Portfolio Assignment 1

CSI2108 – Cryptographic Concepts

James O’Grady (10561121)

Table of Contents

Table of Figures

Part 1 – Stream Cipher

Keystream Creation

The keystream creation process is divided into two separate parts, which I have named seeding and streaming. Within the algorithm, the seeding phase utilises a user-specified key to the StreamCipher object. This key hashed using the sha512 algorithm, which produces a 128-character hexadecimal representation. I then use the internal integer type to convert this hex representation into a base-10 integer, which is used as a seed for the random modules pseudo-random number generator (PRNG).

Setting the seed of the PRNG has the desirable property repeatability across platform and python version, which means that by seeding it with the user-specified key we can create an almost unique PRNG that is both cross-platform as well as predictable (assuming the recipient is able to specify the key). This makes it useful for the streaming part of the algorithm, in which the PRNG generates an integer for each character within the plaintext, which are then XOR'd together and added to the ciphertext stream.

Usage

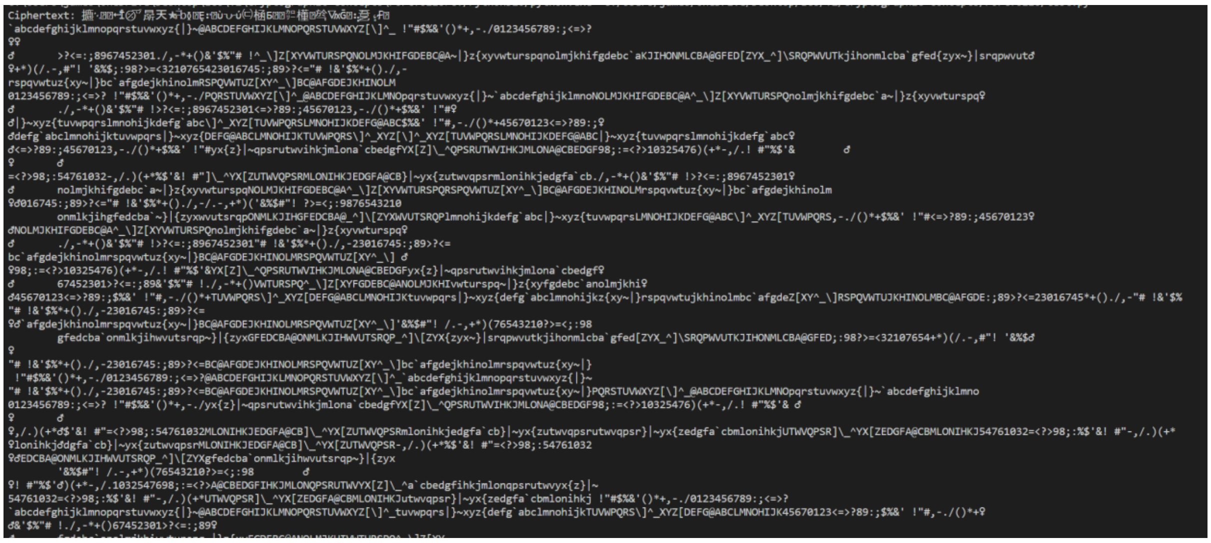
Assuming Alice has Python installed, it is very simple to encrypt and decrypt any text that she would like. Using the StreamCipher class within the StreamCipher.py file, she can initialise a class object with a key and immediately begin to encrypt or decrypt data at will. A single object can be used multiple times as there is no persistent data between methods apart from the key. To encrypt a piece of data, she may call the encrypt method, specifying the plaintext. This will return ciphertext that has been encrypted.

For Bob to decrypt the ciphertext, it is an almost identical process. Because the cipher uses XOR as its core encryption method, the encrypt method functions as both the encrypt and decrypt method. This works because A ^ B = C and C ^ B = A, where B is the common key value that is used to encrypt the data. For Bob to decrypt, he simply instantiates the object with the correct key, then calls the encrypt method on the given ciphertext.

Design Decisions

As part of my cipher, I wanted it to be very resistant to bruteforce attacks, whilst still being very easy to use. This meant that instead of relying on the user to choose a reliably secure password, I wanted to somehow use their input to generate a predictable yet still pseudo-random stream to use. This resulted in my use of the SHA512 algorithm, since it is guaranteed to produce an output of a 128-character hexadecimal representation, that when converted to base 10 will produce a usable seed with a length of 153-155 characters, depending on the input.

