Lab 1 – Secret-Key Encryption (Seed Labs)

Taylor Adams, CS 542 - Cryptography, tkadams1@crimson.ua.edu

***Abstract*—The goal of the lab is to allow the user to gain a greater understanding of Secret key encryption. This lab demonstrates the use of secure and non-secure encryption algorithms. This lab shows how some encryptions are not secure by having the user take advantage of their flaws to decrypt the messages without the keys or use available information to find the key. I was able to follow each of the task to crack a monoalphabetic cipher using frequency analysis, to determine which encryption modes are secure vs not secure, to create several programs to run known-plaintext attacks, chosen-plaintext attacks, and dictionary attacks. My results display the weaknesses of certain algorithms as described by the lab.**

I. Lab Definition

THE goal of this lab is to give students the opportunity to learn about various encryption algorithms and encryption modes. The lab requires students to be hands on and evokes the critical thinking skills of the student by requiring them to to analyze flaws in encryption algorithms such as using a monoalphabetic cipher to encrypt confidential data. In task 2 this lab also gave the student the opportunity to experiment with different ciphers and encryption modes. Task 3 requires the student to gain an understanding of the differences between the ECB and CBC encryption modes by encrypting an image with both modes. The student then is able to see the flaws within the ECB encryption mode where information can still be made out. Task 4 discusses block ciphers and the need for utilizing padding for them. The student demonstrates which ciphers need padding and which do not based on the encrypting and decrypting while utilizing a flag to keep the padding in the decrypted plaintext. Task 5 discusses the availability of information stored in a corrupted encrypted file. This task allows the user to visualize error propagation throughout some common encryption methods. Task 6 focuses on initial vectors (IV). This task has the student go into detail about the need for secure and random IVs. It discusses attacks that can be used against known IVs. These attacks include the known-plaintext attack and the chosen-plaintext attack. The user is then asked to perform both of these tasks on test data provided by the lab. Task 7 has the user implement a sort of dictionary attack against a provided English word list. The Task 7 ciphertext was provided by Dr. Xiao and we were asked to find the key used in the encryption, which was one of the words in the words.txt file provided. The user was given the plaintext, ciphertext and the IV. The user could then use these values to run the dictionary attack to find the key used for the encryption.

Manuscript received February 18, 2022.

II. Step-By-Step Instructions - Lab Task

A. Task A – Setting up the Virtual Machine

In order to begin this lab, I chose to utilize VirtualBox along with the pre-built VM provided at https://seedsecuritylabs.org/labsetup.html [10]. This VM utilizes Ubuntu 20.04. After downloading the SEED-Ubuntu20.04.vdi file, I proceeded to create a new virtual machine to utilize the image with. I followed the setup instructions provided in the above url to complete the setup of the lab. I have frequently utilized virtualbox in the past, so this setup task was no problem at all.

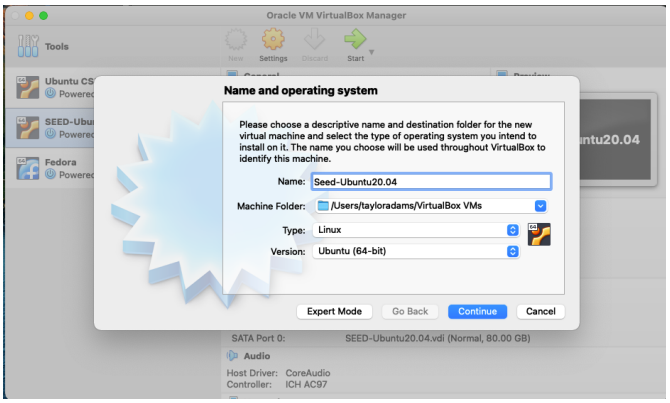


Fig. 1. Creating the Virtual Machine

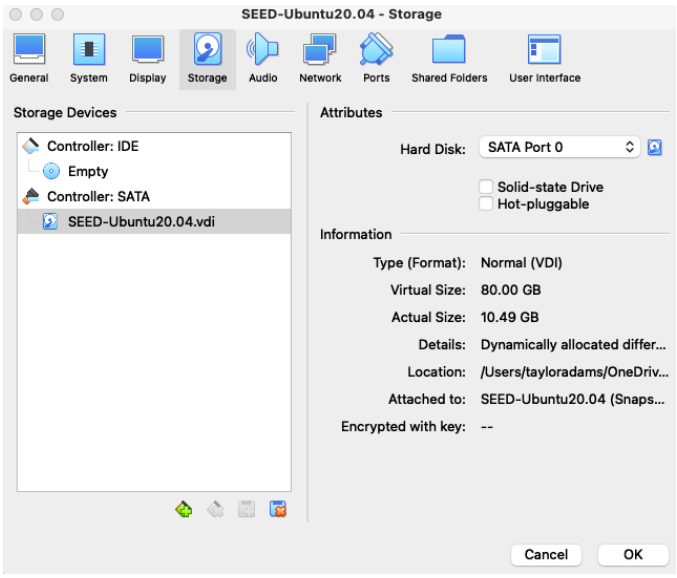


Fig. 1a. Pre-Built VDI file placed in the virtual disk

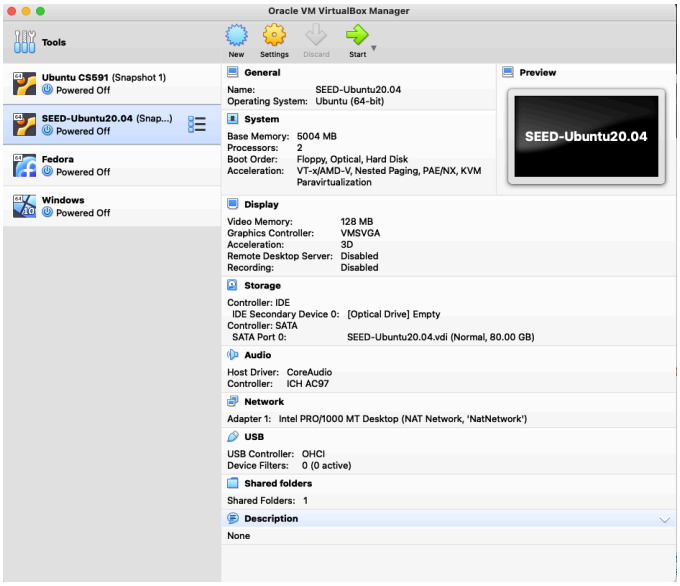


Fig. 1b. SEED Virtual Machine Complete Settings

B. Task 1: Frequency Analysis

For task 1, I was given ciphertext encrypted with a monoalphabetic cipher.

