CSCI 476: Lab 09

Nathan Stouffer

April 12, 2020

Task 1

In Task 1, we are given a cipher text encrypted using a monoalphabetic cipher. Since we know that the plaintext was written in english and monoalphabetic ciphers are bijective maps between the alphabet and a permutation of the alphabet, we can use frequency analysis on the cipher text to deduce the cipher used to encrypt the data.

Using frequency analysis, we determine that the most common trigrams in the cipher text are "ytn" and "vup." Based on the fact that the most common trigrams in the English language are "the" and "and," we can guess the following maps from plain to cipher text:

$$t \mapsto y \quad h \mapsto t \quad e \mapsto n \quad a \mapsto v \quad n \mapsto u \quad d \mapsto p$$

After applying those maps to the cipher text, we can make another few guesses based on words that are mostly filled with known letters. We now guess the following:

$$o \mapsto x \quad l \mapsto i \quad b \mapsto g \quad u \mapsto z \quad w \mapsto l \quad c \mapsto a$$

Repeating this process, we can deduce the remaining unknown letters:

$$f \mapsto b \quad i \mapsto m \quad s \mapsto q \quad g \mapsto r \quad r \mapsto h \quad m \mapsto c \quad p \mapsto e$$
$$x \mapsto k \quad y \mapsto d \quad z \mapsto w \quad v \mapsto f \quad q \mapsto j \quad k \mapsto s \quad j \mapsto o$$

Thus, in abbreviated form, we could express the entire mapping as

$$\text{“}abcdefghijklmnopqrstuvwxyz\text{”} \mapsto \text{“}vgapnbrtmosicuxejhqyzflkdw\text{”}$$

We can the use inverse mapping (since the map is bijective) to decrypt the cipher text. A portion of the decrypted text is displayed in Figure 1.

Figure 1: Decrypted text

```
10.0.2.4 seed ~/Documents/comp-security/lab09/code/task1
$ cat plain.txt
THE OSCARS TURN  ON SUNDAY WHICH SEEMS ABOUT RIGHT AFTER THIS LONG STRANGE
AWARDS TRIP THE BAGGER FEELS LIKE A NONAGENARIAN TOO

THE AWARDS RACE WAS BOOKENDED BY THE DEMISE OF HARVEY WEINSTEIN AT ITS OUTSET
AND THE APPARENT IMPLOSION OF HIS FILM COMPANY AT THE END AND IT WAS SHAPED BY
THE EMERGENCE OF METOO TIMES UP BLACKGOWN POLITICS ARMCANDY ACTIVISM AND
A NATIONAL CONVERSATION AS BRIEF AND MAD AS A FEVER DREAM ABOUT WHETHER THERE
OUGHT TO BE A PRESIDENT WINFREY THE SEASON DIDNT JUST SEEM EXTRA LONG IT WAS
EXTRA LONG BECAUSE THE OSCARS WERE MOVED TO THE FIRST WEEKEND IN MARCH TO
AVOID CONFLICTING WITH THE CLOSING CEREMONY OF THE WINTER OLYMPICS THANKS
PYEONGCHANG

ONE BIG QUESTION SURROUNDING THIS YEARS ACADEMY AWARDS IS HOW OR IF THE
CEREMONY WILL ADDRESS METOO ESPECIALLY AFTER THE GOLDEN GLOBES WHICH BECAME
A JUBILANT COMINGOUT PARTY FOR TIMES UP THE MOVEMENT SPEARHEADED BY
POWERFUL HOLLYWOOD WOMEN WHO HELPED RAISE MILLIONS OF DOLLARS TO FIGHT SEXUAL
HARASSMENT AROUND THE COUNTRY

SIGNALING THEIR SUPPORT GOLDEN GLOBES ATTENDEES SWATHED THEMSELVES IN BLACK
SPORTED LAPEL PINS AND SOUNDED OFF ABOUT SEXIST POWER IMBALANCES FROM THE RED
CARPET AND THE STAGE ON THE AIR E WAS CALLED OUT ABOUT PAY INEQUITY AFTER
ITS FORMER ANCHOR CATT SADLER QUIT ONCE SHE LEARNED THAT SHE WAS MAKING FAR
LESS THAN A MALE COHOST AND DURING THE CEREMONY NATALIE PORTMAN TOOK A BLUNT
AND SATISFYING DIG AT THE ALLMALE ROSTER OF NOMINATED DIRECTORS HOW COULD
THAT BE TOPPED

AS IT TURNS OUT AT LEAST IN TERMS OF THE OSCARS IT PROBABLY WONT BE
```

Note

A quick note for Tasks 2-5. We now turn to investigating the Advanced Encryption Standard (AES) and the modes within. We will use the following key and initialization vector when necessary.