To remove the threat of a bruteforce, a different key for each character of the plaintext was used. This resulting key-space means there is ~2**14, or 16385, different keys possible for each character. Even if a bruteforce attack was attempted, due to the modular nature of XOR there are many different plaintext characters possible for each character in the ciphertext, meaning it’s almost impossible to re-piece together the message, as seen in the example show below:



```
Ciphertext: 擴[illegible]
`abcdefghijklmnopqrstuvwxyz{|}~@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_ !"#$%&'()*+,-./0123456789:;<=>?
♀♀
♂       >?<=:;8967452301./,-*+()&'$%"# !^_\]Z[XYVWTURSPQNOLMJKHIFGDEBC@A~|}z{xyvwturspqnolmjkhifgdebc`aKJIHONMLCBA@GFED[ZYX_^]\SRQPWVUTkjihonmlcba`gfed{zyx~}|srqpwvut♂
♀+*)(/.-,#"! '&%$;:98?>=<3210765423016745:;89>?<="# !&'$%*+()./,-
rspqvwtuz{xy~|}bc`afgdejkhinolmRSPQVWTUZ[XY^_\]BC@AFGDEJKHINOLM
0123456789:;<=>? !"#$%&'()*+,-./PQRSTUVWXYZ[\]^_@ABCDEFGHIJKLMNOpqrstuvwxyz{|}~`abcdefghijklmnoNOLMJKHIFGDEBC@A^_\]Z[XYVWTURSPQnolmjkhifgdebc`a~|}z{xyvwturspq♀
♂       ./,-*+()&'$%"# !>?<=:;8967452301<=>?89:;45670123,-./()*+$%&' !"#♀
♂|}~xyz{tuvwpqrslmnohijkdefg`abc\]^_XYZ[TUVWPQRSLMNOHIJKDEFG@ABC$%&' !"#,-./()*+45670123<=>?89:;♀
♂defg`abclmnohijktuvwpqrs|}~xyz{DEFG@ABCLMNOHIJKTUVWPQRS\]^_XYZ[\]^_XYZ[TUVWPQRSLMNOHIJKDEFG@ABC|}~xyz{tuvwpqrslmnohijkdefg`abc♀
♂<=>?89:;45670123,-./()*+$%&' !"#yx{z}|~qpsrutwvihkjmlona`cbedgfYX[Z]\_^QPSRUTWVIHKJMLONA@CBEDGF98;:=<?>10325476)(+*-,/.! #"%$'&        ♂
♀       ♂
=<?>98;:54761032-,/.)(+*%$'&! #"]\_^YX[ZUTWVQPSRMLONIHKJEDGFA@CB}|~yx{zutwvqpsrmlonihkjedgfa`cb./,-*+()&'$%"# !>?<=:;8967452301♀
♂       nolmjkhifgdebc`a~|}z{xyvwturspqNOLMJKHIFGDEBC@A^_\]Z[XYVWTURSPQRSPQVWTUZ[XY^_\]BC@AFGDEJKHINOLMrspqvwtuz{xy~|}bc`afgdejkhinolm
♀♂016745:;89>?<="# !&'$%*+()./,-/.-,+*)('&%$#"! ?>=<;:9876543210
        onmlkjihgfedcba`~}|{zyxwvutsrqpONMLKJIHGFEDCBA@_^]\[ZYXWVUTSRQPlmnohijkdefg`abc|}~xyz{tuvwpqrsLMNOHIJKDEFG@ABC\]^_XYZ[TUVWPQRS,-./()*+$%&' !"#<=>?89:;45670123♀
♂NOLMJKHIFGDEBC@A^_\]Z[XYVWTURSPQnolmjkhifgdebc`a~|}z{xyvwturspq♀