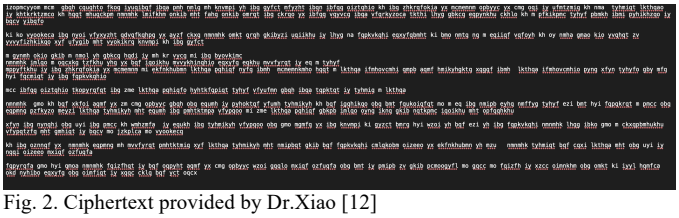


Fig. 2. Ciphertext provided by Dr.Xiao [12]

The goal of this task was to decrypt the ciphertext using frequency analysis of the English language. To begin tackling this problem I read the Wikipedia articles on Frequency Analysis, Bigrams, and Trigrams.

I used the "freq.py" provided by the lab to find the frequency analysis of the provided ciphertext. In the freq.py file I made a slight modification to print all 26 letters of the alphabet, instead of just the top 20. When running the frequency analysis I found the letter q appeared 220 times in the ciphertext. The next highest letter was 'm' with 167 occurrences. Letter 'q' appeared to be equal to 'e' based on the frequency analysis. Following this I moved on to look at the trigram analysis of the ciphertext. 'ibq' stood out with 23 occurrences where the next closest trigram had 12 occurrences. Going off my initial assumption of 'q' == 'e', from Wikipedia the trigram 'the' appears 1.81% where 'and' is the next closest trigram with a frequency of 0.73% [11]. Therefore, I made the assumption that 'ibq' == 'the'. In the ciphertext the following most frequent trigrams were 'qhi' and 'tyh'. Knowing 'i' == 't' I assumed 'qhi' == 'ent' from cross-referencing the Wikipedia trigrams article and the ciphertext frequency analysis. I repeated this pattern of assumption for several more trigrams until I had several letters decrypted. I then wrote a script to decrypt the current letters I knew in the ciphertext. (I made the decrypted letters capitalized to denote what letters had been decrypted.) I then scanned the partially decrypted text and found partially decrypted words such as 'TOuETHEf'. In this case I was able to assume this word was 'together' and assumed 'u' == 'g' and 'f' == 'r'.

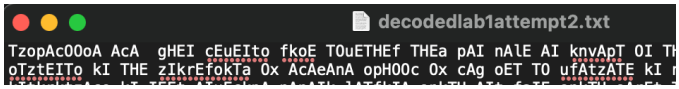


Fig. 3. Snippet from decodedlab1attempt2.txt showing 'TOuETHEf'

In another example I looked at my partially decrypted text and noticed the word 'gHEN'. I assumed 'g' == 'w' and checked both the freq.py output and the Wikipedia frequency analysis article and noticed both 'g' and 'w' lined up similarly in terms of their frequency analysis in the ciphertext and the English language. I repeated the process, several more times, of further decrypting and scanning the text for common English words. This method helped me to decrypt most of the ciphertext, while only having a few wrong assumptions along the way which were easily corrected by noticing when decrypted text did not make sense. After much trial and error, I was able to discover the key and plaintext.

qmihykofbtncpzgvaxeulrdwj' > 'EATNOISRHDMLCUWPYFBGKVXJQ'

Fig. 4. Decryption key

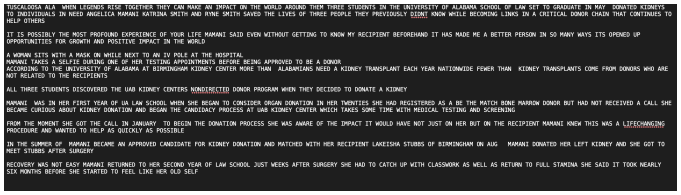


Fig. 5. Decrypted plaintext

C. Task 2: Encryption using Different Ciphers and Modes

In Task 2 I was requested to experiment with several different Ciphers and cipher modes. I began this task by finding a recent news article to copy text from for testing encryption. I stored this data in a file named 'testText.txt'. The ciphers and modes I chose were:

- AES-128-cbc
- AES-128-ecb
- Camilla-128-cbc
- DES-ECB

After choosing the ciphers to use I researched the man pages for both openssl ('*man openssl*') and openssl enc ('*man enc*'). After reviewing the man pages I decided to use the openssl enc commands to generate keys and initial vectors for all of the Ciphers chosen. I wrote a Bash script 'Lab1Task2KeyGeneration.sh' which generates the keys and initial vectors. This script then stores the values in a text file 'testText.txt'. The text I used to encrypt was from the article "Alabama Athletics Plans to Move Forward with Phase II of The Crimson Standard" [2].

```
1
2 touch Task2Keys.txt
3
4 #Generate secret key and initial vectors for each cipher
5 printf "\nAES-128-CBC\n" > Task2Keys.txt
6 openssl enc -aes-128-cbc -iter 123 -k secret -P -md sha1 >> Task2Keys.txt
7
8 printf "\nAES-128-ECB\n" >> Task2Keys.txt
9 openssl enc -aes-128-ecb -iter 456 -k secret -P -md sha1 >> Task2Keys.txt
10
11 printf "\nCamellia-128-CBC\n" >> Task2Keys.txt
12 openssl enc -camellia-128-cbc -iter 789 -k secret -P -md sha1 >> Task2Keys.txt
13
14 printf "\nDES-ECB\n" >> Task2Keys.txt
15 openssl enc -des-ecb -iter 1112 -k secret -P -md sha1 >> Task2Keys.txt
```

Fig. 6. Key Generation shell File

Following the key generation I created a second bash script to encrypt the 'testText.txt' document. This script runs all of the ciphers I chose and generates the corresponding encrypted data in a .bin format.

