**Assignment 2**

-yixi rao u6826541

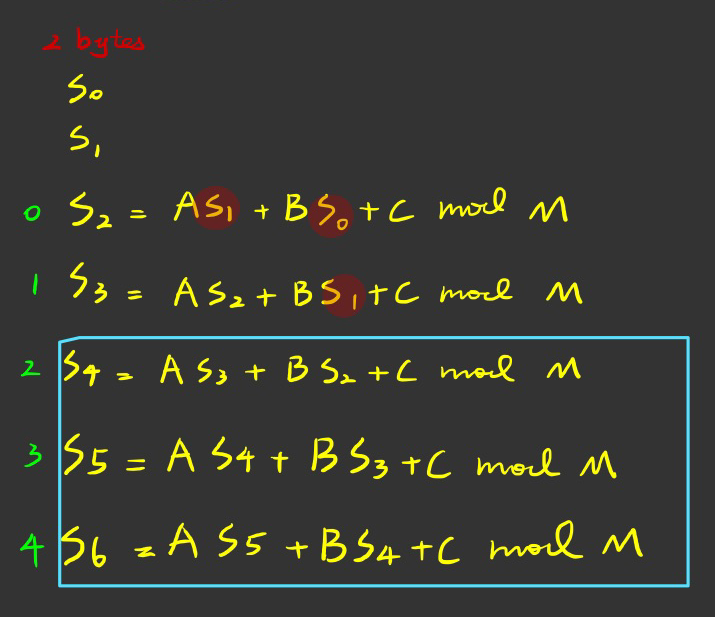
**Question 2 ELCG Cipher**

I first analyze the ELCG, the and are secret seed since the stream starts with and also the A and B are secret. So, in order to get the value of all the secret keys, we need to explore the equation , there are three unknows coefficients in the equation (A, B, C), therefore we need at least three stream keys equation to solve it. We also need to know the value of three consecutive stream keys and the modular M. In order to get the three consecutive stream keys, the property of stream cipher can be used, which is and also the fact “every random number is encoded as two bytes in the key stream” can be used. In general, if we can get the first 10 bytes of the plaint text and the first 10 bytes of the cipher text, then we can use linear equation with three unknows to solve it and finally we can use the keys to decrypt the file.

* **Vulnerability**: Bad cryptographic properties due to the linearity of most PRNGs and the Output can be reproduced and can be predicted.

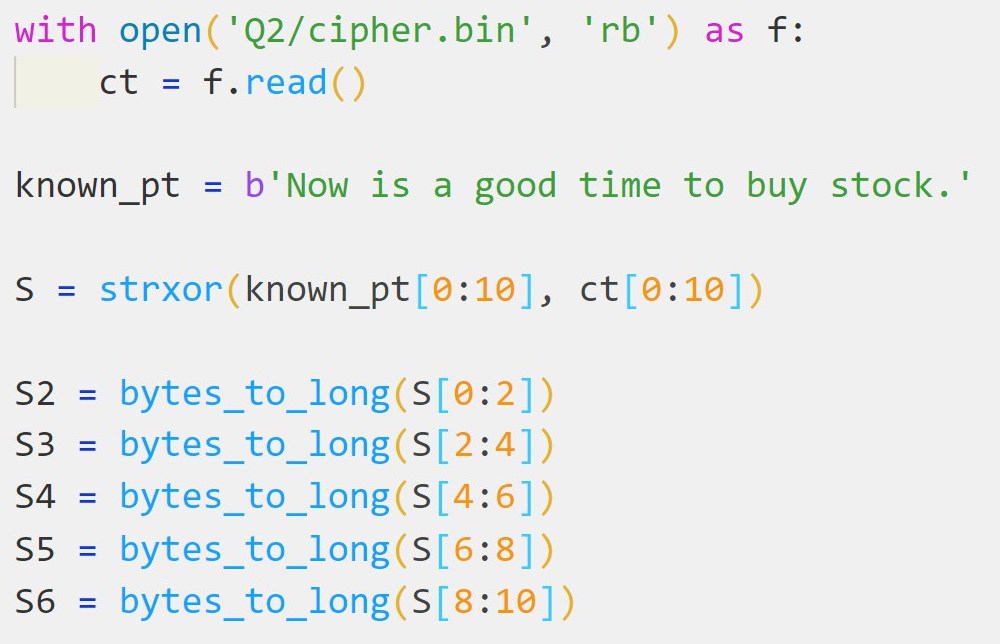
We know that the stream cipher is generated using an extended version of linear congruential generator, so this is a Pseudorandom Number Generator which has the property that the PRNGs are not random in a true sense because they can be computed and are thus completely deterministic by using the linear equation it defined. As a result, if the attacker knows 10 bytes of the plaint text and 10 bytes of the cipher text, then the attacker can predict the future number correctly.