♂       ./,-*+()&'$%"# !>?<=:;8967452301"# !&'$%*+()./,-23016745:;89>?<=
bc`afgdejkhinolmrspqvwtuz{xy~|}BC@AFGDEJKHINOLMRSPQVWTUZ[XY^_\] ♂
♀98;:=<?>10325476)(+*-,/.! #"%$'&YX[Z]\_^QPSRUTWVIHKJMLONA@CBEDGFyx{z}|~qpsrutwvihkjmlona`cbedgf♀
♂       67452301>?<=:;89&'$%"# !./,-*+()VWTURSPQ^_\]Z[XYFGDEBC@ANOLMJKHIvwturspq~|}z{xyfgdebc`anolmjkhi♀
♂45670123<=>?89:;$%&' !"#,-./()*+TUVWPQRS\]^_XYZ[DEFG@ABCLMNOHIJKtuvwpqrs|}~xyz{defg`abclmnohijkz{xy~|}rspqvwtujkhinolmbc`afgdeZ[XY^_\]RSPQVWTUJKHINOLMBC@AFGDE:;89>?<=23016745*+()./,-"# !&'$%
"# !&'$%*+()./,-23016745:;89>?<=
♀♂`afgdejkhinolmrspqvwtuz{xy~|}BC@AFGDEJKHINOLMRSPQVWTUZ[XY^_\]'&%$#"! /.-,+*)(76543210?>=<;:98
        gfedcba`onmlkjihwvutsrqp~}|{zyxGFEDCBA@ONMLKJIHWVUTSRQP_^]\[ZYX{zyx~}|srqpwvutkjihonmlcba`gfed[ZYX_^]\SRQPWVUTKJIHONMLCBA@GFED;:98?>=<32107654+*)(/.-,#"! '&%$♂
♀
"# !&'$%*+()./,-23016745:;89>?<=BC@AFGDEJKHINOLMRSPQVWTUZ[XY^_\]bc`afgdejkhinolmrspqvwtuz{xy~|}
 !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
"# !&'$%*+()./,-23016745:;89>?<=BC@AFGDEJKHINOLMRSPQVWTUZ[XY^_\]bc`afgdejkhinolmrspqvwtuz{xy~|}PQRSTUVWXYZ[\]^_@ABCDEFGHIJKLMNOpqrstuvwxyz{|}~`abcdefghijklmno
0123456789:;<=>? !"#$%&'()*+,-./yx{z}|~qpsrutwvihkjmlona`cbedgfYX[Z]\_^QPSRUTWVIHKJMLONA@CBEDGF98;:=<?>10325476)(+*-,/.! #"%$'& ♂
♀       ♂
♀,/.)(+*♂$'&! #"=<?>98;:54761032MLONIHKJEDGFA@CB]\_^YX[ZUTWVQPSRmlonihkjedgfa`cb}|~yx{zutwvqpsrutwvqpsr}|~yx{zedgfa`cbmlonihkjUTWVQPSR]\_^YX[ZEDGFA@CBMLONIHKJ54761032=<?>98;:%$'&! #"-,/.)(+*
♀lonihkj♂dgfa`cb}|~yx{zutwvqpsrMLONIHKJEDGFA@CB]\_^YX[ZUTWVQPSR-,/.)(+*%$'&! #"=<?>98;:54761032
♀♂EDCBA@ONMLKJIHWVUTSRQP_^]\[ZYXgfedcba`onmlkjihwvutsrqp~}|{zyx
        '&%$#"! /.-,+*)(76543210?>=<;:98        ♂
♀! #"%$'♂)(+*-,/.1032547698;:=<?>A@CBEDGFIHKJMLONQPSRUTWVYX[Z]\_^a`cbedgfihkjmlonqpsrutwvyx{z}|~
54761032=<?>98;:%$'&! #"-,/.)(+*UTWVQPSR]\_^YX[ZEDGFA@CBMLONIHKJutwvqpsr}|~yx{zedgfa`cbmlonihkj !"#$%&'()*+,-./0123456789:;<=>?
`abcdefghijklmnopqrstuvwxyz{|}~@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_tuvwpqrs|}~xyz{defg`abclmnohijkTUVWPQRS\]^_XYZ[DEFG@ABCLMNOHIJK45670123<=>?89:;$%&' !"#,-./()*+♀
♂&'$%"# !./,-*+()67452301>?<=:;89♀
```

