Nick Charchenko CSCE 465 HW 3: Secret Key Encryption Due 3.5.2018

Textbook Problems:

2.2: Anyone who knows which hash function is being used can forge a message, which would go undetected.

2.3: Not really. This arrangement would require any of the three people to know the other's secret keys for verifying answers to the given challenge. Because Alice would have to know Carol's secret key to verify Carol's answer, she can impersonate Carol to Bob. This extends to any combination of the three people.

2.4 : You can forge a signature on any message with the same hash as one that has been legitimately signed.

3.2: All you need is a clock and some cryptographic scheme containing the key. At every tick of the clock, you generate a new number to display. The number could just be the secret key encryption of the time, or it could be the next output of a cryptographic pseudo-random number generator (such as RC4). The device's clock would have to not drift (or at least-drift in a predictable fashion).

3.3: There are 2^56 possible keys and 2^64 possible ciphertext blocks for a particular plaintext block. So only ab1/256 of the possible ciphertext blocks can be obtained with a DES key.

3.5: DES consists of an initial permutation, 16 DES rounds (Feistel cipher), swapping the left and right halves, and then inversing the initial permutation. If our mangler function always outputs 0, then a DES round simply swaps the left and right halves. The initial 64-bit plaintext would not change. It would simply be an initial permutation, swapping the left and right halves, and then a final permutation.

4.2: K{IV} will be the first block to repeat. To prove this, let b_i denote the i-fold encryption of IV. So the pad sequence is b_1, b_2, b_1, ..., where b_{i+1} is the encryption of b_i and b_i is the decryption of b_{i+1} (because decryption is the inverse of encryption). Let b_k be the first repeat element and let $b_k=b_j$ where $j < k$. If $j=1$ then we're finished. If $j > 1$ then $b_{j-1} = b_{k-1}$ (since bj = b_k). So b_k is not the first repeat element. This is a contradiction. So $b_k = b_1$.

4.4: Let p be the 64-bit block of plaintext and c be the 64-bit block of ciphertext. Make a table with n entries by the following loop: choose a key at random; encrypt p with that key. If the result is already in the table, continue. Otherwise, store the result together with the key, continuing if the table is not yet full. To find a triple of keys, pick a pair of keys at random, decrypt c with the second key, encrypt the result with the first key, and see if the result is in the table. If it is. you 've found a triple (the key from the table entry together with the pair of keys just chosen). Otherwise, try again.

4.6: Yes, it would work. However, if all the plaintext blocks are the same, then all the ciphertext bocks will be the same, excluding the first block. This is because the first block is XOR'd with the initialization vector instead of the previous block's ciphertext.

Lab Task 1: Encryption using different ciphers and modes

For this task, we were tasked with exploring the different encryption algorithms (AES, DES, Triple DES) and the different modes of block cipher encryption (CBC, CFB, OFB) For the task, I did AES 128-bit encryption, DES encryption, and Triple DES with EDE using two keys. I chose to do the modes

CBC, CFB, and OFB, as we were only required to do at least three. The plaintext I chose to encrypt is a file with the sentence, “Hi, I’m a very important secret!” I have included all of the ciphertext files with the submission, but here is an example of what ciphertext looks like in GHex.

