes equal “0”, there are no errors. The difference between them is that Hamming codes only work with the syndromes' values to detect and correct the error occurring. Whereas BCH codes generate new values (P, Q, R) from the syndromes and use these values to find out whether they are dealing with one, two, or more errors. Their error correction is computationally heavier, as they undergo more mathematical operations, but they are also more efficient due to their larger scale.

*Testing of BCH encoding:*

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

Testing of the **BCH (6, 10)** program indicates that the program successfully recognizes one, two, and more errors. It also successfully corrects up to 2 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 |

Additional testing on test cases with more **than two errors**:

|  |  |
| --- | --- |
| Test Case | Test Result |
| 3121195876 | Pass |
| 1135694766 | Pass |
| 0888888074 | Pass |
| 5614216009 | Pass |
| 9990909923 | Pass |
| 1836703776 | Pass |
| 9885980731 | Pass |

**Cryptography**

During transmission, data should not only be correct but protected too. That is where cryptography comes in handy with encryption algorithms. Some of the more popular ones are - SHA-1, MD5, RSA, SHA-256. The applications demonstrated in this report are all using SHA-1 encryption. Encryption is done by hashing a given plain text until it is no longer recognizable and has no meaning. To recover the original plain text, the hash must get deciphered.

The first two applications for demonstration are both exhibiting deciphering or “cracking passwords” with brute force. The goal for program A is to decipher passwords of length up to six bits that can include both numbers (0-9) and lowercase letters (a-z). The program generates random strings of lengths within the given range, hashes each string, and compares it with the targets list. The goal for program B is to decipher three valid BCH codes. It generates random strings of six digits and finds their parity checks. Verifies that the result is a valid BCH code, and if it is, compares its hash with the list of targets.

The possibility of randomly generating a particular password might seem unlikely, but both programs have accomplished their goals within a reasonable amount of time. Program A has broken all twelve passwords within a 24-minute run. Program B breaks all passwords for about three seconds. The downside is that different iterations may repeat some of the random strings. However, when generating consecutive strings (e.g., 000, 001), the position of the target alters the execution time - ''111" will take longer than "000".

*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)

1. Compare the produced hash with the targets list

Another method for deciphering hashes is creating a Dictionary. The dictionary records passwords and their hashes but is limited to a certain set of passwords. It is not a very convenient method, as it takes too much hard drive space. An improvement of the use of hard drive space is the Rainbow Tables method. Rainbow tables use minimal space by building chains of passwords and only recording their first and final strings.

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

Chains get constructed mainly by two types of functions – Hashing and Reduction. The hashing function is straightforward - it produces hashes of a given plain text. The reduction functions create new plain text from the hashes using a set of mathematical operations. The name “Rainbow” comes from the different reduction functions that get used throughout the table, as they 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 |

When building a large-scale table, the start of each chain is a randomly generated string within the desired length of passwords. The length of passwords, length of chains, and the number of chains should be global variables to allow easy accustoming to different tasks. The building of a rainbow table is very time expensive, as the table needs to have around 100 million passwords for a decent coverage.

*Pseudo Code for Building a Rainbow Table:*

1. Configure alphabet set, length of passwords, length of chains, and number of chains
2. Loop until all chains are created

* Generate a random password as the start of each chain
* Hash -> Reduce until the end of the chain is reached
* Check if the end of the chain is already in the table

. If yes, discard chain

. If no, add chain to table

Once the table gets built, deciphering is a much quicker process. There are two main functions – one goes down from the end to the start of the chain. The other goes from the current position back to the end of the chain, checking if that end is a valid key in the table. Keys are computationally faster to find than values, which is why the end of the chain gets recorded as a key in the table. Finding a valid key means finding the chain containing the targeted password. Reducing and hashing from the start while going down the chain will eventually find the plain text password.

*Testing of the Rainbow Table:*

|  |  |
| --- | --- |
| Test Case | Result |
| ad9966bd4b4a82e086b3f96fb4132cbf284efdb9 | Found (zzaapf) |
| dc551ddda247d9307a340a57ce2679f9fbf70b71 | Found(oidhje) |
| cadf11f0ed2fbdba016fa2935a466fe424e30565 | Found(poemda) |
| d9fba47a4be2b9c68349ecef481fc90fb10cca73 | Found(wichen) |
| 4eb3078d5c65f923173b5dc0a2be5af361362a61 | Found(okrmrw) |
| 907916580d665b4fe4323f316cb3acd2d814e6f4 | Found (uuuxxx) |
| a2b7caddbc353bd7d7ace2067b8c4e34db2097a3 | Unsuccessful |
| aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d | Unsuccessful |
| fbc8fae6b1390136c802d43f16890134bfe73df7 | Unsuccessful |

Logically, the more the table’s coverage increased, the more passwords it started to crack. With chain length of 5 000 and about 25 000 chains, it was able to find three of the passwords. When the number of chains increased to about 40 000, it was able to crack seven passwords. This testing concludes that the table has about 70% accuracy.