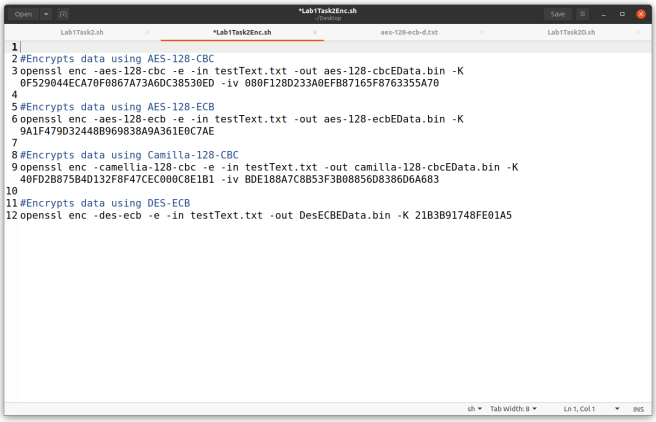


Fig. 7. Encryption shell script

I then opened a terminal and used the cat command to inspect the contents of the .bin files to ensure the encryption occurred. Each file appeared to be encrypted.

Fig. 8. Running the Cat command on the encrypted file utilizing the aes-128-cbc encryption method

After the data was encrypted, I wrote a third script to begin the decryption by changing the '-e' flag from the encryption bash script to '-d' and the input and output file names where changed.

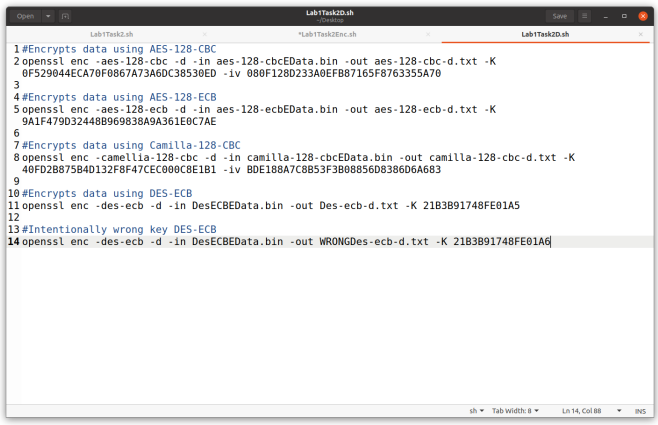


Fig. 9. Decryption shell file

I ran the script and was able to successfully decode the encrypted .bin files for each algorithm. I also ran a test by purposefully running the DES-ECB decryption using an incorrect key. Utilizing this incorrect key demonstrated expect results with the decryption failing.

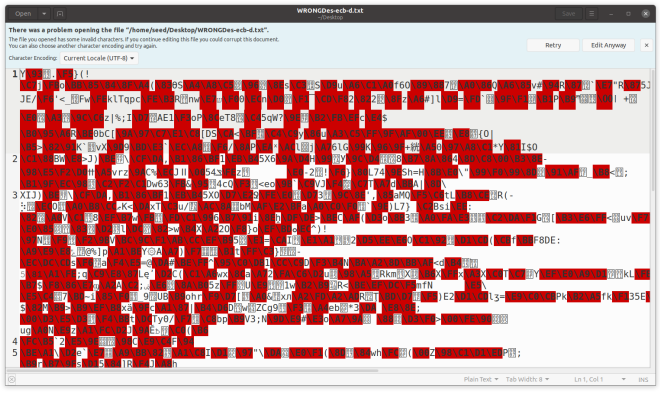


Fig. 10. DES decryption with incorrect key

D. Task 3: Encryption Mode – ECB vs. CBC

For this task I was required to encrypt a given image using cbc and ecb encryption methods. I was subsequently tasked with finding my own image and repeating the process. I began this task by choosing the AES-128 cipher and utilized both the cbc (Cipher Block Chaining) and ebc (Electronic Code Book) modes. I wrote a script to generate new keys for both aes-128-cbc and aes-128-ecb.

```
1 touch Task3Keys.txt
2
3 #Generate secret key and initial vectors for each cipher
4 printf "\nAES-128-CBC\n" > Task3Keys.txt
5 openssl enc -aes-128-cbc -iter 123 -k secret -P -md sha1 >> Task3Keys.txt
6
7 printf "\nAES-128-ECB\n" >> Task3Keys.txt
8 openssl enc -aes-128-ecb -iter 456 -k secret -P -md sha1 >> Task3Keys.txt
```

Fig. 11. Key generation script for aes-128-ebe and aes-128-cbc

Following the key generation, I wrote a second script called "task3enc.sh". This shell script uses the keys generated by "task3keyIV.sh" and encrypts the specified image. Following the encryption, it then repairs the bitmap image header by utilizing the commands provided by the lab and generates the

repaired bitmap images for both encryption modes.

```
#Encrypts data using AES-128-CBC
openssl enc -aes-128-cbc -e -in pic_original.bmp -out pic_cbc.bmp -K 368F70E8784B5363A0871B31B217678E -iv
 B7C04A8744BAECF2397281F5204B3EB6

#Encrypts data using AES-128-ECB
openssl enc -aes-128-ecb -e -in pic_original.bmp -out pic_ecb.bmp -K 9472115647B9075E7F5FFAAE99C2BEAB

#Repair the encrypted bmp images
head -c 54 pic_original.bmp  > header
tail -c +55 pic_ecb.bmp > body
cat header body > ecb_repaired.bmp

tail -c +55 pic_cbc.bmp > body
cat header body > cbc_repaired.bmp
```

Fig. 12. Task3enc.sh shell script for encrypting and repairing .bmp images

Utilizing the above script generated the following results:

Fig. 13. pic_original.bmp provided by lab

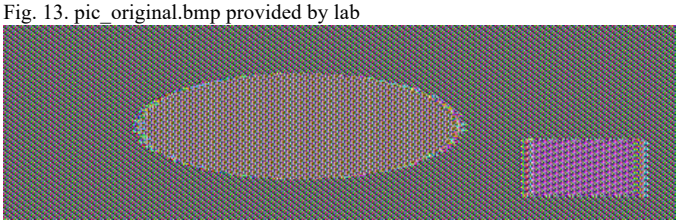


Fig. 14. pic_original.bmp encoded with aes-128-ecb

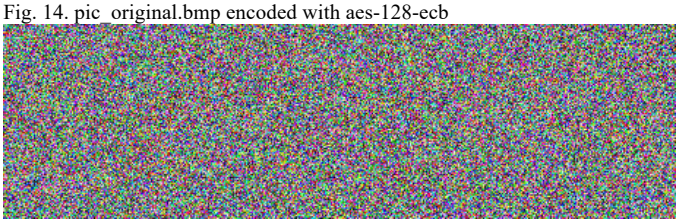


Fig. 15. pic_original.bmp encoded with aes-128-cbc

Fig. 16. UA seal to be encrypted

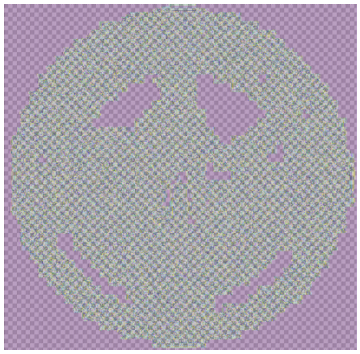


Fig. 17. UA logo encrypted with aes-128-ecb

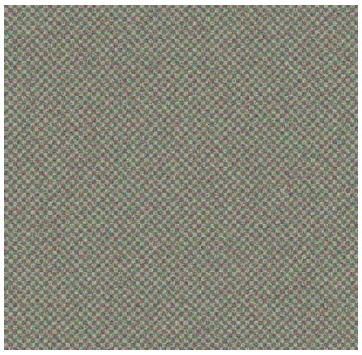


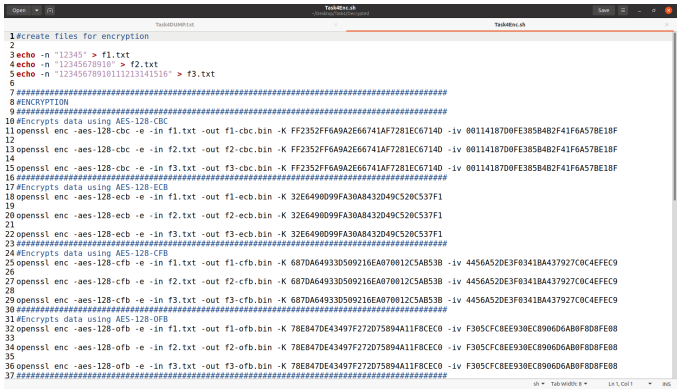
Fig. 18. UA logo encrypted with aes-128-cbc

Observations of task 3:
After encrypting the images, it can clearly be seen that some of the information is still legible using ecb while the images encrypted in cbc mode appear as random noise. Cbc therefore appears to be a more secure mode of encryption, for bitmap images at least.

E. Task 4: Padding

The goal of this task is to understand how padding works with different modes. Padding is used when the file to be encrypted does not have zero remainder byte size when divided by the number of bytes in a block for the encryption mode. This means padding will need to be used to fill the remaining space for the encryption to work.

I repeated the step from the previous task of creating a shell script to generate keys for each of the encryption modes. AES was chosen as the cipher since it is the most popular cipher in use today. I then made another shell script 'Task4Enc.sh' which creates 3 text files containing 5, 10, and 16 bytes. This script then encrypts the 3 text files using each of the encryption modes: CBC, ECB, CFB, OFB.



```
#create files for encryption

echo -n "12345" > f1.txt
echo -n "1234567890" > f2.txt
echo -n "12345678910111213141516" > f3.txt

################################################################################
#ENCRYPTION
################################################################################
#Encrypts data using AES-128-CBC
openssl enc -aes-128-cbc -e -in f1.txt -out f1-cbc.bin -K FF2352FF6A9A2E66741AF7281EC6714D -iv 00114187D0FE385B4B2F41F6A57BE18F

openssl enc -aes-128-cbc -e -in f2.txt -out f2-cbc.bin -K FF2352FF6A9A2E66741AF7281EC6714D -iv 00114187D0FE385B4B2F41F6A57BE18F

openssl enc -aes-128-cbc -e -in f3.txt -out f3-cbc.bin -K FF2352FF6A9A2E66741AF7281EC6714D -iv 00114187D0FE385B4B2F41F6A57BE18F
################################################################################
#Encrypts data using AES-128-ECB
openssl enc -aes-128-ecb -e -in f1.txt -out f1-ecb.bin -K 32E6490D99FA30A8432D49C520C537F1

openssl enc -aes-128-ecb -e -in f2.txt -out f2-ecb.bin -K 32E6490D99FA30A8432D49C520C537F1

openssl enc -aes-128-ecb -e -in f3.txt -out f3-ecb.bin -K 32E6490D99FA30A8432D49C520C537F1
################################################################################
#Encrypts data using AES-128-CFB
openssl enc -aes-128-cfb -e -in f1.txt -out f1-cfb.bin -K 687DA64933D509216EA070012C5AB53B -iv 4456A52DE3F0341BA437927C0C4EFEC9

openssl enc -aes-128-cfb -e -in f2.txt -out f2-cfb.bin -K 687DA64933D509216EA070012C5AB53B -iv 4456A52DE3F0341BA437927C0C4EFEC9

openssl enc -aes-128-cfb -e -in f3.txt -out f3-cfb.bin -K 687DA64933D509216EA070012C5AB53B -iv 4456A52DE3F0341BA437927C0C4EFEC9
################################################################################
#Encrypts data using AES-128-OFB
openssl enc -aes-128-ofb -e -in f1.txt -out f1-ofb.bin -K 78E847DE43497F272D75894A11F8CEC0 -iv F305CFC8EE930EC8906D6AB0F8D8FE08

openssl enc -aes-128-ofb -e -in f2.txt -out f2-ofb.bin -K 78E847DE43497F272D75894A11F8CEC0 -iv F305CFC8EE930EC8906D6AB0F8D8FE08

openssl enc -aes-128-ofb -e -in f3.txt -out f3-ofb.bin -K 78E847DE43497F272D75894A11F8CEC0 -iv F305CFC8EE930EC8906D6AB0F8D8FE08
################################################################################
```

Fig. 19. Encryption and Decryption file Task4Enc.sh

After encrypting the files the script subsequently decrypts the files into text format with the "-nopad" modifier inserted to keep the padding in the text file after the decryption process completes. After the binary files are decrypted I created a script "HEXDUMP.sh" to dump the hex contents into a text file so the padding can be readable. Without this hexdump the padding appears as illegible characters.