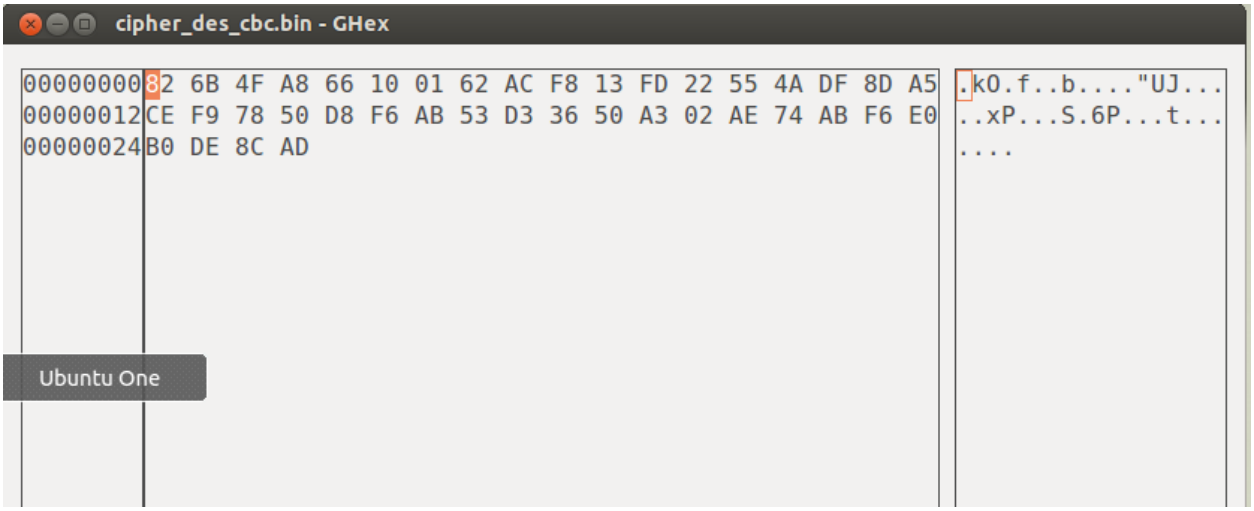

Figure 1: Ciphertext from DES CBC shown in GHex

Task 2: Encryption Mode- ECB vs CBC

For this task, we are given a picture and we encrypt it using AES-128 in both ECB mode and CBC mode.