-K 00112233445566778899AABBCCDDEEFF

-iv 000102030405060708090A0B0C0D0E0F

Task 2

In Task 2, we are asked to explore three different encryption modes within AES and note their differences. The three modes used were ECB, CBC, and CFB. We begin with a 65 byte file "plain.txt" and use each mode to create a seperate, enrypted file. We can now note two primary differences. First, there is the size of the encrypted files (shown in Figure 2). Note that the file produced by the CFB mode has fewer bits than the file produced by modes CBC and ECB. This is because CFB is a stream cipher and does not require padding.

Figure 2: Encrypted file sizes

```
10.0.2.4 seed ~/Documents/comp-security/lab09/code/task2
$ ls -l
total 16
-rw-rw-r-- 1 seed seed 80 Apr  6 14:49 cbc-cipher.bin
-rw-rw-r-- 1 seed seed 65 Apr  6 14:50 cfb-cipher.bin
-rw-rw-r-- 1 seed seed 80 Apr  6 14:47 ecb-cipher.bin
-rw-rw-r-- 1 seed seed 65 Apr  6 14:44 plain.txt
```

We can also notice a weak point in the ECB mode. Figure 3 displays the bytes of each encrypted file. We can see that there is no discernible patterns in the files encrypted with the CBC and CFB modes. However, in the ECB file, the exact same output is repeated 4 times. This is because "plain.txt" contains 4 copies of a 16 byte message and the ECB mode does not take appropriate precautions.

Figure 3: Encrypted file contents

```
10.0.2.4 seed ~/Documents/comp-security/lab09/code/task2
$ xxd ecb-cipher.bin
00000000: 2b74 649a 7905 96c0 3b99 b6fb de91 a0ee  +td.y...;.......
00000010: 2b74 649a 7905 96c0 3b99 b6fb de91 a0ee  +td.y...;.......
00000020: 2b74 649a 7905 96c0 3b99 b6fb de91 a0ee  +td.y...;.......
00000030: 2b74 649a 7905 96c0 3b99 b6fb de91 a0ee  +td.y...;.......
00000040: dc8d 1001 f582 e258 f7d2 9180 0ee8 5150  .......X......QP

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task2
$ xxd cbc-cipher.bin
00000000: 9623 0317 da31 a53f 0de7 603d 7dab c99c  .#...1.?..`=}...
00000010: 543d dc2b 1504 9d47 0c62 d9a5 da77 fe1c  T=.+...G.b...w..
00000020: 9660 feb9 f9ca 292e e13b 4f31 2e9e 6dfc  .`....)..;O1..m.
00000030: b6ff 02ec ef4e 94fa 92ef f4e9 4d78 dbef  .....N......Mx..
00000040: 693b 1958 3dbd 4f36 96e0 0b88 22b4 eae7  i;.X=.O6....".."

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task2
$ xxd cfb-cipher.bin
00000000: 54ea c72f 0752 603b ece9 eb82 f183 9364  T../.R`;.......d
00000010: 0ecc 7dc6 a855 e0bf ea58 a38e fc5c 3229  ..}..U...X...\2)
00000020: 51cc 1010 f930 5dff e23e 7055 19e6 cf86  Q....0]..>pU....
00000030: 9491 6ec9 aa4c 3ca0 42de c94a b608 5686  ..n..L<.B..J..V.
00000040: f5                                       .
```

Task 3

In Task 3, we investigate the differences between the cipher texts of the modes ECB and CBC. In each subtask, we will begin with an image, and encrypt it twice (once in ECB mode and the other in CBC mode).

We would then like to view the contents of the encrypted file as an image. However, we can't since we have also encrypted the bmp header. However, this can be overcome since the header is a standard number of bytes (54 to be exact). To view the encrypted image, we can overwrite the encrypted header with the original header and use a standard image viewer (such as eog). Once this is done, we can view the encrypted image and make conclusions.