Figure 1 A Crude Bruteforce Attempt

Due to the number of keys used in the cipher, frequency analysis is not as effective as it normally is against XOR ciphers. This is generally effective because once the length of the key is known, the attacker is essentially solving K-many Caesar ciphers, where K is the length of the key. However, as demonstrated above, solving for a letter of the key is not very effective when it is indistinguishable from the random possibilities that it could also be.

The cipher also maintains punctuation and case when encrypting and decrypting, as well as supporting many languages such as English, Chinese and Russian – The only ones that have been tested, though theoretically it can handle anything in Unicode. There is no real message length cap that could be found, it has been tested with several megabytes of data and several tens of thousands of characters and handles it perfectly fine.

Improvements

Although the Mersenne Twister (the underlying PRNG in Python) is very useful due to its repeatability, it is not cryptographically secure. This means that with a long enough message, the pseudo-random numbers that are generated will fall into a uniform distribution if the seed does not provide enough entropy. I have offset this by producing a seed using sha512, however it still depends on the entropy given within the initial key set by the user. Observing a sufficient number of iterations will allow an attacker to guess all future iterations, although if they are able to observe iterations in the PRNG then the cipher is already broken, since they can directly decode the message.

When compared to Trivium, Trivium is far more secure due to its initialisation vector (IV), non-linearity and three shift registers. This could serve as an improvement to my stream cipher, since the introduction of non-linearity or a certain number of shift registers could introduce far more cryptographic security. Unfortunately, this price of this is that Trivium requires a form of "warm up phase" before generating a useable stream of bits. Within my cipher, the bits are immediately usable, without wasting compute power to generate and initialise randomness within the shift registers. An IV stream into the input of my cipher could increase the entropy and make the cracking the cipher far more difficult.

Part 2 – Block Cipher

Structure

Similar to my stream cipher, the block cipher begins with a key hashed using sha512. In the block cipher however, there is also a randomised IV that serves as an additional layer of security.

Within the Block Cipher itself, there is an IV injection stage, a substitution stage, a permutation stage and finally a key addition state. The resulting ciphertext is then hashed and used as the IV for the next 8-character block. Once all the blocks have been encrypted, the process is repeated for a total of 4 times for added security.

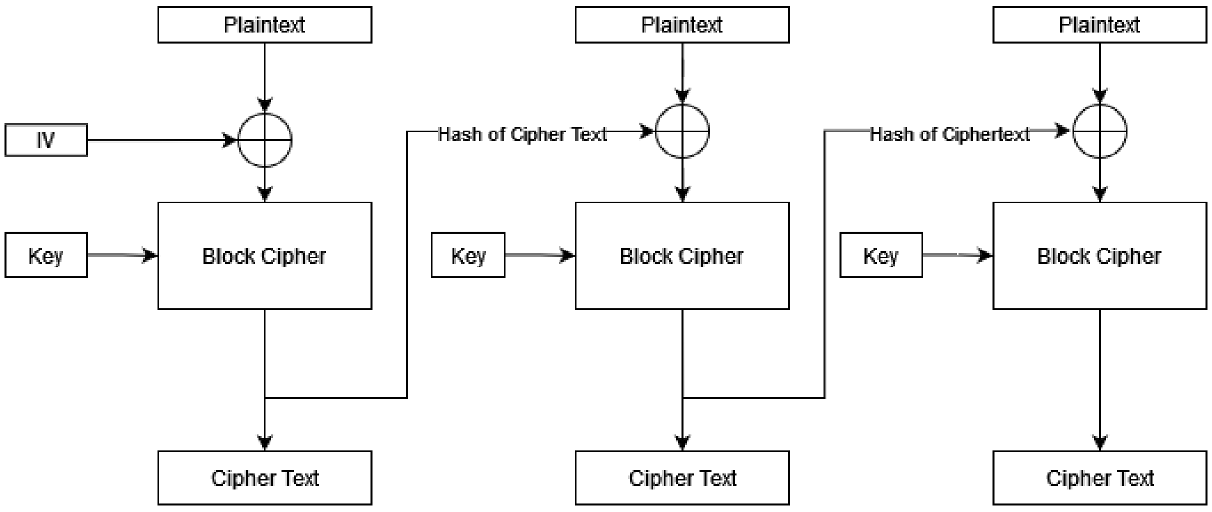


Figure 2 Block Cipher Structure Diagram

Stages

IV Injection

In this stage, the text input is XOR'd with either the IV or the hash of the previous blocks output, depending on the stage of encryption.

Substitution

Within this stage, a seeded PRNG generates a random value from 0:16384, which is then added to the current characters ordinal value to substitute it for another character. This can easily be reversed in decryption by subtracting the same random value, resulting in the original plaintext. Without that random value however, it is almost impossible to know what character it was originally. This adds confusion to the system, obscuring the relationship between plaintext and ciphertext.

