Group uc05

Avery Tan Kirsten Kwong Oleksii Shevchenko

PART 1

Plaintext

The plaintext turned out to be a poem by Pablo Neruda - "Tonight I Can Write The Saddest Lines". Full version of plaintext can be found at plaintext1.txt

Key

Key length is 7 characters long, here are its corresponding hex values:

```
key = ["50", "2f", "08", "7c", "5f", "30", "00"]
```

Automation

We automated finding the key by assuming that the highest occuring byte will be the space ascii character.

We ensured that the key was correct by trying to decrypt gven file with the right key. The process of finding the key is automated, but its verification is not.

Source code

Instructions for using part 1 code:

- navigate to directory containing Part1 containing ctext1hexdump.txt, part1.py
- in the terminal enter `python3 part1.py`
- a plaintext file named 'plaintext1.txt' will be generated containing the plaintext of ciphertext1.

Approach

Explanation: we determined the gaps between high frequency of bytes used in the ciphertext and found out that there was a significantly higher spike in word frequency every 7th byte. We thus deduce that the key length is 7.

We then processed every 7th character since the key length is 7 and that would mean every 7th character would be encrypted using the same key value. Once we have this frequency analysis of every 7th byte, we assign the highest occuring byte to be equal to the plain ascii value of a space, '0x20'. We do this to find our key.

We then use this key to decrypt the ciphertext to produce 'plaintext1.txt'

Plaintext compression

In the case of file being compressed before encrypting, we would analyze byte frequency of compressed text files, and find the key using the same algorythm based on common most frequent byte. We would then ensure that our encryption is correct by checking file header and comparing it to a proper zip/gzip file header standard.

Part 2

Plaintext

The plaintext turned out to be an jpg picture of a sidewalk and a bottom of a street sign. Full version of plaintext can be found at plaintext2

Key

Key length is 23 characters long, here are its corresponding hex values:

```
key = ['35', '33', '2e', '35', '30', '33', '35', '36', '33', '4e', '2c', '2d', '31',
'31', '33', '2e', '35', '32', '38', '38', '39', '34', '57']
```

Automation

We ensured that the key was correct by trying to decrypt gven file with the right key. The process of finding the key is automated, but its verification is not.

Source code

Instructions for using part 2 code:

- navigate to directory Part2 containing cipher2short, ctext2hexdump.txt, part2.py
- in the terminal enter `python3 part2.py`
- a binary file named 'plaintext2' will be generated containing the plaintext of ciphertext2.

Approach

Similarly to Part 1 approach, we found the key length to be 23.

We then processed every 23rd character since the key length is 23 and that would mean every 23rd character would be encrypted using the same key value.

However, in addition to Part 1 approach, we analyzed byte frequency of various differrent file types and determined that '0x00' is the most frequent byte for most non-text file

Once we have this frequency analysis of every 23rd byte, we assign the highest occuring byte to be equal to '0x00'. We do this to find our key.

We then use this key to decrypt the ciphertext and produce resulting file 'plaintext2'.

The benefits of partially known plaintext allowed us to verify the right file header, as well as most frequently used byte.

Part 3

> Password used in all encryptions/decryptions is 'purple'.
>
> Part3 folder contains all generated files for part 3 with descriptive names.
>
> > Modified ciphertext 3 is named ciphertext3_modified

ECB

Electronic CodeBook ever block will concatenate the next block encryption is deterministic since same input results in same output you can see patterns in the encryption very easily very easy to move block 2 to block 1 is relatively error prone. If something happesnt o data in block 1, only that block gets cuorrupted. Can decrypt multiple blocks in parallel.

We thus see in our pre-manipulated ciphertext that under the ecb option, a distinguishable pattern is easily spotted. There are 8 bytes at the very end that do not follow the pattern since they are padding. This also results in our plaintext file being smaller in size.

When we make a substitution in the 19th bit of our ecb encrypted ciphertext and decrypt it using our apssword, only the block containing our corrupted substituted byte is not decrypted. The size of the ciphertext is also grater than the plaintext file since the encryption process adds padding. This is because every block will concatenate the next block which also allows encryption to run in parallel.

CBC

instead of processing each block separately, every block will be XOR'ed with the encrypted previous block. Thus each block depends on the output of the previous block. You cannot deduce the plaintext by looking at the encrypted blocks separately. Thus parallel encryption is not possible. Failure of one block will cause all subsequent blocks to fail.

We thus see that there is no pattern observable from viewing the pre-manipulated ciphertext. There is also the presence of padding at the end of the ciphertext. This again makes our plaintext smaller in size when compared to our ciphertext.

When we see that our manipulated byte affects the decryption of the next block. The size is also greater than the plaintext file since the encryption process adds padding. The CBC encrypted ciphertext is 8 bytes longer than it's plaintext counterpart.

CFB

is a mode of operation for a block cipher. In CFB the previous ciphertext block is encrypted and the output is XOR'ed. This operation conceals plaintext patterns. This entropy can be implemented as a stream cipher.

The pre-manipulated ciphertext has the first 8 bytes that are the same with the OFB encrypted ciphertext. The encryption pattern also appears random. There is no padding in this algorithm

Thus we see that our corrupted byte is influencing the decryption of the next block.

OFB

Makes a block cipher into a synchrnous stream cipher. Generates keystream blocks, which are then XOR'ed with plaintext blocks to get the ciphertext.

The pre-manipulated ciphertext has the first 8 bytes that are the same with the OFB encrypted ciphertext. The encryption pattern also appears random. There is no padding in this algorithm

When we make our substitution and decrypt the file, there is only one byte corrupted since OFB makes a block cipher into a synchronous stream cipher.