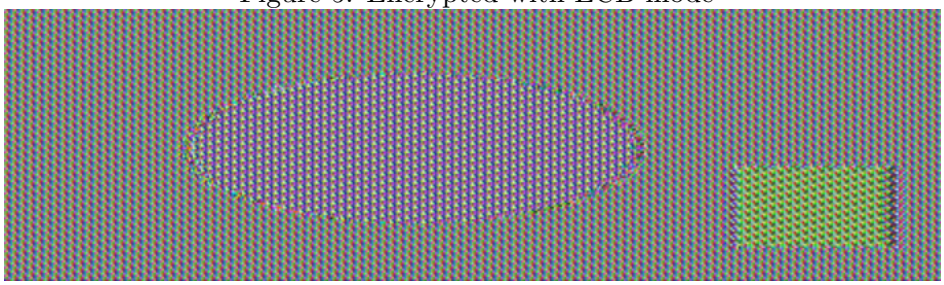
Task 3.1

In Task 3.1, we perform the above process for the image given in Figure 4. Since there are few color changes, many of the bytes that represent this file will show up in repeated patterns.

Figure 4: Unencrypted image

This effect can be clearly seen in Figure 5. Even though the image is encrypted, we can still deduce that the original image contained an ellipse and a rectangle. This is because Figure 5 was encrypted using the ECB mode.

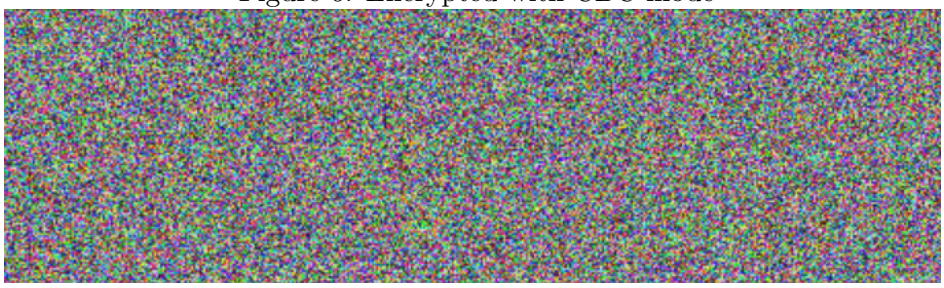
Figure 5: Encrypted with ECB mode



On the other hand, Figure 6 was encrypted using the CBC mode. This is a stronger mode because the encrypted image is entirely gibberish despite the fact that the input had a consistent pattern.

Experiments with this image show the weakness of the ECB mode and the importance of initialization vectors (since the use of initialization vectors strengthen the encrypted data by removing patterns in the encrypted data).

Figure 6: Encrypted with CBC mode



Task 3.2

We now repeat the experiment from Task 3.1 using the image displayed in Figure 7. Before showing the experiment, we should note certain features about the image in Figure 7. First, there is a lot of space in the image with identical bits. This means that we should expect to find a pattern in the image encrypted with ECB mode. However, the triangle contains some information that should be lost: the color. When viewing the image produced with the ECB mode, we should only see that there *was* a triangle, not it's color.

Figure 7: Unencrypted image

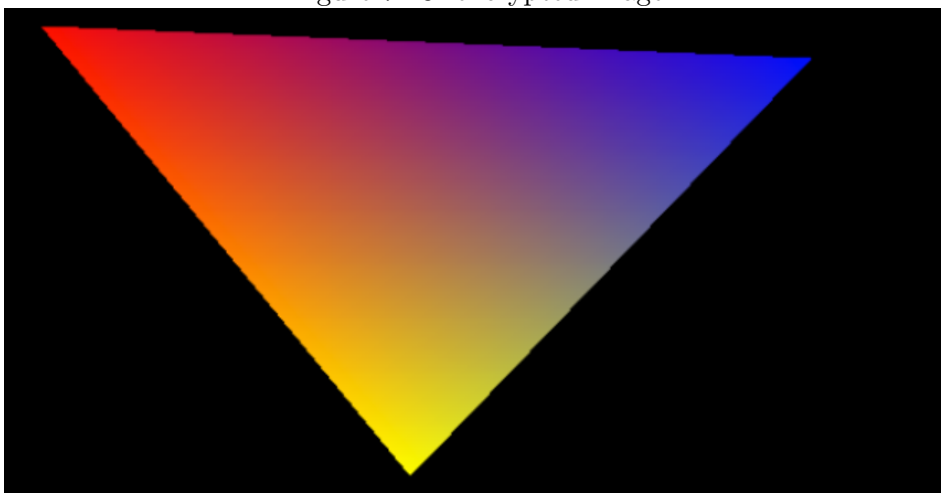


Figure 8 shows the image generated using the ECB mode. Note that the output is similar to what we expected. Inside the triangle, there is no pattern and we cannot even tell that the original triangle was colored. However, the pattern outside the triangle reveals that the original image contained a triangle.