The exploitation is divided into three steps, the first step is to get the first 5 stream cipher using the given hint.txt and cipher.bin, the reason why I get the first 5 stream cipher rather than 3 stream cipher is that the first two stream cipher contains S1 and S0 that we do not know so it cannot be used to calculate the linear equation (formular in graph 2.1)

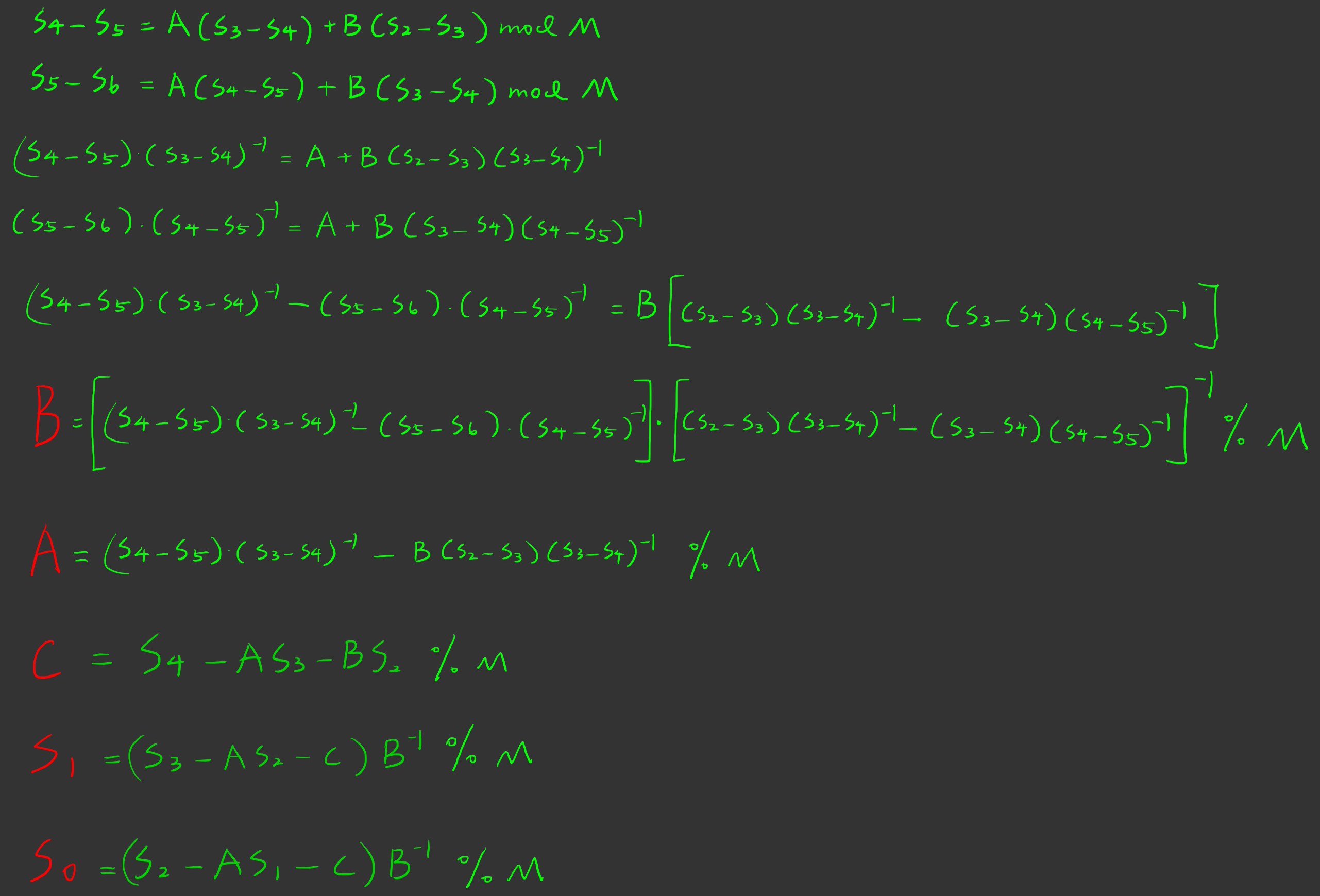


graph 2.

