**GROUP 3**

**Part 1**

**Key**

iceman

**Plaintext**

Hope: What did you do to this booze? That's what we'd like to hear. Bejees, you done something. There's no life or kick in it now. (He appeals mechanically to Jimmy Tomorrow.) Ain't that right, Jimmy?

Jimmy: (More than any of them, his face has a wax-figure blankness that makes it look embalmed. He answers in a precise, completely lifeless voice, but his reply is not to Harry's question, and he does not look at him or anyone else.) Yes. Quite right. It was all a stupid lie--my nonsense about tomorrow. Naturally, they would never give me my position back. I would never dream of asking them. It would be hopeless. I didn't resign. I was fired for drunkenness. And that was years ago. I'm much worse now. And it was absurd of me to excuse my drunkenness by pretending it was my wife's adultery that ruined my life. As Hickey guessed, I was a drunkard before that. Long before. I discovered early in life that living frightened me when I was sober. I have forgotten why I married Marjorie. I can't even remember now if she was pretty. She was a blonde, I think, but I couldn't swear to it. I had some idea of wanting a home, perhaps. But, of course, I much preferred the nearest pub. Why Marjorie married me, God knows. It's impossible to believe she loved me. She soon found I much preferred drinking all night with my pals to being in bed with her. So, naturally, she was unfaithful. I didn't blame her. I really didn't care. I was glad to be free--even grateful to her, I think, for giving me such a good tragic excuse to drink as much as I damned well pleased. (He stops like a mechanical doll that has run down. No one gives any sign of having heard him. There is a heavy silence. Then Rocky, at the table in the bar, turns grouchily as he hears a noise behind him. Two men come quietly forward. One, Moran, is middle-aged. The other, Lieb, is in his twenties. They look ordinary in every way, without anything distinctive to indicate what they do for a living.)

**Explanation**

The first step in deciphering the ciphertext was to first find the key size. If you are aware of the key size, the ciphertext can be divided into n columns, where n is the length of the key. For the characters in each column, they would have been encrypted by the same byte from the key. Thus the idea is to then solve these smaller individual ciphers, then to combine the results together to form the plaintext.

To get the key length, a Kasiski's cryptanalysis test was performed on the ciphertext. The principle behind this test is to examine the character distances between repeated substrings of size 3 and 4 in order to find common distances. More specifically, we were trying to find common multiples of the key length. Once a list of common multiples were found, the list was sorted to find the most frequently occurring divisor. Note that we omitted key lengths of 1, 2, and 3 because these key lengths are quite trivial. We were looking at other methods to determine the key length, such as using the Index of Coincidence, however these methods are harder to implement and may not give an accurate result.

As stated before, first the character distances of repeated substrings were counted. Then this list was sorted based on the most frequently occurring character to give us the key length (excluding trivial key lengths). Once a key length is acquired, the ciphertext was divided into n columns where n is the length of the key. Each column is encrypted by one byte of the key, essentially being a Caesar cipher. At this point, we have the cipher bytes, now we needed some candidates for key bytes. Since the key is a sequence of printable ASCII characters, we can narrow down the set of possible key bytes from 256 to 96. This is because the upper bits of the keys must be printable, thus they range from 0x20 to 0x70 (and the lower 4 bits may range from 0x00 to 0x0F). So at this point, we have narrowed down our search space by more than half. For each key in the set of possible keys, we try to first translate the cipher bytes into printable ASCII characters. To translate a cipher byte to an ASCII character, we use the fact we are given the cipher byte and the current test key byte, we can traverse the map to find a matching plaintext byte. We then check to see if this plaintext byte is in the range of printable ASCII. If it is not, then the key is wrong and we can stop and try a different key. This check saves quite a bit of computational time. Once we have successfully translated the ciphertext into some printable ASCII, we then need to check if these sequences of characters are indeed English sentences. To check if it English, the frequencies of each character are counted. Then, if the most frequent characters are spaces or lowercase alphabet characters, then it is most likely an English sentence. We noticed that some invalid strings contained large amounts of symbols and punctuation, so it was enough to filter out strings based on this fact. For this ciphertext, it was not necessary to check for frequencies of vowels. At this point, we have successfully deciphered one column/bucket using our key byte, thus we can return and repeat for all columns/buckets following the same procedure.