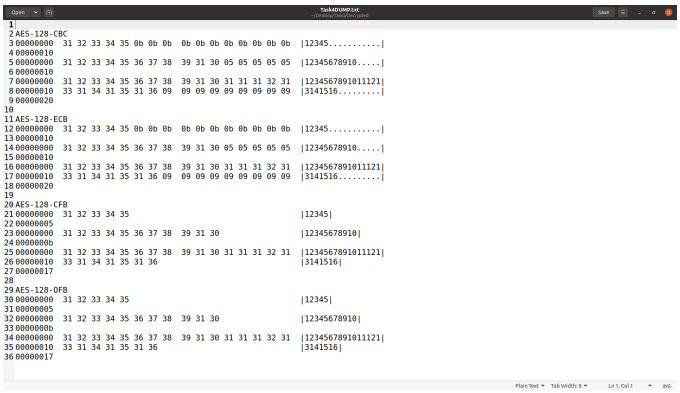


Fig. 20. Hexdumped data

Following decryption, it appears the padding for both the CBC and ECB encryption modes was a “.”. Encryption is required for CBC and ECB encryption because they are block ciphers. Both CFB and OFB modes do not utilize padding. Padding is not used in CFB or OFB modes because they are both stream ciphers. “For other modes of encryption, such as "counter" mode (CTR) or OFB or CFB, padding is not required. In these cases the ciphertext is always the same length as the plaintext, and a padding method is not applicable.”[6].

F. Task 5: Error Propagation – Corrupted Cipher Text

Predictions:

Encryption Mode	Data recovery with corrupted byte
CBC	Most data recoverable with a few corrupted bytes
ECB	Most data recoverable
CFB	Single byte effected
OFB	Single byte effected

After making my predications for each algorithm I modified the shell script I created from task 4 to fit the requirements for task 5. I also created the text document to be encrypted. After encrypting the text file with the CFB, CBC, ECB, and OFB encryption modes I utilized the bless text editor to change the 55 byte of each of the encrypted file. I then wrote a decryption script which decrypted all the files and received my results.

Actual:

Encryption Mode	Data recovery with corrupted byte	Prediction Correct?
CBC	Most data recoverable with a few corrupted bytes	Yes
ECB	Most data recoverable	Yes
CFB	Single byte effected	No
OFB	Single byte effected	Yes