Permutation
This stage first generates a random permutation order for the letters to be shifted in. This is a block length character string that describes the place in which letters should be moved to within the block. As an example, for a block length of 8, the permutation “12046357” applied to “abcdefgh” would result in “cabfdeh”. This can be easily reversed by generating the reverse permutation string, which when applied in the same manner will produce the original text. This helps to diffuse the system, spreading the influence of each symbol over many ciphertext symbols to hide statistical properties.

Key addition
Key addition is done throughout the Substitution and Permutation stages in the seeding of the PRNGs, however within this stage the key is directly XOR’d with the ciphertext before it is returned from the method. This adds more confusion to the system, as well as giving the key a greater impact on the cipher.

Mode of Operation
I have implemented the Cipher-Block Chaining (CBC) mode of operation. This mode implements an IV to introduce randomness, as well as chains all the blocks together so that they cannot easily be identified and cannot be substituted, solving the issues that are found in simple Electronic Code Book (ECB) operation mode ciphers. It is also suitable for multiple rounds of encryption, affording much greater security then a single round that could potentially be compromised due to lack of confusion / diffusion.

Security Evaluation
Improvements for security:

- Multiple Rounds
- An IV Value
- A Salt Value for the hash functions
- A larger minimum key size

Multiple rounds have been implemented into the block cipher at a standard of 4 rounds. This helps to prevent side channel attacks and cryptoanalysis taking place on the algorithm, as well as generally adding a higher level of confusion and diffusion.

The IV Value has also been implemented due to the CBC mode of operation. This value increases the initial randomness of the first block, which then cascades into the next N-many block depending on the size of the message. This works well in combination with the multiple rounds, since the IV generated from the hash of the previous block will always change, except for the first block, since the blocks content is changing alongside it as the effects cascade further.

A salt value for the hash functions that are used to generate both the key and nonce (IV) values will help to prevent a rainbow table attack from affecting the cipher. A rainbow table is a precomputed table of hash values that allows an attacker to quickly look up a hash and see its corresponding plaintext value. This could severely affect security, as IVs are generally considered public information, meaning that if the key is taken by the attacker they could easily decode and change the message at will. Multiple rounds will not defend against this since it will be a valid way of decrypting the cipher text.

A larger minimum key size can prevent a rainbow table attack, as well as providing a greater level of entropy for the key system. If a key is small in length, it increases the likelihood that it has either been precomputed and stored or will be brute forced very quickly should anyone attempt to. Likewise, if a key is large, it adversely affects the attackers' prospects of a brute force or other side channel attacks. This affects the upper bounds of security that the cipher can reach.

The confusion of the system is very high. Changing a single bit of the user-inputted key will change all the resulting output thanks to the hash function that serves as the key generation and the CBC hash chaining that means a change to one block will affect the plaintext input to the next.

Comparison to AES

AES	My Block Cipher
Uses permutation and substitution	Uses permutation and substitution
Has multiple modes of operation	Utilises only CBC
Does not provide proof that ciphertext is tamper-free	Does not provide proof that ciphertext is tamper-free
Cannot be parallelised to go faster	Cannot be parallelised, still very fast however (0.0008s avg for a large input)
Uses different round numbers for different sized keys	Can use different round counts while maintaining key length
Utilises matrixes and substitution boxes	Makes use of PRNG (less secure)
Fixed block sizes	Can change block size depending on specifications

Part 3 – Recommendation

In my opinion, my block cipher is more secure than my stream cipher. Although both are resistant to brute force attacks, the multiple rounds of encryption, the IV addition and the substitution / permutation contained within the block make it far more resistant to other side channel attacks then my stream cipher implementation. It is however slower, with an average run time of 8.0 x 10^-4s encrypting and 6.5 x 10^-4s decrypting compared to 4.8 x 10 ^-5s encrypting and 5.6 x 10^-5 decrypting using the stream cipher.

Overall, I would recommend the block cipher to Alice. It has inbuilt mechanisms to try and prevent substitution attacks that might try to change the room number, and due to the CBC operating mode it is hard for an attacker to identify where to change exactly. Although there is a difference in speed between the two ciphers, it is negligible compared to the speed of the data transfer between Alice and Bob, such as Wi-Fi or SMS. If speed is really that important, and the message is short such as the current scenario, then the stream cipher could suffice but for longer messages or for better general security, the block cipher is the better option.