Figure 2 Unencrypted picture

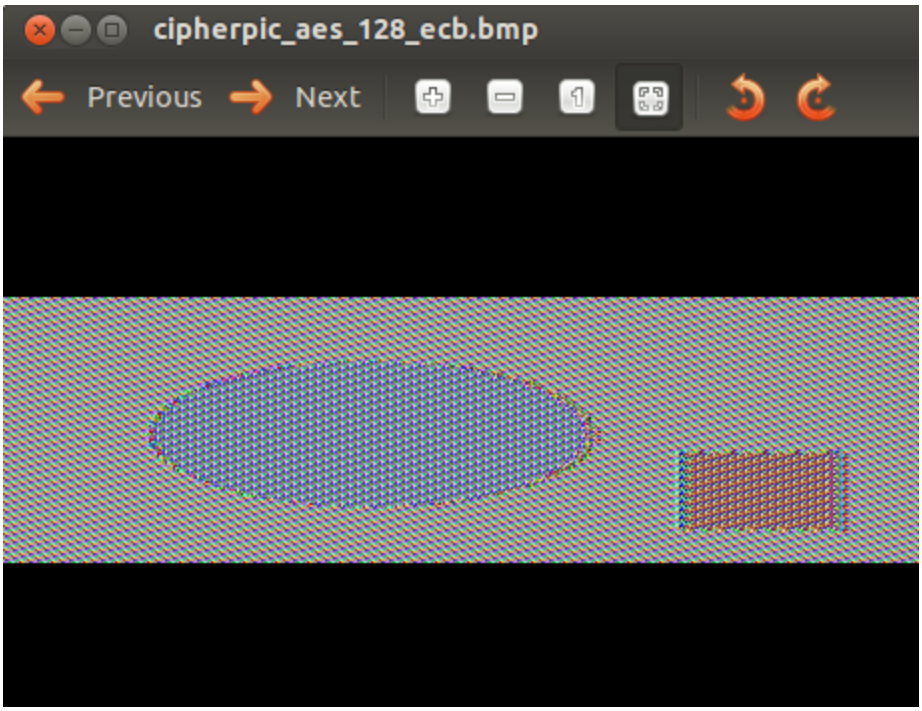

Figure 3 Encrypted with ECB

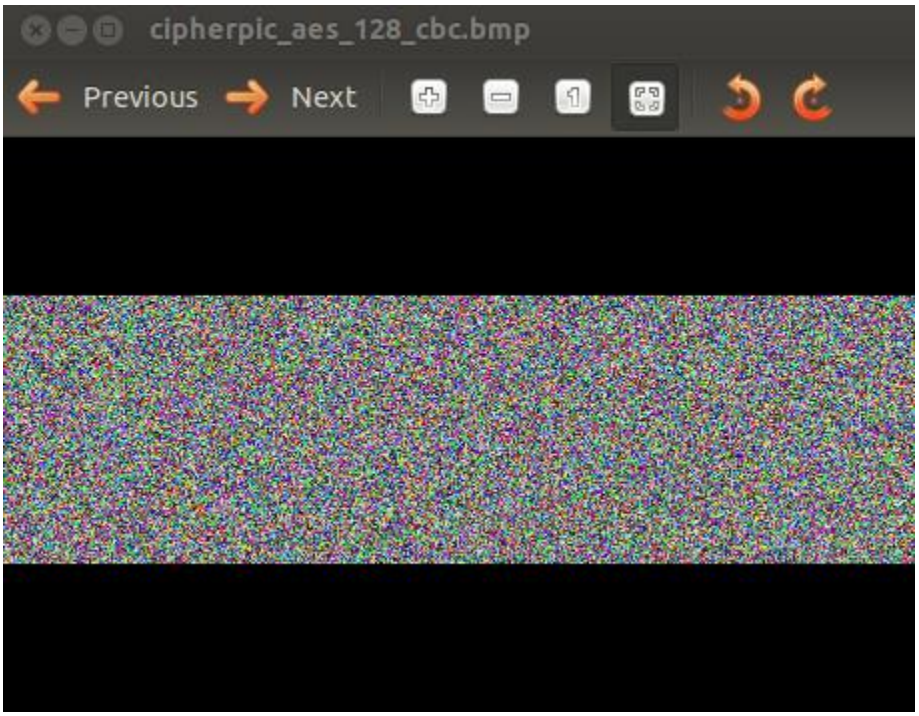

Figure 4 Encrypted with CBC

We can observe from looking at the ECB encrypted photo the general shape of the image (there is an oval and a rectangle) and that they are different colors (the encrypted oval is blue and the encrypted rectangle is red with some green and purple). This is a pretty significant information leak as an attacker who intercepts this “encrypted” message can make out details which could be

compromising. This happens because the ECB encryption mode simply encrypts every block of plaintext **independently** such that repeated blocks of plaintext will produce identical blocks of ciphertext. On the other hand, the CBC encrypted image is completely unrecognizable. This is because the CBC mode XORs the previous block's ciphertext (or an initialization vector for the first block) with the plaintext before encrypting. Repeated plaintext blocks won't have the same ciphertext.

Task 3: Encryption Mode-Corrupted Cipher Text

```
Hi there, I'm top secret!
Very, very secret.
Please make sure unauthorized users cannot read me.
```

Figure 5 Task 3 plaintext

For Task 3, I created a plaintext of size 97 bytes. I then encrypted it using AES-128 and the ECB, CBC, CFB, and OFB modes. For each ciphertext output, I changed the first bit of the 30th byte to corrupt the encrypted file.