Figure 8: Encrypted with ECB mode

Figure 9 shows the output generated using the CBC mode. There is no discernible pattern in the cipher image.

Figure 9: Encrypted with CBC mode

Task 4

In Task 4, we focus on padding. In Task 4.1, we investigate the padding in the CBC mode and then we explore padding in modes ECB, CFB, and OFB in Task 4.2. Within each mode, we consider the following three files and their contents:

plain5.txt: 01234
plain10.txt: 0123456789
plain16.txt: 0123457890abcdef

We will then encrypt encrypt each file to produce a cipher text. From there, we decrypt the cipher text using the "-nopad" option. That way, we can view and analyze how padding works in each mode.

Task 4.1

Cipher Block Chaining

We begin by looking at the CBC mode. The encrypted file sizes are displayed as follows:

plain5.txt: 16 bytes
plain10.txt: 16 bytes
plain16.txt: 32 bytes

Note that the file sizes are multiples of 16 bytes, this is because the CBC mode requires that the data comes in blocks of 128 bits (equivalent to 16 bytes). To perform the encryption process, padding must be added to the input. We can then decrypt (while keeping the added padding) and view how the padding works. The decrypted file is shown in Figure 10.

Figure 10: Binary files for CBC

```
10.0.2.4 seed ~/Documents/comp-security/lab09/code/task4/cbc
$ hexdump -C dec5.txt
00000000  30 31 32 33 34 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |01234...........|
00000010

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task4/cbc
$ hexdump -C dec10.txt
00000000  30 31 32 33 34 35 36 37  38 39 06 06 06 06 06 06  |0123456789......|
00000010

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task4/cbc
$ hexdump -C dec16.txt
00000000  30 31 32 33 34 35 36 37  38 39 61 62 63 64 65 66  |0123456789abcdef|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020
```

Keep in mind that "0b" in hex is 12 in decimal and "10" in hex is 16 decimal. From this information, we can describe the pattern for padding. Say there are n bytes of data in the last block. We would then need to pad $16 - n$ bytes (all computations are done in decimal). From Figure 10, the value that is used to pad is $16 - n$ (converted to hex). This rule works in most cases, but note **dec16.txt** where the last block is 16 bytes. In this case, an entire block of padding is added.

Task 4.2

We now move on to analyzing the padding methodology in the other encryption modes.

Electronic Code Book

As Figure 11 shows, the ECB mode uses the same padding process as CBC. That is, for a block with n bytes, $16 - n$ bytes of padding are added (excepting the case where n is 16 bytes). The encrypted file sizes are shown below.

plain5.txt: 16 bytes
plain10.txt: 16 bytes
plain16.txt: 32 bytes

Figure 11: Binary files for EBC

```
10.0.2.4 seed ~/Documents/comp-security/lab09/code/task4/ecb
$ hexdump -C dec5.txt
00000000  30 31 32 33 34 0b 0b 0b  0b 0b 0b 0b 0b 0b 0b 0b  |01234...........|
00000010

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task4/ecb
$ hexdump -C dec10.txt
00000000  30 31 32 33 34 35 36 37  38 39 06 06 06 06 06 06  |0123456789......|
00000010

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task4/ecb
$ hexdump -C dec16.txt
00000000  30 31 32 33 34 35 36 37  38 39 61 62 63 64 65 66  |0123456789abcdef|
00000010  10 10 10 10 10 10 10 10  10 10 10 10 10 10 10 10  |................|
00000020
```

Cipher Feedback

In the CFB mode, we encounter a difference in padding methodologies. We first encounter this when comparing the encrypted file sizes. They are as follows:

plain5.txt: 5 bytes
plain10.txt: 10 bytes
plain16.txt: 16 bytes

Note that the encrypted sizes match the input sizes exactly. This is because CFB mode is a stream cipher, so there is no need for padding. Figure 12 shows the decrypted files without padding.