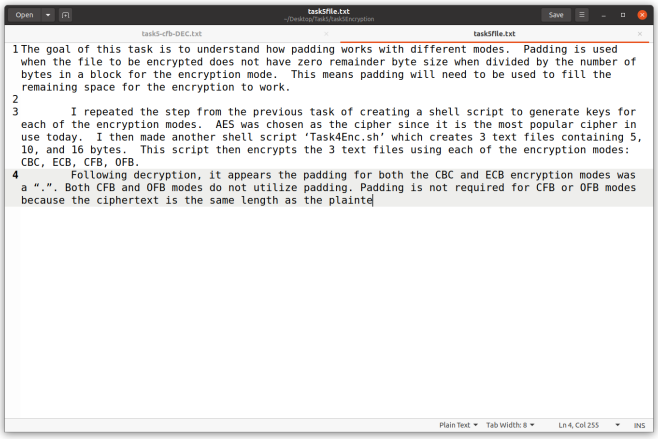


Fig. 21. Original Text

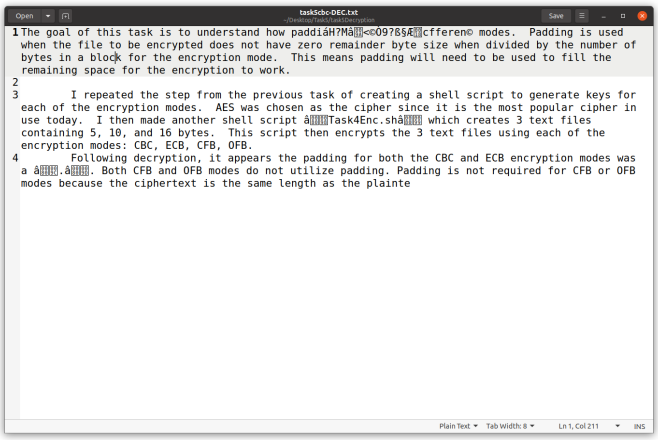


Fig. 22. CBC decrypted text

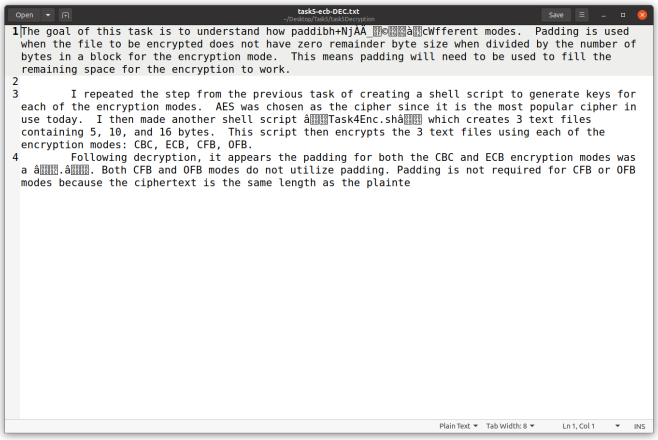


Fig. 23. ECB decrypted text

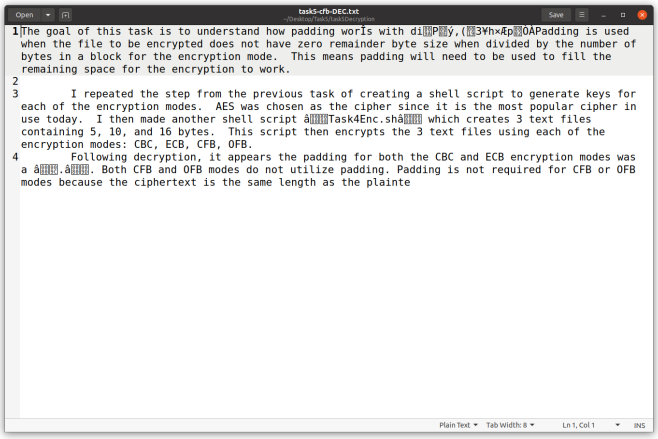


Fig. 24. CFB decrypted text

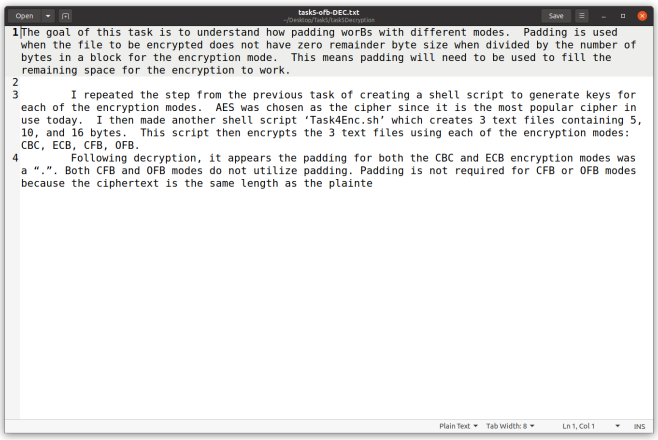


Fig. 25. OFB decrypted text

G. *Task 6: Initial Vector (IV) and Common Mistakes*

1) *Task 6.1. IV Experiment*

Task 6.1 asked to demonstrate using the same IV under the same key when encrypting and then test using a different IV using the same key. This task is to stress the importance of not reusing an IV under the same key. To begin this task I encrypted a text file named 'Task6-1.bin'.

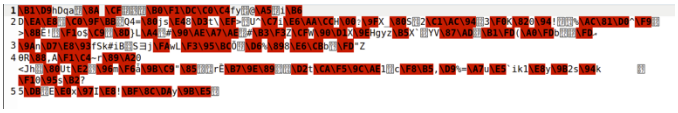


Fig. 26. Task6-1.bin

I then reran the encryption with the same IV, key, and plaintext. Upon encrypting the same file with with the same IV as the previously decrypted file. The exact same ciphertext was generated.

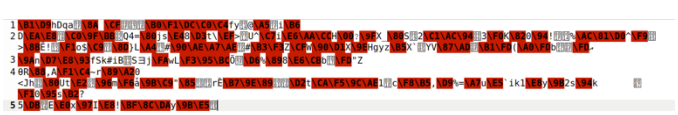


Fig. 27. Task6-1SIV.bin

For the second part of this task I was asked to encrypt the same file with the same key, but with a different IV. I stored this file as 'Task6-1DIV.bin'.

Fig. 28. Task6-1DIV.bin

With a new IV the ciphertext generated looks completely different from when the same IV was used in 'Task6-1.bin' and Task6-1SIV.bin'. The reason it is important to utilize a unique IV each time is because if the same plaintext is encrypted with the same IV and key multiple times patterns can emerge making your encryption less secure to adversaries.

2) *Task 6.2. Common Mistake: Use the Same IV*

For task 6.2 I was asked how much information could be gathered using a known plaintext attack against ciphertext encrypted with CFB. Initially I was having trouble with task 6. I managed to find a youtube video [7] which was very helpful in explaining the concept of a known-plaintext attack. I was also able to utilize this knowledge to help in figuring out task 6-3. I chose to demonstrate this to better understand the known plaintext attack. A known plaintext attack is performed by using the xor operation on a known plaintext and its ciphertext to determine the key used to encrypt it. For this lab I was given a file named "sample_code.py". I modified this file slightly to show P2 from the example provided by the lab.

```python
#!/usr/bin/python3

# XOR two bytearrays
def xor(first, second):
   return bytearray(x^y for x,y in zip(first, second))

MSG   = "This is a known message!"
HEX_1 = "a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159"
HEX_2 = "bf73bcd3509299d566c35b5d450337e1bb175f903fafc159"

HEX_3 = "EE10A1FA65430C9C902CD8CB1A3E6C8B43787529A22850CCDDAC"
HEX_4 = "F50AACEC37105FF09079DDC61D69638BC2D9E189D31B439C03"
# Convert ascii string to bytearray
D1 = bytes(MSG, 'utf-8')

# Convert hex string to bytearray
D2 = bytearray.fromhex(HEX_3)
D3 = bytearray.fromhex(HEX_4)

r1 = xor(D1, D2)
r2 = xor(D2, D3)
r3 = xor(D2, D2)
r4 = xor(r1, D3)
print(r1.hex())
print(r2.hex())
print(r3.hex())
print(r4)
```

Fig. 29. Modified 'sample_code.py'

This example utilized OFB encryption. With the OFB example I found I could decrypt the entire text with a known plaintext attack. The text found was 'Order: Launch a missile!'

```
(base) Taylors-MacBook-Pro:Task Files tayloradams$ /Use
sityofAlabama/School/Grad School/Spring 2022/CS 542/Lab
f001d8b622a8b99907b6353e2d2356c1d67e2ce356c3a478
1b1a0d165253536c0055050d07570f00000c0000080b0000
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
bytearray(b'Order: Launch a missile!')
(base) Taylors-MacBook-Pro:Task Files tayloradams$
```

Fig.30. OFB decrypted example

For CFB testing I reused the same code as before and re-encrypted the same message from the example. I then performed a known plaintext attack against the file I encrypted with CFB and was given the plaintext 'Order: Launch a

\xec\xc4\xe7\xd3\x10Tvq'. This shows I was able to decrypt about two-thirds of the ciphertext encoded with CFB using the known plaintext attack.

```
(base) Taylors-MacBook-Pro:Task Files tayloradams$ /Users/tayloradams/Univer
sityofAlabama/School/Grad School/Spring 2022/CS 542/Lab 1/Task Files/task3.p
ba78c889452a7fbcf10cb3a5754902ab2e1d065ac34f35ed
1b1a0d165253536c0055050d07570f0081a194a071331350de
0000000000000000000000000000000000000000000000000000000000000000
bytearray(b'Order: Launch a \xec\xc4\xe7\xd3\x10Tvq')
(base) Taylors-MacBook-Pro:Task Files tayloradams$
```

Fig .31. CFB decrypted example

3) *Task 6.3. Common Mistake: Use a Predictable IV*

In task 6-3 I was asked to demonstrate a chosen-plaintext attack. We are given bob's secret message which is either "Yes" or "No" without quotes. For this test the encrypted yes or no was '3b1580ae7f7cc2fced76ac8406311d29'. The IV used was '83213d591fae807d3c9eaa029d8b358b' and the next IV was '538c4c7a1fae807d3c9eaa029d8b358b'. I wrote a simple python program which takes in theses variables, adds padding to yes and no, and performs an xor operation on the previous IV and the chosen plaintext (Yes or No).