Next, we can use the property of to get the first 5 stream cipher using python script.

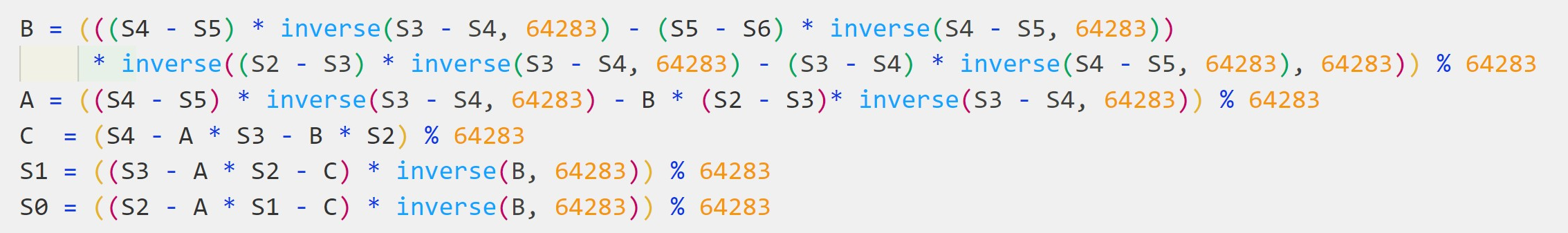


The second step is to calculate the secret key A, B, C, S0, S1 by using the S4, S5, S6 to solve the linear equation with three unknows, and the derivative process of this equation is shown is graph 2.2



graph .2

The python script of second step is shown below.



In the last step we only need to use the given elcgcipher.py file and the keys obtained in second step to decrypt the cipher.bin and get the flag. The full python script is provided in appendix.

 **Question 3 CTR MAC**

In this question, we are required to create a MAC forgery attack by using the fst.bin and the MAC value of it. We first explore the CTRMAC algorithm, it first padding to the fst.bin to create , and then using the AES-128 to encrypt it using CTR mode to gain , next, it rotates each block of by using the index of it, and the will be obtained. The equation of is where means the will be rotated to bytes to the right. And finally, all the blocks in are XORed to get the final MAC:

Now we explore the fst.bin and the mac1.txt, we find that it is exactly characters, which means it will have 18 blocks. So, if we can find a second pre-image of the fst.bin that has the same MAC value, then we can create a forgery. To find the snd.bin, we need to explore the relation of deeply, we first expand the equation as:

And the is:

Because the rotation is the byte-wise rotaion

**Question 4 BBC (Bad Block cipher Chaining)**

In this question, we are required to recover the flag in cipher2.bin that is encrypted by the BBC by using the known plain text and cipher text. In the bbc\_encrypt function, it first defines an AES cipher to encrypt the plain1.txt in ECB mode, and then the cipher text is obtained by iteratively XOR the previous cipher blocks with the encrypted plain text. This is not like traditional CBC, which first XOR the plaintext block with the previous ciphertext block. The difference is shown below.

