Lab 1

Ally Smith (Section A)

March 16, 2022

1. Encryption using different ciphers and modes

I tried the AES-128-CBC, DES-CBC, and DES3 encryption modes, each with a few key values. For each of the encryptions I made, even a small difference in the key made a significant change in the cipher text, demonstrating the avalanche effect.

2. Encryption Mode — ECB vs. CBC

I encrypted the picture using they key and initial value from task 1. After fixing the header to the bitmap file, I was able to see no patterns in the encrypted image, seen below. The image that was generated looks to be random 'noise,' so no useful information can be derived from the picture.

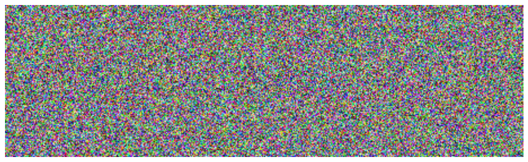

Figure 1: Ciphered image

3. Encryption Mode — Corrupted Cipher Text

I suspect that using ECB on the corrupted cipher text will have the most recoverable text, as there is no feedback, so the bit error will not propagate beyond the block it is in. In CBC, the previous block's cipher text is XORed with the decrypted block, so the bit error will affect the decrypted text in the block *after* that which the error occurred in the cipher text. In CFB, I suspect that the error will greatly affect the deciphered text, as the ciphertext block is used as feedback to the next block. Because feedback in OFB is independent of the plaintext or ciphertext, it is possible that a small (1 bit) change in the ciphertext will not affect the decryption process at all.

After performing the encryptions, changing a single bit in the 30th byte, and decrypting, I was surprised by some of the results. For all of the methods used, over 90% of the plaintext was recoverable. Below is a table showing how many had changed in the decrypted plaintext from the plaintext (out of a total of 446 characters).

ECB	16
CBC	15
CFB	18
OFB	0

4. Padding

4.1) For the first exercise, I used AES-128-CBC to encrypt texts of length 20, 31, and 32 bytes. The sizes of their respective cipher texts were 32, 32, and 48 bytes. This implies that the block size is 16, and that a plaintext must be strictly less than a multiple of the block size to have a corresponding ciphertext of that size.

4.2) I chose to use the AES cipher for this exercise. The ECB and CBC modes of encryption required padding. I verified this by comparing the size in bytes of the plaintext and the ciphertext from each mode. CFB and OFB don't require padding because the encryption method is used only to generate pseudo-random bits to be XORed with the message. Therefore, it doesn't require that the message be an integer-multiple of a block size.

5. Programming using the Crypto Library

I was able to implement this using some file reading, looping, and using the EVP interface. In the end, the word that generated the same ciphertext with the given parameters was `median`. Once my program found this, I was able to verify the result using the `openssl` command line program.

Below is the code that I used to determine the word that was used for the key during the encryption:

Listing 1: Source code

```python
from cryptography.hazmat.backends import default_backend
from cryptography.hazmat.primitives.ciphers import Cipher
from cryptography.hazmat.primitives.ciphers.algorithms import AES
from cryptography.hazmat.primitives.ciphers.modes import CBC
from cryptography.hazmat.primitives.padding import PKCS7

iv = b'0000000000000000'

solution = "8d20e5056a8d24d0462ce74e4904c1b513e10d1df4a2ef2ad4540fae1ca0aaf9"
plain = 'This is a top secret.'

if __name__ == '__main__':

    words_file = open('words.txt', 'r')
    # words = words_file.readlines()
    words = ['median']

    backend = default_backend()
    padder = PKCS7(128).padder()

    for word in words:
        print(f'KEY: {word}')
        keystr = word
        if (len(keystr) >= 16):
            continue # skip words that are too long

        # pad key with spaces
        while (len(keystr) < 16):
            keystr += ' '

        key = bytes(keystr, 'ascii')
        print(f'BYTES:')
        print(key.hex(' '))

        aes = AES(key)
        cbc = CBC(iv)

        cipher = Cipher(aes, cbc, backend)
        enc = cipher.encryptor()
        cipher_text = enc.update(bytes(plain, 'ascii')) + enc.finalize()
        print(cipher_text)

        print('CIPHER TEXT:')
        print(cipher_text.hex(' '))

        if (cipher_text.hex('') == solution):
            print(f"FOUND SOLUTION! {word}")
            break
```

6. Pseudo Random Number Generation

(a) Measure the Entropy of Kernel

When measuring the entropy on Isengard, I got a value right around 3500. After clicking and typing the value would increase. The more I typed, the more the value would increase.

(b) Get Pseudo Random Numbers from `/dev/random`

By typing and moving the mouse around more, you can increase the entropy of a system enough to unblock `/dev/random`.

(c) Get Random Numbers from `/dev/urandom`

I wasn't able to get `/dev/urandom` to block when running the command. In fact, the entropy of the system increased some from typing it in.