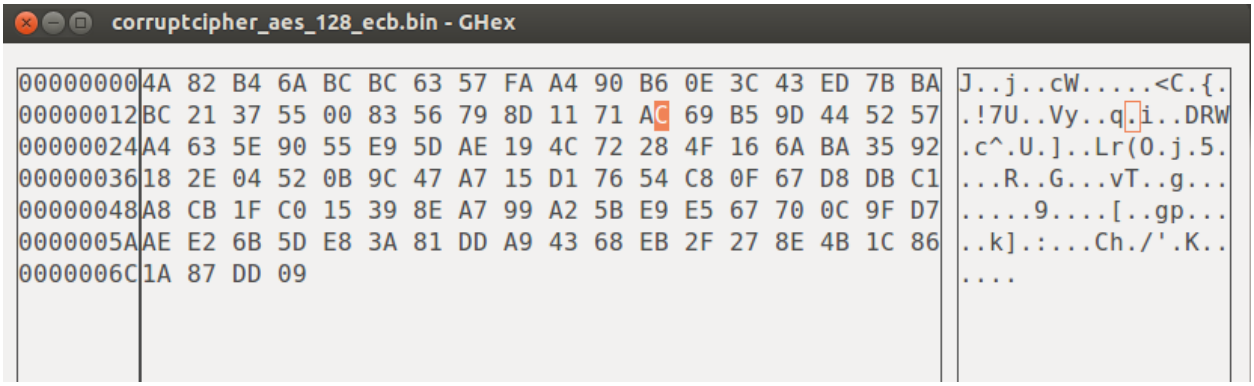

Figure 6 Corrupting the encrypted file. The original ciphertext had 9C for the 30th byte.

ECB: I expect the error to not propagate beyond the corrupted bits themselves because blocks are encrypted/decrypted independently.

CBC: I expect the error to propagate to the next block. This is because only the next block depends on the corrupted ciphertext.

CFB: I expect the worst error propagation from this mode because the corrupted ciphertext is immediately fed back into the encryption algorithm before being XOR'd with plaintext.

OFB: I do not expect error propagation from this mode because **ciphertext is never used** with the encryption process.

After performing the task of encryption, corruption, and decryption, I got these plaintext files.