The chosen-plaintext attack is similar to the known-plaintext attack.

```python

from Crypto.Util.Padding import pad

def add_padding(string):
    return pad(bytes(string, 'utf-8'), 16)

def xor(first, second):
   return bytearray(x^y for x,y in zip(first, second))

def hex_to_bytes(string):
    return bytearray.fromhex(string)

def main():

    #bob_CT = '3b1580ae7f7cc2fced76ac8406311d29'
    bob_Y = 'Yes'
    bob_N = 'No'
    next_IV = '538c4c7a1fae807d3c9eaa029d8b358b'
    prev_IV = '83213d591fae807d3c9eaa029d8b358b'

    bob_Y = add_padding(bob_Y)
    bob_N = add_padding(bob_N)

    prev_IV = hex_to_bytes(prev_IV)
    next_IV = hex_to_bytes(next_IV)

    my_P_yes = xor(xor(bob_Y, prev_IV), next_IV)
    print("Yes\t:", my_P_yes.hex())


    my_P_no = xor(xor(bob_N, prev_IV), next_IV)
    print("No\t:", my_P_no.hex())

main()
```

Fig. 32. 6-3.py

I was able to send the 'Yes' chosen plaintext with the xor operation performed on it with the next initial vector to the oracle provided by the lab and I received bob's cipher text so I knew bob said 'yes' in the previous correspondence.

```
[02/17/22]seed@VM:~/.../encryption_oracle$ nc 10.9.0.80 3000
Bob's secret message is either "Yes" or "No", without quotations.
Bob's ciphertex: 3b1580ae7f7cc2fced76ac8406311d29
The IV used    : 83213d591fae807d3c9eaa029d8b358b

Next IV        : 538c4c7a1fae807d3c9eaa029d8b358b
Your plaintext : 89c8022e0d0d0d0d0d0d0d0d0d0d0d0d
Your ciphertext: 3b1580ae7f7cc2fced76ac8406311d29f259b642054f65b184b0e4242fcb358
1

Next IV        : 304b4c9d1fae807d3c9eaa029d8b358b
Your plaintext :
```

Fig. 33. 6-3.py output

H. *Task 7: Programming using the Crypto Library*

In task 7, I was tasked with utilizing the Crypto Library to decrypt a given ciphertext. The following information was known:

- Ciphertext
- Initial Vector (IV)
- The key is a word in the English language less than or equal to 16 characters in length.
- The key may utilizing a padding composed of the '#' character
- AES-128-CBC was the encryption used

Because I am much stronger in Python than in C, I chose to utilize Python 3.9.5 for task 7. I began by searching for the equivalent Crypto library in Python as to the Crypto library in C. I came across pyCrypto read the documentation [4] and imported the AES cipher [1] into to my python file 'task7.py'. Following the import I copied the IV(hex), plaintext, and the ciphertext (hex) from blackboard and created variables for each of them in my python file. I also utilized the word list provided by the lab.

The code opens the 'words.txt' file. Then it enters a while loop which updates the keyword to test. It ensures the current keyword in the loop is of the proper length. It then converts the key and IV into binary format for the AES encryption methods to use as variables. After the data format conversions the program runs the 'AES.new()' [1] method which takes in the key, Encryption Mode, and Initial value. The program then runs the 'AES.encrypt' [1] method which takes in the plaintext in binary format. (The plaintext must have the pad method surrounding it for CBC encryption.)

```python
AES_enc = AES.new(key, AES.MODE_CBC, IV_bytes)
#encrypt the plaintext
#print(AES.block_size) = 16
ciphertext = AES_enc.encrypt(pad(text, AES.block_size))
```

Fig. 34. AES methods

Following the encryption of the plaintext the ciphertext is converted to hexadecimal and checked against the known ciphertext. If the known ciphertext and the generated ciphertext match the program prints out the key used to generate that ciphertext along with both the known ciphertext and the generated ciphertext for visual comparison. The program then breaks the initial while loop and completes. After running the program with the following values:

- IV_hex = '0102030405060708090a0b0c0d0e0f

- plaintext = 'This is CS542 class Lab 1 task 7.'
- ciphertext_hex_to_find = '8befe2fe4733ce57e2884cef864f66ed1e7c8659e9b031e3b87e951427b10213675873dff380fe0862e9068bdf43eb7e'

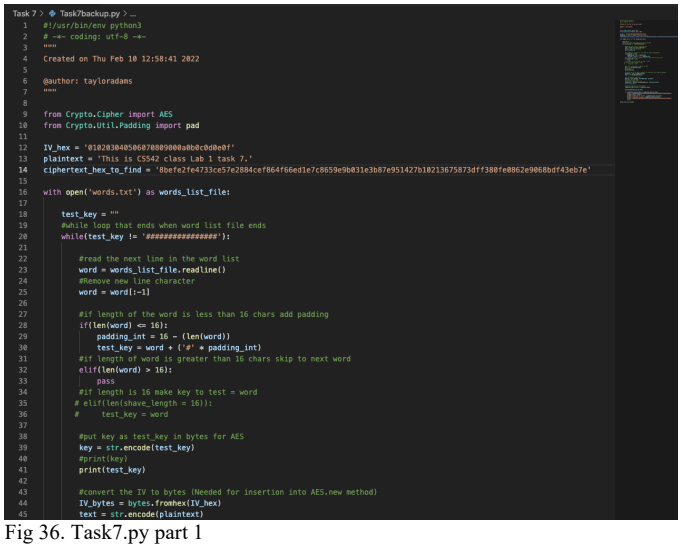
The keyword found to be used in the encryption was **adrenaline**.

```
##########################################################

Key: adrenaline
Ciphertext to find      : 8befe2fe4733ce57e2884cef864f66ed1e7c8659e9b031e3b87e951427b10213675873dff380fe0862e9068bdf43eb7e
Ciphertext Found        : 8befe2fe4733ce57e2884cef864f66ed1e7c8659e9b031e3b87e951427b10213675873dff380fe0862e9068bdf43eb7e

##########################################################
(base) TayJorgs-MBP:Files tayloradams$
```