Figure 12: Binary files for CFB

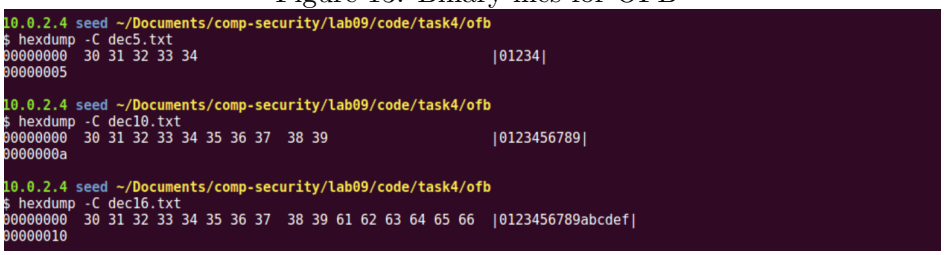
```
10.0.2.4 seed ~/Documents/comp-security/lab09/code/task4/cfb
$ hexdump -C dec5.txt
00000000  30 31 32 33 34                                    |01234|
00000005

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task4/cfb
$ hexdump -C dec10.txt
00000000  30 31 32 33 34 35 36 37  38 39                    |0123456789|
0000000a

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task4/cfb
$ hexdump -C dec16.txt
00000000  30 31 32 33 34 35 36 37  38 39 61 62 63 64 65 66  |0123456789abcdef|
00000010
```

Output Feedback

The OFB mode is similar to CFB mode in that it is also a stream cipher. So we should see the same file sizes as well as decrypted data. Both are shown below.

plain5.txt: 5 bytes
plain10.txt: 10 bytes
plain16.txt: 16 bytes

Figure 13: Binary files for OFB



Task 5

Task 5 investigates error propagation. We will encrypt a file, change a single bit in the cipher text, and decrypt to view the results.

Task 5.1

Before performing the experiment, we answer the following question for each encryption mode: *How much information can you recover by decrypting the corrupted file?*

1. **ECB:** In the ECB mode, we will lose a 16 byte block of information. Since only one bit in the ciper text will be flipped and each block is evaluated independently, only one block in the output will be affected. However, we will lose the entire block since it is now a different number. The decryption process will decrypt the cipher text just like any other block, which will result in an entirely different number than the original plain text.
2. **CBC:** The CBC mode will not only lose a 16 byte block (we call this block B_n), but also a single byte in block B_{n+1}, the block following B_n. Block B_n is lost for the same reason as in ECB. We will lose a single byte in block B_{n+1} because the cipher text of block B_n is XORed with B_{n+1} in the decryption process. This is will corrupt a single bit in the output of B_{n+1}.
3. **CFB:** The CFB mode's corrupted output will be quite similar to the CBC mode's output. The only difference is that the order will be flipped. That is, the corrupted bit will be in block B_n and block B_{n+1} is lost entirely. This is because the cipher text in CFB is OXRed with B_n and then fed into the decryption process to compute B_{n+1}, losing that entire block.
4. **OFB:** The OFB mode will lose only the corrupted bit. This is because the cipher text is only used in the XOR process, it is never fed into the decrypt process.

Task 5.2

We now perform the experiment for each of the encryption modes. The process is described as follows. For each encryption mode, we encrypt the same input file, flip a bit in the 55 byte of the cipher text, and decrypt to analyze the results. Figure 14 displays the first three paragraphs of the original input to each encryption mode.