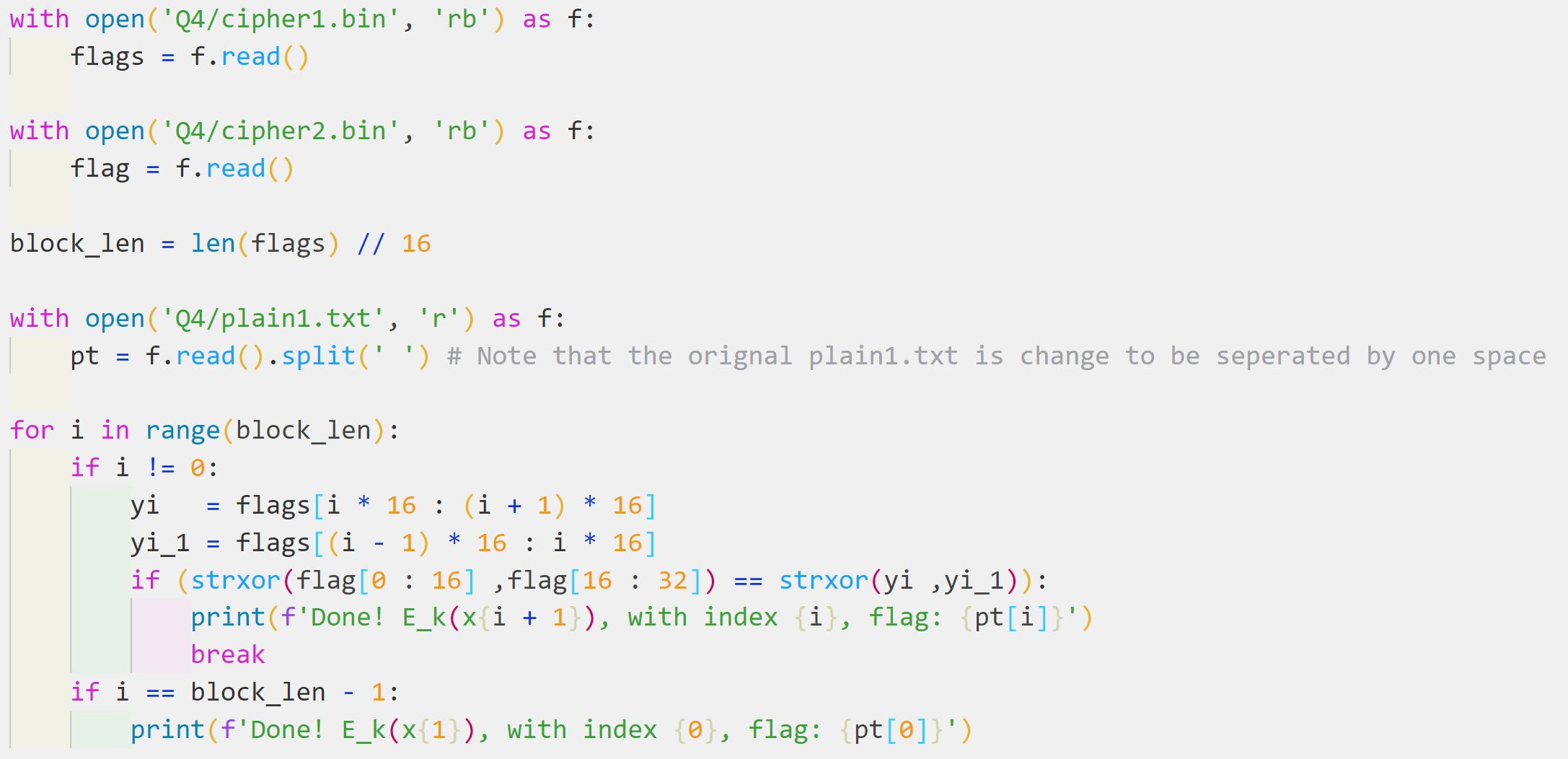
Note that we can use the property of to get the AES encryption of the plaintext blocks of this form . And all the are known so we can easily know all the bytes of the encrypted plain1.txt by using the cipher1.bin. The next exploration step is trying to figure out the IV since we need to compare each with the 16 bytes in cipher2.bin, this is reasonable because we know that the two binary files is encrypted with the same key and IV. However, this idea is abandoned because finding the IV is impossible and the cipher2.bin is not 16 bytes it is 48 bytes and the flag is in the second block of cipher2.bin, so the composition of it is

We can easily get the , by using the and like . Now, the idea is clear, we know all the except the since the IV is unknown, and we also know the , therefore we can compare each with , if they are match then we know the index of the answer flag. And if it cannot find any match then we know the will be the correct encrypted flag.

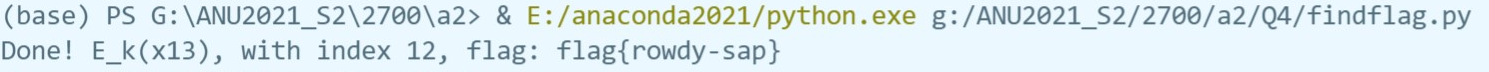
* **Vulnerability**: As discussed in the exploration above, this modified CBC can be turned into ECB easily by stripping away the previous ciphertext block from each block of the ciphertext. So, all the vulnerabilities of the ECB will be applied to this BBC. Such as
* ECB encrypts highly deterministically
* Identical plaintexts are mapped to identical ciphertexts
* an attacker recognizes if the same message has been sent twice
* plaintext blocks are encrypted independently of previous blocks

So once a particular plaintext to ciphertext block mapping is known, a sequence of ciphertext blocks can easily be manipulated, the practical example is the electronic bank transfer attack. For this question, if the BBC applied in the real-world application, the BBC will be more vulnerable because the IV is generally considered public so we don’t need the extra blocks as in cipher2.bin to compare the ciphertext blocks. The attacker just needs a 16 bytes flag encryption with the same key and IV to get the flag.