Fig. 35. Keyword adrenaline found after running the program



```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Feb 10 12:58:41 2022

@author: tayloradams
"""

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

IV_hex = '0102030405060708090a0b0c0d0e0f'
plaintext = 'This is CS542 class Lab 1 task 7.'
ciphertext_hex_to_find = '8befe2fe4733ce57e2884cef864f66ed1e7c8659e9b031e3b87e951427b10213675873dff380fe0862e9068bdf43eb7e'

with open('words.txt') as words_list_file:

    test_key = ""
    #while loop that ends when word list file ends
    while(test_key != '###############'):

        #read the next line in the word list
        word = words_list_file.readline()
        #Remove new line character
        word = word[:-1]

        #if length of the word is less than 16 chars add padding
        if(len(word) <= 16):
            padding_int = 16 - (len(word))
            test_key = word + ('#' * padding_int)
        #if length of word is greater than 16 chars skip to next word
        elif(len(word) > 16):
            pass
        #if length is 16 make key to test = word
        # elif(len(shave_length = 16)):
        #     test_key = word

        #put key as test_key in bytes for AES
        key = str.encode(test_key)
        #print(key)
        print(test_key)

        #convert the IV to bytes (Needed for insertion into AES.new method)
        IV_bytes = bytes.fromhex(IV_hex)
        text = str.encode(plaintext)
```

Fig 36. Task7.py part 1

```
        #convert the IV to bytes (Needed for insertion into AES.new method)
        IV_bytes = bytes.fromhex(IV_hex)
        text = str.encode(plaintext)

        #set encryption algorithm
        AES_enc = AES.new(key, AES.MODE_CBC, IV_bytes)
        #encrypt the plaintext
        #print(AES.block_size) = 16
        ciphertext = AES_enc.encrypt(pad(text, AES.block_size))
        #print(ciphertext)

        #convert ciphertext to hex format
        ciphertext_hex_to_test = ciphertext.hex()

        print(ciphertext_hex_to_test)

        if (ciphertext_hex_to_test == ciphertext_hex_to_find):
            print("\n##########################################################")
            print(f"\nKey: {word}")
            print(f"Ciphertext to find\t: {ciphertext_hex_to_find}")
            print(f"Ciphertext Found\t: {ciphertext_hex_to_find}")
            print("\n##########################################################")
            break

words_list_file.close()
```

Fig 37. Task7.py part 2

III. Discussions

This lab has allowed me to gain a much greater understanding of modern ciphers and their encryption modes. Not only how they work, but also their known flaws and vulnerabilities. This lab also taught about different attacks methods such as the known plaintext attack and the chosen plaintext attack. Task 7 allowed me to utilize the Crypto libraries within python and gave me a greater understanding on how to implement encryption into my programming. Having not only learned but executed on this knowledge will be a very valuable skill moving forward. Hopefully in future professional positions I will be able to implement the knowledge learned in the lab in real world projects.

1) Lab Difficulties

This lab overall was very challenging. In the past I have not had to implement cryptography in my projects as most of the time potential use cases have been provided by default. Task 1 was difficult to get started, but once I began finding patterns it became much easier to decrypt.

Task 6 and 7 were also challenging as they required a lot more critical thinking then the previous task did. Task 6 also recommend we use the textbook chapter 21, but this book was not required for the course and I did not have access to that chapter. I was able to get a better understanding of the known-plaintext attack in task 6.2 by finding a youtube video explaining it [7]. Task 7 required me too find the necessary libraries to be able to complete this task. In order to complete this I had to import the libraries into my anaconda version which initially gave me issues installing. I ran into many runtime errors creating the program for task 7 which caused me to spend a lot of time debugging the issues. One major hurdle was ensuring the padding was the correct length for the AES encryption.

IV. Conclusion

I was able to learn a lot from this lab and correlate the information learned in class with real world applications. By completing this lab, I now have a much better grasp of the content taught in class lecture. This lab along with our textbook should allow me to gain a foothold on cryptography and the challenges of its implementation.

References

[1] "AES — PyCryptodome 3.14.1 documentation." https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html (accessed Feb. 13, 2022).

[2] "Alabama Athletics Plans to Move Forward with Phase II of The Crimson Standard," University of Alabama Athletics. https://rolltide.com/news/2022/2/3/general-alabama-athletics-plans-to-move-forward-with-phase-ii-of-the-crimson-standard.aspx (accessed Feb. 04, 2022).

[3] "Classic modes of operation for symmetric block ciphers — PyCryptodome 3.14.1 documentation." https://pycryptodome.readthedocs.io/en/latest/src/cipher/classic.html#cbc-mode (accessed Feb. 13, 2022).

[4] "Crypto.Cipher package — PyCryptodome 3.14.1 documentation." https://pycryptodome.readthedocs.io/en/latest/src/cipher/cipher.html (accessed Feb. 13, 2022).

[5] "Generating AES keys and password - IBM Documentation." https://www.ibm.com/docs/en/imdm/11.6?topic=encryption-generating-aes-keys-password (accessed Jan. 26, 2022).

[6] "Padding schemes for block ciphers." https://www.cryptosys.net/pki/manpki/pki_paddingschemes.html (accessed Feb. 08, 2022).

[7] CryptoCat, XOR Known-Plaintext Attack - Twizzty Buzzinezz (Crypto/Reversing) [K3RN3L CTF], (Nov. 17, 2021). Accessed: Feb. 17, 2022. [Online]. Available: https://www.youtube.com/watch?v=jSyLy_SfoyQ

[8] "Bigram," Wikipedia. Nov. 01, 2021. Accessed: Feb. 02, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Bigram&oldid=1053021764

[9] "Frequency analysis," Wikipedia. Jan. 10, 2021. Accessed: Feb. 02, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Frequency_analysis&oldid=999424353

[10] "SEED Project." https://seedsecuritylabs.org/labsetup.html (accessed Jan. 27, 2022).

[11] "Trigram," Wikipedia. Oct. 06, 2021. Accessed: Feb. 02, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Trigram&oldid=1048493768

[12]“Tentative Schedule.”
https://ualearn.blackboard.com/bbcswebdav/courses/20163.202210/CS442-542/Spring2022_CS442_Tentative_Schedule.html (accessed Feb. 18, 2022).