**Sources:**
https://crypto.interactive-maths.com/kasiski-analysis-breaking-the-code.html

## Part 2

Our team was not able to successfully decrypt the message, key, or the file type. The main techniques that we tried was to only analyze the first 2000 bytes of the ciphertext to try to figure out what the file type is. To find the key length, we used the same key length function which was defined in part 1, but we had multiple candidates for the key lengths (namely 9 and 15). Trying out these key lengths, we divided the ciphertexts into n columns/buckets, where n = key length. From this point, we could generate possible sets of candidate plaintexts for each column/bucket. However, we could not find good ways to filter out some of these candidate plaintexts into anything usable. Referring to popular file formats, we tried to see if any of these decrypted strings had matching bytes to formats such as PDF, PNG, BMP, ISO, etc., but we could not successfully find anything matching. We also tried looking at the tail of the file to see if it could give some insight on the nature of the file, however, we could not find anything of use.

## Part 3

**Key = “bayo”**

**File1 (560 bytes)**

- **File1ecb (568 bytes)**
  - **Pattern:** Every 8 bytes is repeated. This is because every 8 characters(8 bytes) in the plaintext file is repeated and the ecb mode of operation splits the file into a block 8bytes and encrypts each block(which will be identical because the plaintext is always the same).

  - **Size:** the encrypted file is 8bytes more. This is the padding added because the ecb mode of operation requires padding if the file ends on a block boundary. The message is 560 bytes which is an exact multiple of 8, so when the ecb mode of operation divides the file into 8btyes, the last block would be exactly 8bytes(which is the block boundary), hence an extra padding block would be added.

- **File1cbc (568 bytes)**
  - **Pattern:** There is no observable pattern, this is because the cbc mode operation XORs each block of plaintext, with the previous cipher text before encrypting.(each previous cipher text will be different which will lead to different cipher texts overall)
  - **Size:** the encrypted file is 8bytes more. This is the padding added because the cbc mode of operation requires padding if the file ends on a block boundary. The message is 560 bytes which is an exact multiple of 8, so when the ecb mode of operation divides the file into 8btyes, the last block would be exactly 8bytes (which is the block boundary), hence an extra padding block would be added.

- **File1cfb (560 bytes)**
  - **Pattern:** There is no observable pattern, cfb uses the entire output of the block cipher(the plaintext XOR the output of the block cipher) to encrypt the next block.(each previous cipher text will be different which will lead to different cipher texts overall)
  - **Size:** the file size is the same because no padding is required

- **File1ofb (560 bytes)**
  - **Pattern:** There is no observable pattern, because ofb generates key stream blocks which are XORed with the plain text blocks to get the cipher text.
  - **Size:** the file size is the same because no padding is required

**File2 (600 bytes)**

- **File2ecb (608 bytes)**
  - **Pattern:** Every 24 bytes is repeated. This is because every 12 characters(12 bytes) is repeated, the ecb mode of operation splits the file 8 bytes and for this file after 3 each splits, bytes of the plaintexts repeat(become identical).(3 splits = 3x8 = 24bytes).
  - **Size:** the encrypted file is 8bytes more. This is the padding added because the ecb mode of operation requires padding if the file ends on a block boundary. The message is 560 bytes which is an exact multiple of 8, so when the ecb mode of operation divides the file into 8btyes, the last block would be exactly 8bytes(which is the block boundary), hence an extra padding block would be added.

- **File2cbc(608 bytes)**
  - **Pattern:** There is no observable pattern, this is because the cbc mode operation XORs each block of plaintext, with the previous cipher text before encrypting.(each previous cipher text will be different which will lead to different cipher texts overall)
  - **Size:** the encrypted file is 8bytes more. This is the padding added because the cbc mode of operation requires padding if the file ends on a block boundary. The message is 560 bytes which is an exact multiple of 8, so when the ecb mode of operation divides the file into 8btyes, the last block would be exactly 8bytes (which is the block boundary), hence an extra padding block would be added.

- **File2cfb(600 bytes)**
  - **Pattern:** There is no observable pattern, cfb uses the entire output of the block cipher(the plaintext XOR the output of the block cipher) to encrypt the next block.(each previous cipher text will be different which will lead to different cipher texts overall)
  - **Size:** the file size is the same because no padding is required

- **File2ofb(600 bytes)**
  - **Pattern:** There is no observable pattern, because ofb generates key stream blocks which are XORed with the plain text blocks to get the cipher text.
  - **Size:** the file size is the same because no padding is required

**File3 (600 bytes)**

- **File3ecb (608 bytes)**
  - **Pattern:** There is no observable pattern because every character is randomized, so each split into 8 bytes would be different.
  - **Size:** the encrypted file is 8bytes more. This is the padding added because the ecb mode of operation requires padding if the file ends on a block boundary. The message is 560 bytes which is an exact multiple of 8, so when the ecb mode of operation divides the file into 8btyes, the last block would be exactly 8bytes(which is the block boundary), hence an extra padding block would be added.