The attack strategy is then clear, we just need to go through all the 16-bytes blocks (except the first one) in the cipher2.bin, for each loop, it first calculates the by using the previous cipher text block and the current one, and compare it with the , if they are match, the flag will be printed using the index and break the loop. If it reaches the end of cipher2.bin, it will return the first index as the correct one. The Python script and result is shown below (full script see appendix).



graph code



graph result

**Question 5 Bad AES**

From the problem description, we know this modified AES is an AES without the diffusion layer, and the sample.txt is encrypted using this Bad AES in ECB mode, so every block in simple.txt is encrypted independently from other blocks. The flag.enc is encrypted using Bad AES with the same key that was used to encrypt sample.txt. My initial idea was to write the equations of the relation of the plaintext block and the ciphertext block, the equations is:

The idea is first XOR the and the to get the

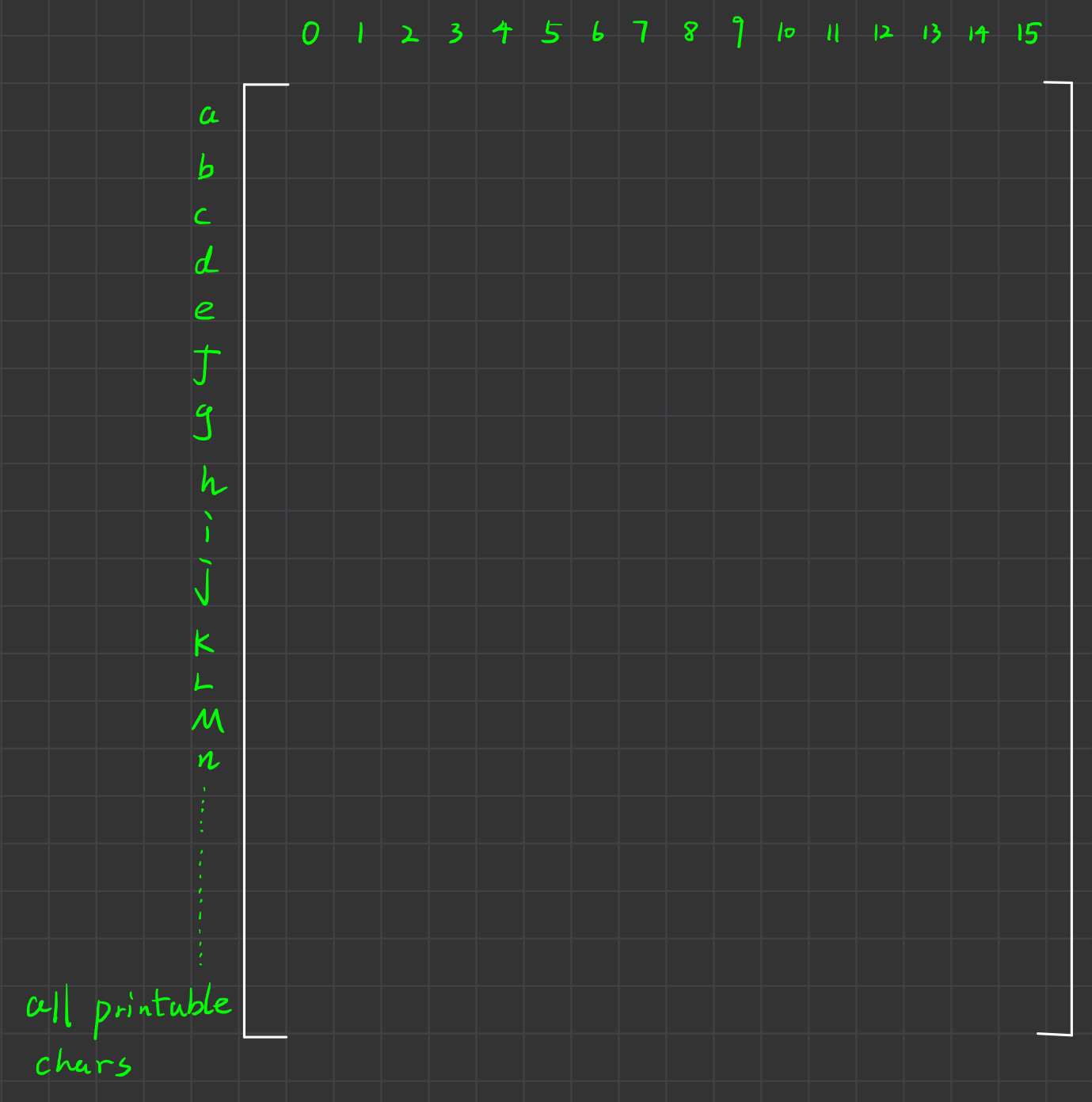
And then to find out whether this property , if this is true then we can eliminate the key from outside to inside, and eventually the only unknow is the flag plaintext . However, this idea is not applicable because byte substitution layer is the nonlinear elements of AES which means so this initial idea is abandoned. Now I consider what will happen on the AES without the Diffusion layer, the function of the diffusion layer is to hide the statistical properties of the plaintext, and the sample.txt is encrypted using ECB mode, which means we might find some mappings from the ciphertext to the plaintext and using that mappings to replace the ciphers in flag.enc and get the plaintext flag. I do some experiment with the sample.txt and sample.enc. some observations are gained.