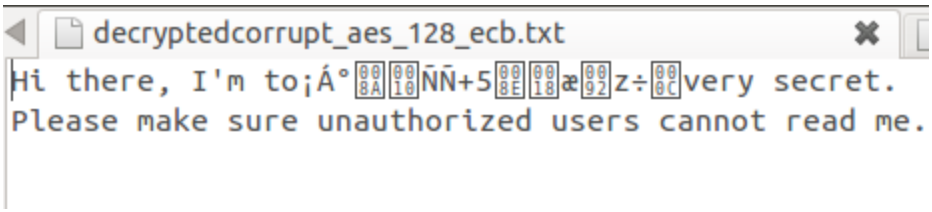

Figure 7 Decrypting corrupted ECB ciphertext

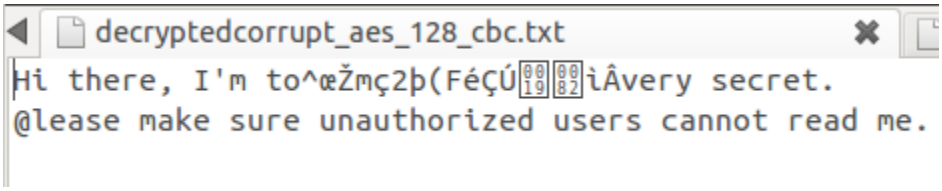

Figure 8 Decrypting corrupted CBC ciphertext

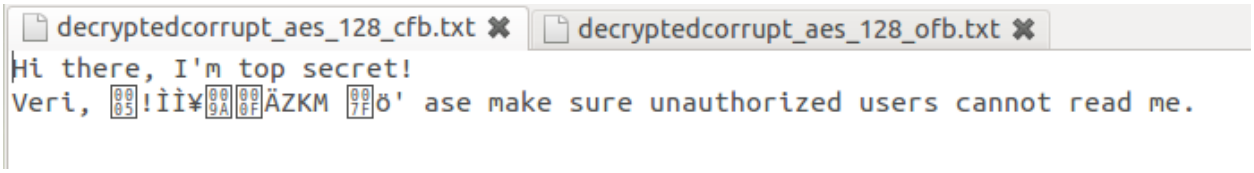

Figure 9 Decrypting corrupted CFB ciphertext

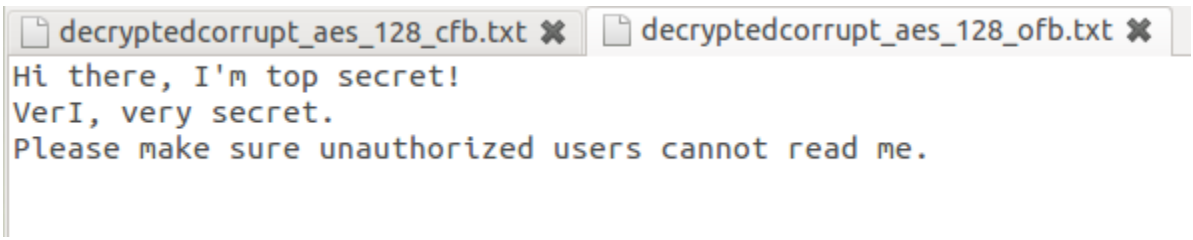

Figure 10 Decrypting corrupted OFB ciphertext

ECB, as expected, only showed corruption in the immediate block. Second, CBC's corruption affected the immediate block, but also propagated to a block further down, unlike ECB, which was expected. Next, CFB, as expected, suffered the worst out of the four modes, but not much worse than ECB. Finally, OFB showed the greatest resilience to corrupted ciphertext, as only one character of plaintext, after decryption, came out different than that of the original plaintext. This shows that **OFB is the most resilient to corrupted or tampered ciphertext.** This is an important defense against a tampering attack on integrity. If a hacker can't steal secrets, then he may as well attempt to ruin the data for the authorized parties by corrupting the ciphertext, which makes error propagation important to consider when selecting a mode of encryption.

Task 4: Programming using the Crypto Library

The goal is to determine the encryption key used to encrypt a known plaintext using this information:

1. Cipher type: AES-128-CBC
2. Initialization Vector: All zeroes
3. Length of the key is less than sixteen characters and is an English word
4. Ciphertext output.

This all sets up nicely for a dictionary attack. We can get every possible key, which is given to us in the words.txt file, and run AES-128-CBC encryption using the given plaintext, “This is a top secret.” and the given initialization vector of all zeroes, after which we compare the ciphertext output with the given ciphertext and see if we get a match. The OpenSSL library proves to be very useful with performing the encryption and verifying our key and IV lengths.

- EVP_CIPHER_CTX_init initializes our cipher context by clearing all information from the cipher context and freeing up allocated memory associated with the ctx (except the ctx itself). This ensures we get consistent encryption.
- OPENSSL_assert is used to validate our key and IV by ensuring their lengths are equal to 16.
- EVP_CipherInit_ex, EVP_CipherUpdate, and EVP_CipherFinal_ex all perform the encryption process.
    - EVP_CipherInit_ex takes arguments that specify the cipher context, mode of encryption/decryption (AES 128-bit with CTC mode), engine implementation (NULL for our purposes, key (from words.txt), initialization vector (sixteen zeroes), and an integer to specify encryption or decryption (1 for encrypt).
    - EVP_CipherUpdate is “where the magic happens.” It encrypts/decrypts plaintext/ciphertext from an “in” buffer and writes the ciphertext/plaintext to the “out” buffer. For our purposes, this function performs encryption.
    - EVIP_Cipher_Final_ex encrypts the last (possibly partial) block.

We use a makefile to include the OpenSSL library so that our program can compile and execute.

From running our findkey program, we can deduce that the word “median” was used to encrypt our plaintext.

```
[03/05/2018 15:06] seed@ubuntu:~/Desktop/csce465hw3dump$ make all
gcc -std=c99 -I/USR/local/ssl/include/ -L/usr/local/ssl/lib/ -o findkey findkey.c -lcryp
to -ldl
[03/05/2018 15:06] seed@ubuntu:~/Desktop/csce465hw3dump$ ./findkey
I found a key and it is median
```

Figure 11 Findkey program in action.