- **File3cbc (608 bytes)**
  - **Pattern:** There is no observable pattern, this is because the cbc mode operation XORs each block of plaintext, with the previous cipher text before encrypting.(each previous cipher text will be different which will lead to different cipher texts overall)
  - **Size:** the encrypted file is 8bytes more. This is the padding added because the cbc mode of operation requires padding if the file ends on a block boundary. The message is 560 bytes which is an exact multiple of 8, so when the ecb mode of operation divides the file into 8btyes, the last block would be exactly 8bytes (which is the block boundary), hence an extra padding block would be added.

- **File3cfb(600 bytes)**
  - **Pattern:** There is no observable pattern, cfb uses the entire output of the block cipher(the plaintext XOR the output of the block cipher) to encrypt the next block.(each previous cipher text will be different which will lead to different cipher texts overall)
  - **Size:** the file size is the same because no padding is required

- **File3ofb(600 bytes)**
  - **Pattern:** There is no observable pattern, because ofb generates key stream blocks which are XORed with the plain text blocks to get the cipher text.
  - **Size:** the file size is the same because no padding is required

**Note:** For all ecb files the additional last 8 are always the same, this is because of the way the ecb mode of operation adds padding to end of the file.

**Decrypting the error files**

**File1ecberror**

- The block in which the single byte was changed became corrupt, but all other blocks was decrypted correctly.

**File1cbcerror**

- The block in which the single byte was changed became corrupt but all other blocks were decrypted correctly.

**File1cfberror**

- The error byte was corrupt and the block immediately after the block with the error byte was corrupt, but all other blocks were decrypted correctly

**File1ofberror**

- Only the byte which was changed was corrupt, all other bytes were successfully decrypted

**File2ecberror**

- The block in which the single byte was changed became corrupt, but all other blocks was decrypted correctly.

**File2cbcerror**

- The block in which the single byte was changed became corrupt and the next byte was corrupt, but all other blocks were decrypted correctly.

**File2cfberror**

- The error byte was corrupt and the block immediately after the block with the error byte was corrupt, but all other blocks were decrypted correctly

**File2ofberror**

- Only the byte which was changed was corrupt, all other bytes were successfully decrypted

**File3ecberror**

- The block in which the single byte was changed became corrupt, but all other blocks was decrypted correctly.

**File3cbcerror**

- The block in which the single byte was changed became corrupt and the next byte was corrupt, but all other blocks were decrypted correctly.

**File3cfberror**

- The error byte was corrupt and the block immediately after the block with the error byte was corrupt, but all other blocks were decrypted correctly

**File3ofberror**

- Only the byte which was changed was corrupt, all other bytes were successfully decrypted

**Summary NOTE:**

**FOR THE ofb error files:**

- only the byte which was changed was affected, because the ofb mode of operation is a stream cipher. When decrypting that byte is XORed with the errored cipher text, hence the only that byte will be decrypted wrongly.

**FOR THE cbc error files:**

- The block of the errored byte was corrupt and the next byte, but all other blocks were decrypted correctly. This is because the decryption of the errored byte block would lead to corrupt block, then that block is XORed with the decryption of the next cipher text byte which would make it corrupt.

**FOR THE cfb error files:**

- The single error byte was corrupt and the block immediately after the block with the error byte was corrupt, but all other blocks were decrypted correctly. This is because that byte would be decrypted incorrectly and the ciphertext byte that was errored is used in the decryption of the entire next block. But the remaining next cipher texts are correct, so they are not affected.

**FOR THE ecb error files:**

- Only the block with the corrupt byte was decrypted incorrectly because only that block is used in decryption.

**Decrypting the salted error files**

- The salted error files have the same decryption outcomes but the positions of the corrupt blocks/bytes are shifted because of the added bytes of the salt to the file.

**SpreadSheet Questions**

**a)** 7 different salaries

**b)** Line 2, 6 , 12 have the same salaries, line 3, 7, 11 have the same salaries and line 5 & 8 have the same salaries