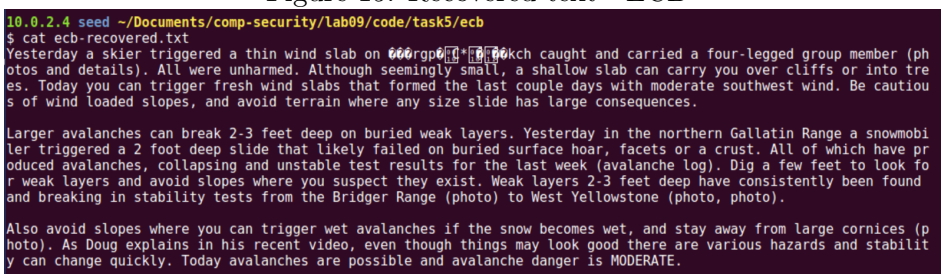
Figure 14: Original input text

```
10.0.2.4 seed ~/Documents/comp-security/lab09/code/task5
$ cat input.txt
Yesterday a skier triggered a thin wind slab on Beehive Peak which caught and carried a four-legged group member (ph
otos and details). All were unharmed. Although seemingly small, a shallow slab can carry you over cliffs or into tre
es. Today you can trigger fresh wind slabs that formed the last couple days with moderate southwest wind. Be cautiou
s of wind loaded slopes, and avoid terrain where any size slide has large consequences.

Larger avalanches can break 2-3 feet deep on buried weak layers. Yesterday in the northern Gallatin Range a snowmobi
ler triggered a 2 foot deep slide that likely failed on buried surface hoar, facets or a crust. All of which have pr
oduced avalanches, collapsing and unstable test results for the last week (avalanche log). Dig a few feet to look fo
r weak layers and avoid slopes where you suspect they exist. Weak layers 2-3 feet deep have consistently been found
and breaking in stability tests from the Bridger Range (photo) to West Yellowstone (photo, photo).

Also avoid slopes where you can trigger wet avalanches if the snow becomes wet, and stay away from large cornices (p
hoto). As Doug explains in his recent video, even though things may look good there are various hazards and stabilit
y can change quickly. Today avalanches are possible and avalanche danger is MODERATE.
```

ECB

Figure 15 shows the recovered text when the ECB encryption mode was used. As expected, the entire block with the corrupted cipher text bit is now lost. Most of the characters in fake block are now unintelligible.

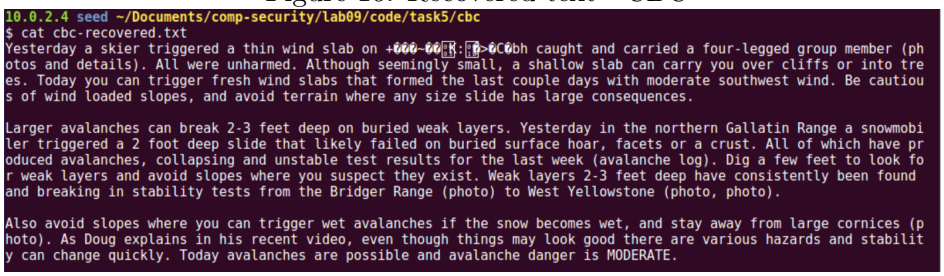
Figure 15: Recovered text - ECB



CBC

Figure 16 shows the recovered text when the CBC encryption mode was used. This mode did not perform as predicted. Only the block with the corrupted bits was affected, while it was expected that an additional bit in the following block would be different.

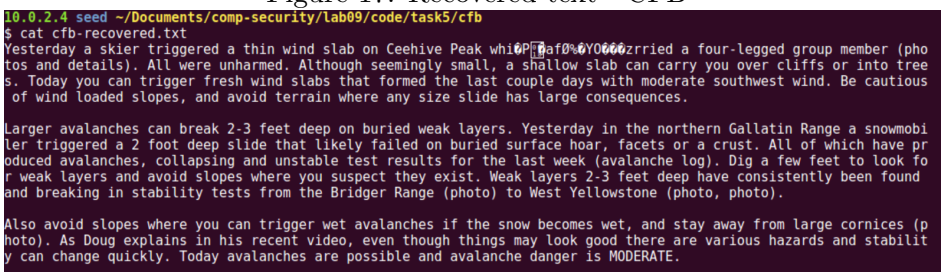
Figure 16: Recovered text - CBC



CFB

Figure 17 shows the recovered text when the CFB encryption mode was used. This mode performed as predicted. The “B” in “Beehive” was changed to a “C” and the entire next block was lost.

Figure 17: Recovered text - CFB



OFB

Figure 18 shows the recovered text when the OFB encryption mode was used. This mode also performed as expected. Only a single character changed, the result of flipping a single bit. This character was a “B” to a “C.”