This observation reveals that different blocks of 16 bytes “----------------” encryption are the same. And the ‘-’ character in different positions in the block will have different encryption but the ‘-’ character in other blocks with the same position will have the same encryption. This is because the key used is the same in different blocks. Now we can make a conclusion:

* the same character in different positions () in the blocks will have different ciphertext.
* the same character in the same position () among different blocks will have same ciphertext.

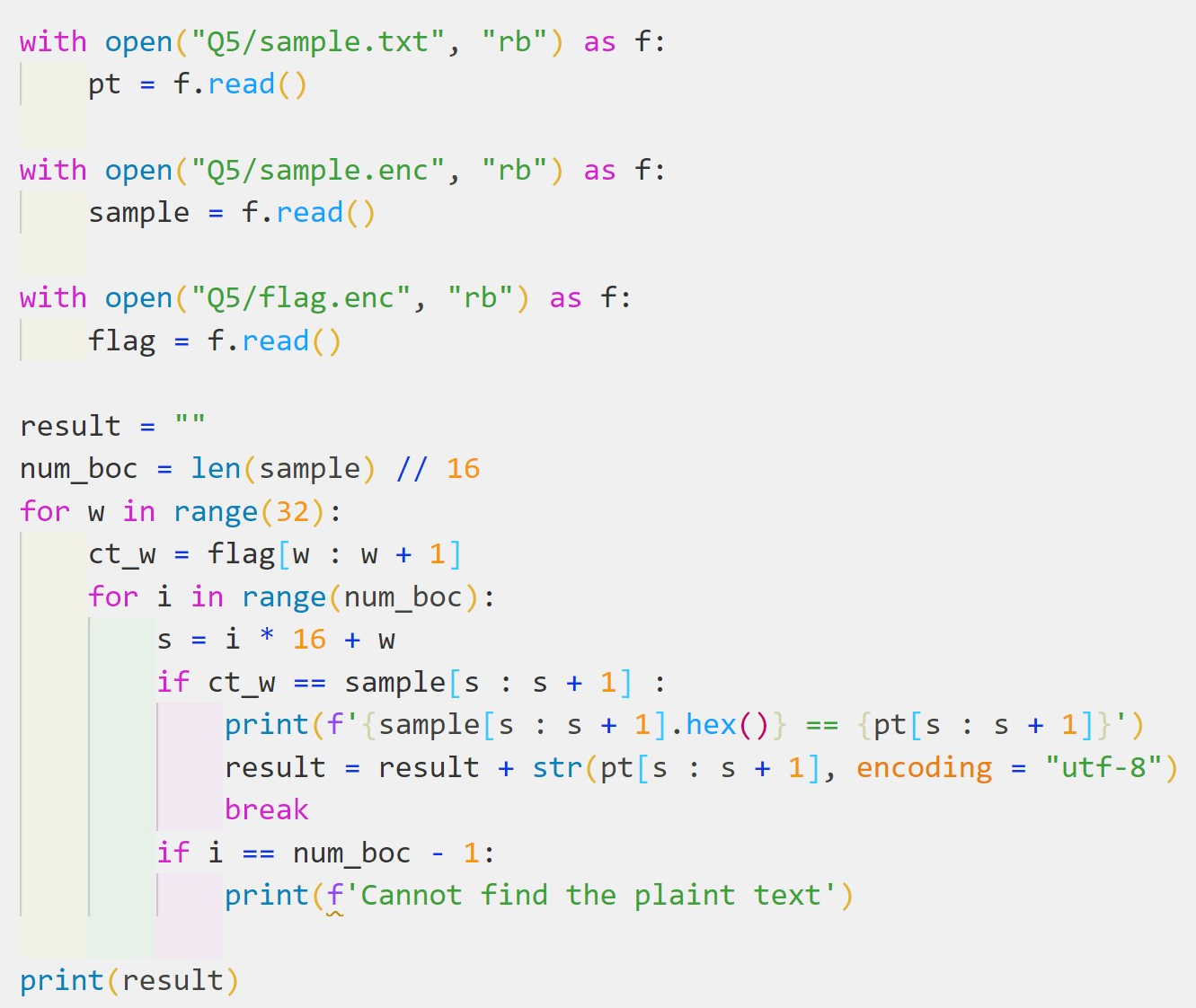
So, the attack strategy will be clear, we just need to build a look-up table by using the sample.txt and sample.enc where the column is the index, the row is the different printable characters (graph 5.1). And then using the look-up table to find the correct flag.