**Combination of Cryptography and Error correction**

Cryptography and Error correction can often collide. An example is a Biometric Authentication System provided by Maiorana and Ercole (2007). The method gets based on user-adaptive error-correcting codes securing the stored templates and canceling them if needed – for example, if a malware attack is detected. Error correction ensures the stored templates’ integrity. The application of distributed cryptography brings hierarchical key management services and improves the system’s fault tolerance. The testing of the system brings success - mostly for signature and iris identification.

**Combining Cryptography with different modules**

Cryptography can be combined with most aspects of computing and computer science. Encryption provides the necessary security that many applications need. In order to be ethical, applications using personal data must encrypt it in their database. Decryption takes part in the maintenance of the application by allowing developers to decipher the gathered data and study it.

The following example combines the modules of Cryptography, Artificial Intelligence, and Ethics. The use of AI chatbots has raised significantly over the past few years. They establish a better connection between app developers and customers. However, to be ethical, chatbots must never store conversations with customers in their database. In that case, how can developers know that the chatbot is doing its job well? One simple answer to this problem is encryption. With encryption, the chatbot can store customers’ conversations and still be ethical, provided that the conversations get encrypted before storage.

Algorithm for encryption of conversations:

**Word w + o + r + d sha1(w) + sha1(o) + sha1(r) + sha1(d)**

Every word in the conversation gets separated into chars. Each char is separately encrypted, while the total length of the word gets recorded as a passive key. These keys are only useful for putting the text back together after deciphering and have nothing to do with security. The decision to choose to hash individual characters got driven by the idea of faster decryption with brute force.

**Hash** **042dc4512fa3d391c5170cf3aa61e6a638f84342**

**8f84342042dc4512fa3d391c5170cf3aa61e6a63**

Shuffle to

All hashes from the same encryption algorithms have the same lengths, regardless of their plain text. For SHA-1, the length is 40 bits. Everyone with this knowledge could separate the big hash into individual hashes and decipher them. In this case, that would be particularly easy, as the hashed password space is 1 (individual chars). It is meant to be fast. To cope with that issue, bits of the hashes get shifted to certain key positions before getting integrated into the big hash.

*This is what a big hash looks like:*

7cf184f475c67a68d5828329ecb19349720b0cae58e6b3a4ba14a112e090df7fc6029add0f3555cc07c342be856e56f00e7f4347842e2e21b774e61d07c342be856e56f00e7f4347842e2e21b774e61d7a81af3e7d591a83c713f8761ea1efe93dcf36150ab8318a1bcaf639e678dd3d02e2b5c343ed4111ca73ab6…

The decryption algorithm for this application extracts the big hash from the database and separates it back to individual hashes of 40 bits each. The hashes gathered got previously shuffled using a certain key pattern. Using the same pattern, each bit gets back to its original place.

**042dc442512fa3d39143c5170cf3a8f8a61e6a63 Shuffled hash**

**042dc4512fa3d391c5170cf3aa61e6a638f84342 Original hash**

The whole idea of hashing individual chars comes from the need for fast decryption for big texts (such as chatbot conversations). Brute forcing 1bit strings is an extremely fast and easy implementation of deciphering, even on large text files. The algorithm decrypts letters while maintaining their sequence. Once all the letters are back to plain text, words get formulated by the passive key values - the length of every word in the original text. The system recognizes all punctuational signs and cannot get broken by entering unknown symbols with the text. It also successfully restores all capital letters and punctuation, keeping the data’s integrity intact.

Testing of this program has given a successful result - texts of all sizes get deciphered in seconds. The program proves suitable for the AI chatbot, as it will ensure fast access for maintenance through the bot’s conversations and its ethical use.

**Conclusion**

Cryptography and error correction applications already take part in our lives. Error correction ensures our data's integrity, whereas cryptography keeps it protected. The collision of their use is greatly beneficial, especially in automated identification using biometrics. The knowledge obtained on this module has been successfully applied to a big-scale project, combining aspects from two other modules as well.

(Total word count: 1 817)

Appendix:

Student number – 19017297

References:

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