It should be noted that data lost in the XOR process is easier to recover than during the actual key-encryption process. Since the data is easier to recover, the XOR operation is not as secure as

Figure 18: Recovered text - OFB

```
10.0.2.4 seed ~/Documents/comp-security/lab09/code/task5/ofb
$ cat ofb-recovered.txt
Yesterday a skier triggered a thin wind slab on Ceehive Peak which caught and carried a four-legged group member (ph
otos and details). All were unharmed. Although seemingly small, a shallow slab can carry you over cliffs or into tre
es. Today you can trigger fresh wind slabs that formed the last couple days with moderate southwest wind. Be cautiou
s of wind loaded slopes, and avoid terrain where any size slide has large consequences.

Larger avalanches can break 2-3 feet deep on buried weak layers. Yesterday in the northern Gallatin Range a snowmobi
ler triggered a 2 foot deep slide that likely failed on buried surface hoar, facets or a crust. All of which have pr
oduced avalanches, collapsing and unstable test results for the last week (avalanche log). Dig a few feet to look fo
r weak layers and avoid slopes where you suspect they exist. Weak layers 2-3 feet deep have consistently been found
and breaking in stability tests from the Bridger Range (photo) to West Yellowstone (photo, photo).

Also avoid slopes where you can trigger wet avalanches if the snow becomes wet, and stay away from large cornices (p
hoto). As Doug explains in his recent video, even though things may look good there are various hazards and stabilit
y can change quickly. Today avalanches are possible and avalanche danger is MODERATE.
```

the key-encryption. Although XOR certainly adds a layer of complexity, it is not as difficult to break as the key-encryption.

Task 6

Task 6.1

We now focus on the initialization vector. Encryption algorithms are completely determined by their inputs. By this, we mean that giving the exact same input will always produce the exact same output; there is no random component. Since the initialization vector is an input to the algorithm, this statement applies to it as well. Figure 19 shows this in actions. It shows the hexidecimal contents of three encrypted files. Two of them ("iv1-0.bin" and "iv1-1.bin") were created using separate encrypt commands with the same key and initialization vector. Their contents are identical. The other file ("iv2-0.bin"), was encrypted using the same key but a different initialization vector. It's contents are different than the other two.

Figure 19: Identical inputs means identical outputs

```
10.0.2.4 seed ~/Documents/comp-security/lab09/code/task6/diff
$ xxd -p iv1-0.bin
16a2ba56231210ff3f0dbcebc771e698

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task6/diff
$ xxd -p iv1-1.bin
16a2ba56231210ff3f0dbcebc771e698

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task6/diff
$ xxd -p iv2-0.bin
83e10971cc062eaf9fb3d116abdeb053
```

Task 6.2

Task 6.2 exploits a vulnerability in OFB when the same initialization vector is used more than once. Let's say that an attacker knows the contents of P1 (some plain text) and C1 (the cipher text of P1). If the same initialization vector and the OFB mode are used, the attacker can decrypt any other message sent with the same key, and they don't even need to know the key!

The OFB mode uses the plain text only in the XOR process. This means that crucial information (the output of the key-encryption process) can be deduced just by computing P1 XOR C1. We will call this value KEY-OUTPUT. Task 6.1 showed that if the same initialization vector is used, the same output will be produced. So for any C2 (another encrypted message), the attacker can just compute KEY-OUTPUT XOR C2 and deduce the plain text. This process is performed in Figure 20.

Figure 20: Decoding C2

```
10.0.2.4 seed ~/Documents/comp-security/lab09/code/task6/known
$ python xor.py key-out.txt p1.txt c1.txt
 first: 5468697320697320612061206b6e6f776e206d657373616765210a
second: a469b1c502c1cab966965e50425438e1bb1b5f9037a4c15913
 final: f001d8b622a8b99907b6353e2d2356c1d67e2ce356c3a47819

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task6/known
$ python xor.py p2.txt key-out.txt c2.txt
 first: f001d8b622a8b99907b6353e2d2356c1d67e2ce356c3a47819
second: bf73bcd3509299d566c35b5d450337e1bb175f903fafc15913
 final: 4f726465723a204c61756e63682061206d697373696c65210a
```

We can then convert the hex value "4f726465723a204c61756e63682061206d697373696c65210a" back to text. This gives the message "Order: Launch a missile!"

Task 6.3

Task 6.3 analyzes the chosen plain text attack. This attack works when the initialization vector follows a predictable pattern. An attacker who is trying to figure out what a message was, given that there are a limited number of options can use this attack.