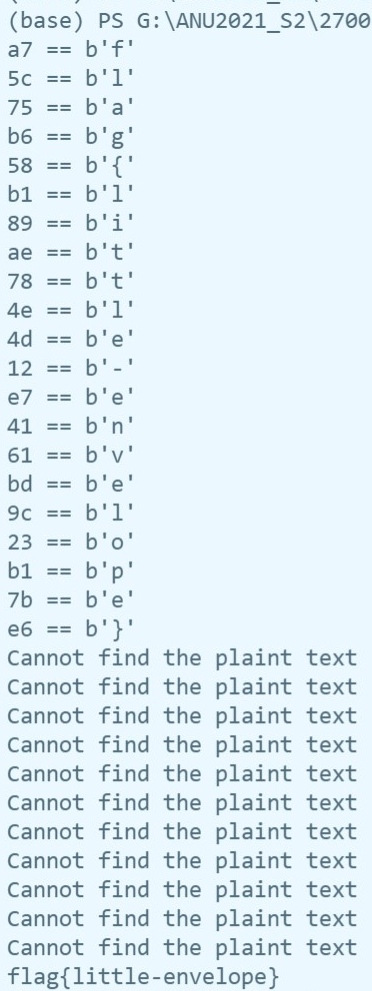


graph 5.1

* **Vulnerability**: the lack of diffusion layer will not have the influence of one plaintext symbol is spread over many ciphertext symbols and expose the statistical properties of the plaintext. Besides, the ECB mode AES encryption will help the attacker find the mapping of each printable characters to the encrypted one. As a result, the attacker can perform the Substitution Attack. If the attacker manages to get a big plaintext and ciphertext pair, then the attacker can build a look-up matrix and he/she can decrypt all the secret message encrypted with this Bad AES using the same key. This problem can be avoided by using the key once but this is expensive and not practical.

For the attack strategy, it is not necessary to build the look-up table because we only care about the characters of the flag and building the table is expensive. So, the exploitation steps are to create a for loop to go through all the bytes in the flag.enc, for each round, we want to find the plaintext of the byte in this round, let’s call it . In every round, another for loop is created to go through all the plaintext and ciphertext of the sample but we only care about the specific bytes of it, let’s call it , now we want to check whether , if this is true, we find the correct flag’s character and will print it to the screen. If it goes through all the sample and cannot find the mapping, the warning message will be printed. We are lucky that all the encrypted can be matched because the database or corpus (sample.txt) is big enough to matches all the different flag’s characters in different 16 positions. The code and the result are shown below. Full code is in appendix.





**Question 6 CFB Hash**