We will narrow our scope even further and say that the attacker knows the CBC mode was used to encrypt the data. Say the attacker wants to know the contents of P1 with corresponding C1 and IV1 (where the attacker knows C1 and IV1). The attacker could construct a new message P2 (with corresponding IV2) and find C2. If C1 and C2 match, the attacker has found the initial input. Since the attacker does not know the key, they cannot actually encrypt P2. They essentially have a block box that encrypts the data using a predictable IV. So the attacker wants to have input subject to the following (equivalent) constraints:

$$P2 \oplus IV2 = P1 \oplus IV1$$

$$P2 = P1 \oplus IV1 \oplus IV2$$

Now the attacker can guess at P1 and use the above formula to compute the necessary input.

We now give an example in Figure 21. A python script was used to run the actual encryption. Note that C1 = "bef65565572ccee2a9f9553154ed9498." Now we know that P1 was either "Yes" or "No." (excluding the period). From there we can use the process described above. The output is shown in Figure 21. In order to provide the correct input to the encryption, we used an entire block. So the python script outputs an extra 16 bytes that we can ignore; we need only focus on the first 16 bytes. With that in mind, we can deduce that the original input was "Yes."

Figure 21: Deducing P1

```
10.0.2.4 seed ~/Documents/comp-security/lab09/code/task6/chosen
$ python enc.py yes-in-hex.txt

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task6/chosen
$ python enc.py no-in-hex.txt

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task6/chosen
$ xxd -p y-out.bin
bef65565572ccee2a9f9553154ed94983402de3f0dd16ce789e5475779ac
a405

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task6/chosen
$ xxd -p n-out.bin
d0680746008a9e91d756e1301d209243fb3699f280364f522168b5a7e4cf
9bad
```

Task 7

Task 7 asks us to find a key that was used to encrypt a cipher text. We know the plain text, the cipher text, and the initialization vector. We also know that the key is an English word fewer than 16 characters and the remaining space is padded with # symbols. We will solve this by writing a program using the Crypto library. The program is displayed in Figure 22.

Figure 22: Python script

```
#!/usr/bin/python3

from Crypto.Cipher import AES
from Crypto.Util import Padding

def decrypt(ciphertext, key, iv):
    # Decrypt the ciphertext
    cipher = AES.new(key, AES.MODE_CBC, iv)
    return cipher.decrypt(ciphertext)

def constructKey(word):
    key_guess = word
    length = len(key_guess)
    # add # symbol
    for i in range(length, 16):
        key_guess += b'#'
    return bytearray(key_guess)

def main():
    # given plain text
    plaintext = "This is a top secret."
    # we know the iv
    iv_hex_string  = 'aabbccddeeff00998877665544332211'
    iv = bytearray.fromhex(iv_hex_string)
    # ciphertext
    ciphertext = bytearray.fromhex("1e479ff8738e2dc5612c00e92782ea811231cbad8a6a9f9f52ff9c9148b9956a")

    # take in file
    fin = open("words.txt")

    key = ''
    decrypted = ''
    for line in fin:
        key_guess = constructKey(line[:-1])
        #print(key_guess)
        if (len(key_guess) == 16):
            dec_guess = decrypt(ciphertext, key_guess, iv)[0:len(plaintext)]
            #print(dec_guess)
            if (plaintext == dec_guess):
                key = key_guess
                decrypted = dec_guess

    print("The key is: " + key)
    print("The decrypted message is: " + decrypted)

                                                                    8,1           Top
```

We now give a quick explanation of the program. Since we know the cipher text, plain text, and initialization vector, all those variables are hard coded into the program. We then read in a file containing English words (possible keys). We construct keys of the appropriate form (length 16 and filling extra space with #) and decrypt to see if recovered text matches the plain text. Eventually a match is found. The output of the program is shown in Figure 23.

Figure 23: Decrypting cipher text

```
10.0.2.4 seed ~/Documents/comp-security/lab09/code/task7
$ python dec.py
The key is: virus##########
The decrypted message is: This is a top secret.

10.0.2.4 seed ~/Documents/comp-security/lab09/code/task7
$
```

Encryption is a highly complex world and there is a lot of room for error. This lab focused on asymmetric encryption and specifically analyzed a number of modes within the Advanced Encryption Standard. Additionally, security holes were discussed.